pdp11

processor handbook

pdp11/04/24/34a/44/70

digital

digital processor handbook pdp11/04/24/34a/44/70 1981

DIGITAL facility, Marlboro, Massachusetts

CORPORATE PROFILE

Digital Equipment Corporation designs, manufactures, sells and services computers and associated peripheral equipment, and related software and supplies. The Company's products are used world-wide in a wide variety of applications and programs, including scientific research, computation, communications, education, data analysis, industrial control, timesharing, commercial data processing, word processing, health care, instrumentation, engineering and simulation.

pdp11

processor handbook

pdp11/04/24/34a/44/70

digital

This handbook was designed, produced, and typeset
by DIGITAL's New Products Marketing
using an in-house text-processing system.

PRINTED IN USA EB-19402-20

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 UNIBUS 10

CHAPTER 3 ADDRESSING MODES 22

CHAPTER 4 INSTRUCTION SET 42

CHAPTER 5 PROGRAMMING TECHNIQUES 92

CHAPTER 6 MEMORY MANAGEMENT 134

CHAPTER 7 PDP-11/04, 11/34A 170

CHAPTER 8 PDP-11/24 182

CHAPTER 9 PDP-11/44 216

CHAPTER 10 PDP-11/70 260

CHAPTER 11 PDP-11 FLOATING POINT 306

CHAPTER 12 COMMERCIAL INSTRUCTION SET 344

APPENDIX A UNIBUS ADDRESSES A-1

APPENDIX B INSTRUCTION TIMING B-1

APPENDIX C FLOATING POINT TIMING C-1

APPENDIX D CONVERSION TABLE D-1

INDEX Index-1

PREFACE

With 1980 marking the 10th anniversary of the PDP-11 processor, the PDP-11 has become the largest selling minicomputer ever made. Over 170,000 are currently in use worldwide.

During the past decade, the PDP-11 family has experienced the most extensive development and the greatest range in growth of any preceding PDP family. To cap this significant decade of achievement, the family has been enhanced with several new software products and the introduction of the first fourth-generation mid-range CPU, the PDP-11/44, in early 1980. This system, which gives customers twice the performance of the PDP-11/34a, also provides an easy migration path to the larger, more powerful PDP-11/70.

This year — 1981 — DIGITAL is proud to announce another new PDP-11 fourth generation member — the PDP-11/24. This processor offers several unique features, including an extended 22-bit memory addressing capability, making it the lowest-cost systems oriented CPU from DIGITAL that can address up to a full megabyte of memory. The PDP-11/24 provides performance and functionality similar to the PDP-11/34A, and its floating point and commercial instruction sets allow programming compatibility with the PDP-11/44.

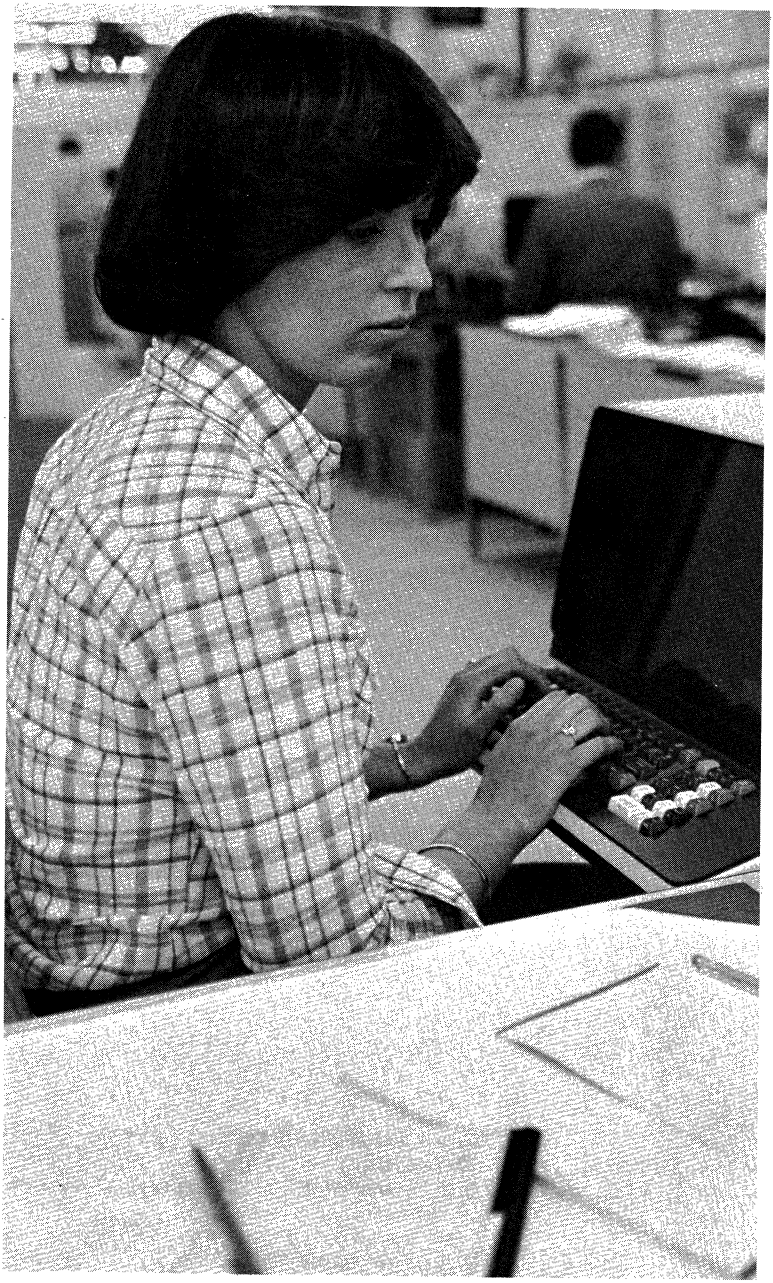
Common to all PDP-11 family members is compatibility, which is inherent in the design of the processors themselves. Programs can be developed on the smallest PDP-11 family member, the PDP-11/03, and with only slight modifications, run on any other PDP-11 system. Peripherals, such as video terminals and line printers, are equally upward and downward compatible in their ability to interface with PDP-11 family members.

This handbook is uniquely divided into four separate processor chapters which discuss the individual and integral features pertinent to the operation of each CPU. It does not attempt to delve into the degree of technicality found in the user documentation delivered with the system.

A complete and current PDP-11 and microcomputer Instruction Set description accompanies this handbook. It is easily identified and immediately accessed by the implementation of a black page tab which runs vertically along the right-hand border of the handbook. Other useful handbook features include chapters on addressing modes, programming techniques, UNIBUS, floating point, and commercial instruction sets, which collectively highlight PDP-11 processor capabilities. An extensively updated chapter on memory management has also been included with this publication. The appendices provide the

most current, accurate, and complete support data and timing to insure the consistency of reference.

At the time this handbook was published, the PDP-11/24 processor instruction set and floating point timings were incomplete. Therefore, they have not been included in this handbook. The next PDP-11 Processor Handbook will furnish these timings.



CHAPTER 1
INTRODUCTION

DIGITAL's PDP-11 processor family is one of the broadest computer product lines in the computer industry. This family consists of microcomputers, minicomputers, system computers, and a powerful multifunction computer—all supported by operating systems, common peripherals and application software.

The processors specifically discussed in this handbook are:

- PDP-11/04
- PDP-11/24
- PDP-11/34A
- PDP-11/44
- PDP-11/70

With DIGITAL'S announcement in 1970 of the first PDP-11, the PDP-11/20, a unique, conceptual change in the computer industry occurred. The PDP-11/20 became the first minicomputer that could interface all system elements—processor, memory and peripherals—to a single, bidirectional, asynchronous bus, called the UNIBUS.

The UNIBUS provides system-to-system compatibility and is a high-speed communications path which links system components and peripheral devises, allowing them to communicate directly without central processor intervention. The UNIBUS, (discussed in detail in chapter 2), and its unique capabilities have provided the flexibility and growth options for the PDP-11 family members discussed in this handbook. Figure 1 illustrates the major categories of PDP-11 processors. Figure 1-2 depicts the block structure of the PDP-11. Figure 1-3 represents the enhancement of performance/functionality versus price with the advent and subsequent development of each succeeding PDP-11 generation. Figure 1-4 compares some of the supported options available for each PDP-11 word processor.

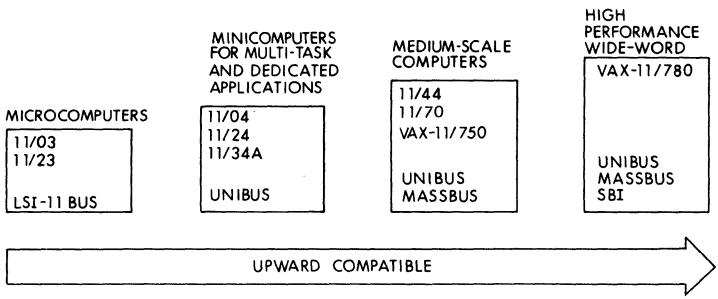


Figure 1-1 Major Categories of PDP-11 Processors

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

PROGRAMMING THE PDP-11

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

The material presented on the PDP-11 instruction set, addressing modes and 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, explains the operating systems and associated software which run on the PDP-11 family of processors. Table 1-1 illustrates these software products.

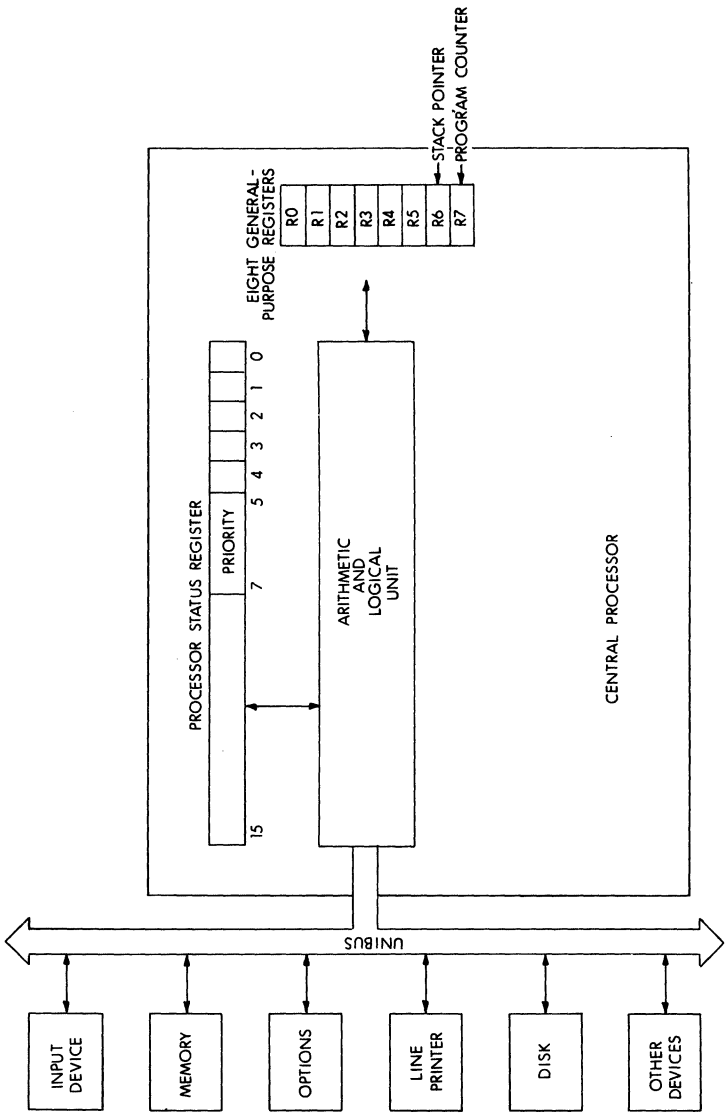


Figure 1-2 PDP-11 Block Structure

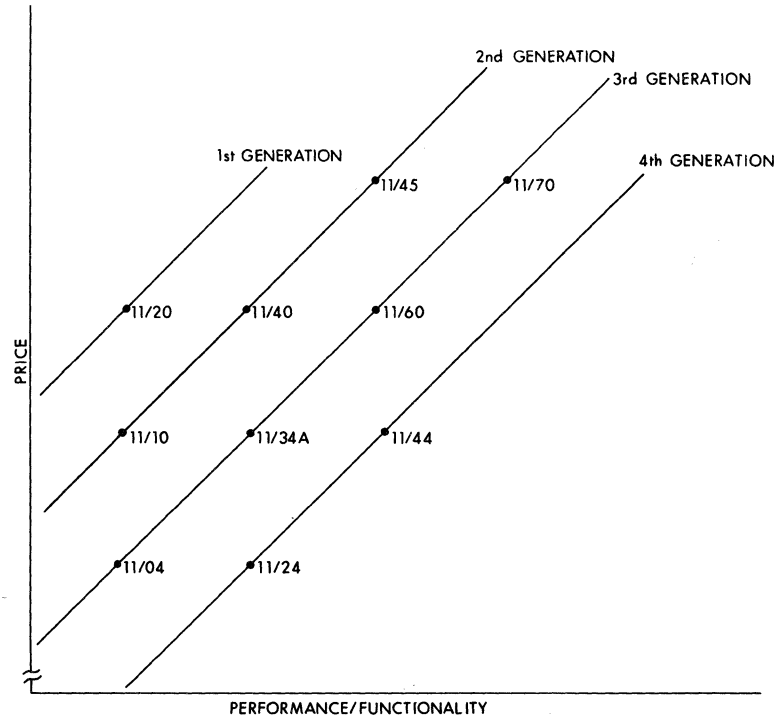


Figure 1-3 PDP-11 Performance/Functionality vs. Price

		SYSTEM OPTIONS				
	PDP	BUS SUPPORT	USABLE MEMORY SUPPORTED	FLOATING POINT PROCESSOR	COMMERCIAL INSTRUCTION SET	CACHE MEMORY
CPU	11/04	UNIBUS	56KB	N/A	N/A	N/A
	11/24	UNIBUS	1MB	OPTIONAL	OPTIONAL	N/A
	11/34A	UNIBUS	248KB	OPTIONAL	N/A	2KB OPTIONAL
	11/44	UNIBUS	1MB	OPTIONAL	OPTIONAL	8KB
	11/70	UNIBUS MASSBUS	4MB	OPTIONAL	N/A	2KB

Figure 1-4 PDP-11 Supported Options Comparison

Table 1-1 PDP-11 Operating Systems

Name	Description
RT-11 and CTS-300	Real-Time Operating System for PDP-11 Processors. A small, single-user foreground/background system that can support a real-time application job's execution in the foreground and an interactive or batch program development job in the background.
DSM-11	DIGITAL Standard MUMPS Operating System for PDP-11 Processors. A small- to large-size timesharing system that offers a unique fast access data storage and retrieval system for large data base processing; originally designed for medical record management and now available for similar data base applications.
RSTS/E and CTS-500	Resource-Sharing Timesharing System/Extended Operating System for PDP-11 Processors. A moderate- to large-size timesharing system that can support up to 63 concurrent jobs, including interactive terminal user jobs, detached jobs, and batch processing.
RSX-11M	Real-Time System Executive Operating System for PDP-11 Processors. A small- to moderate-sized real-time multiprogramming system that can be generated for a wide range of application environments—from small, dedicated systems to large, multipurpose real-time application and program development systems.
RSX-11M-PLUS	Real-Time System Executive Operating System-PLUS for High-end PDP-11 Processors. A large real-time system meant to take advantage of the enhanced hardware features and larger memory available on the PDP-11/44 and PDP-11/70 processors. RSX-11M-PLUS is a superset of RSX-11M.
RSX-11S	Real-Time Multiprogramming Executive Operating System for PDP-11 Processors. A small, execute-only member of the RSX-11 family for dedicated real-time multiprogramming applica-

Name	Description
	tions (requires a host RSX-11M, RSX-11M-PLUS, IAS or VAX/VMS system).
IAS	Interactive Application System for PDP-11 Processors.
	A large multiuser timesharing system, allowing real-time application execution concurrent with time-shared interactive and batch processing.

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

PERIPHERALS

DIGITAL manufactures a full range of peripheral equipment designed to meet specific needs as well as to maintain PDP-11 family compatibility. I/O and storage devices range from cassette tape devices through high-volume disk packs, and from the DECwriter to the intelligent terminals which provide both hard copy and video display. There is a complete spectrum of peripheral devices available to complement the software, and to provide the complete answer to customer needs in all market areas—business, education, industry, laboratory, and engineering.

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

SPECIALIZED SYSTEMS

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

PACKAGED SYSTEMS

DIGITAL's Packaged Systems program offers you the opportunity to purchase a well-defined, pretested, hardware/software system, rather than purchasing the options separately. Packaged systems are fully equipped PDP-11 configurations including operating system, disk storage and loading device. Entry level systems consist of the correct

minimum set of options defined in the *Software Product Description (SPD)* as necessary to run the operating system. Medium and high performance systems have expanded configurations that in some cases substantially exceed minimum SPD requirements. Packaged 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. Packaged systems are priced less than the sum of the individual options. Figure 1-5 illustrates the PDP-11 CPUs which are currently supported by operating systems and available as packaged systems. Those CPUs which are supported by operating systems only, and others that are not supported by operating systems are also shown in figure 1-5, below.

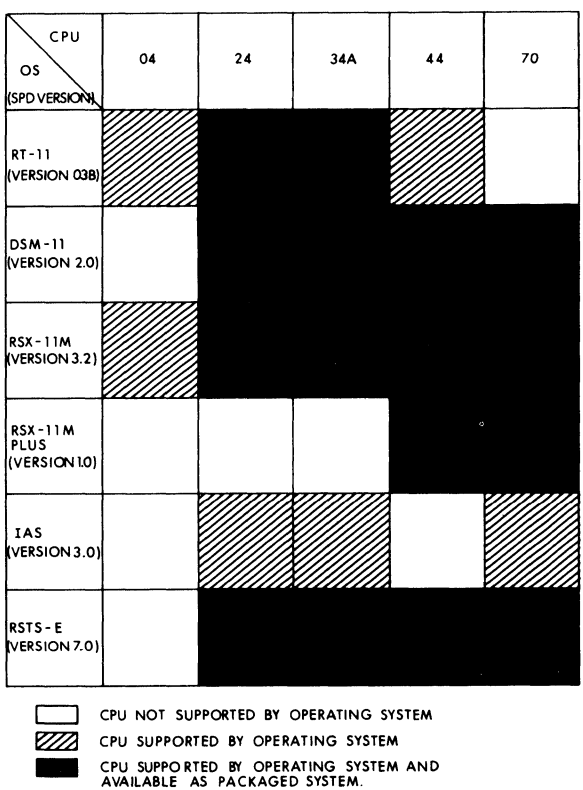


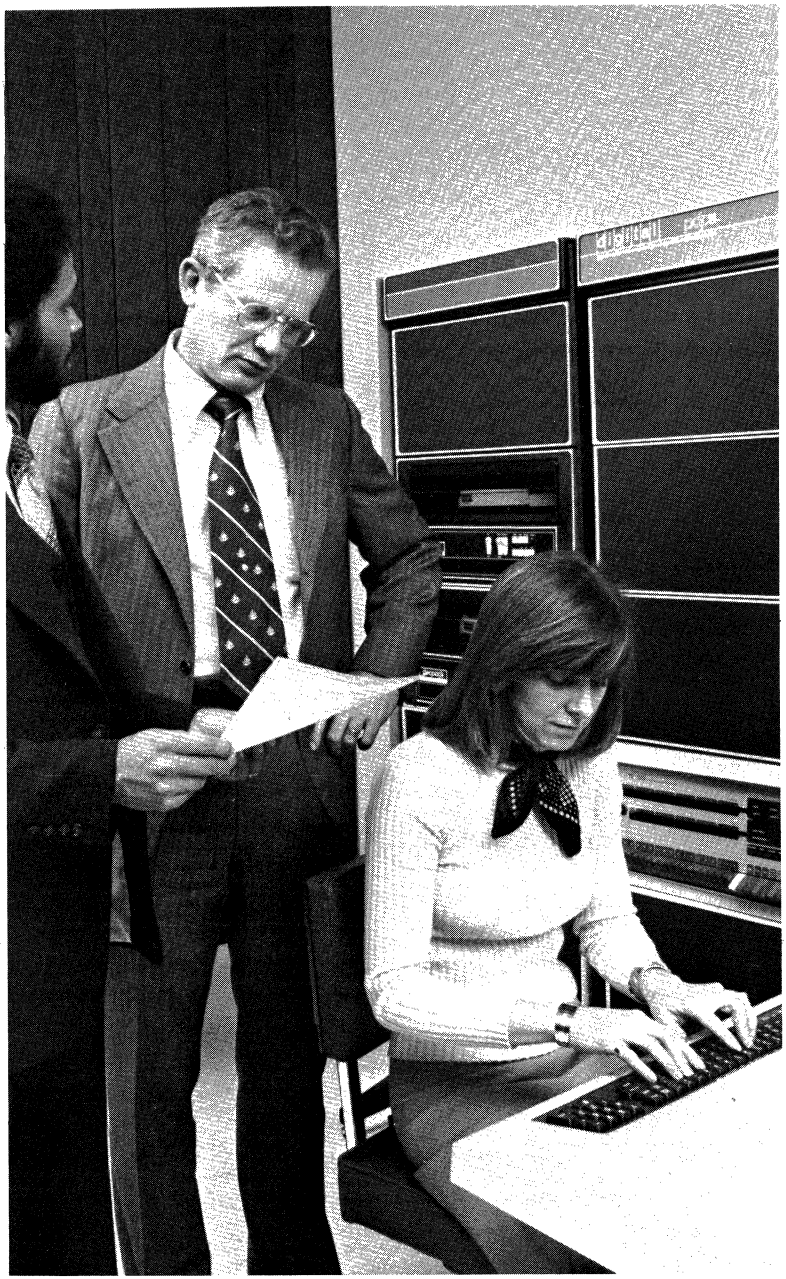
Figure 1-5 Packaged Systems

DOCUMENTATION

DIGITAL offers several levels of documentation describing PDP-11 software and hardware. The PDP-11 Handbook series, which includes the Peripherals Handbook, the Terminals and Communications Handbook, and the Software Handbook, presents an introductory technical level of PDP-11 family information. The hardware user documentation and software tutorial documentation which accompany the delivery of a PDP-11 computer system offer the most detailed levels of information. There are also several books published commercially which discuss the PDP-11 family. Specific topics such as microprogramming are also covered extensively in commercially available books. If you have a specific documentation need, discuss the issue with a DIGITAL sales representative, who will guide you to the appropriate literature.

NUMERICAL NOTATION

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



CHAPTER 2
UNIBUS

The UNIBUS is an outstanding design feature that makes possible the strengths and flexibilities 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, PDP-11/24, PDP-11/34A, PDP-11/44 and PDP-11/70 processors. The UNIBUS was the first data bus in the history of the minicomputer industry to enable devices to send, receive, or exchange data without processor intervention and without intermediate buffering in memory.

PDP-11 ARCHITECTURE AND THE UNIBUS

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

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

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

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

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

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

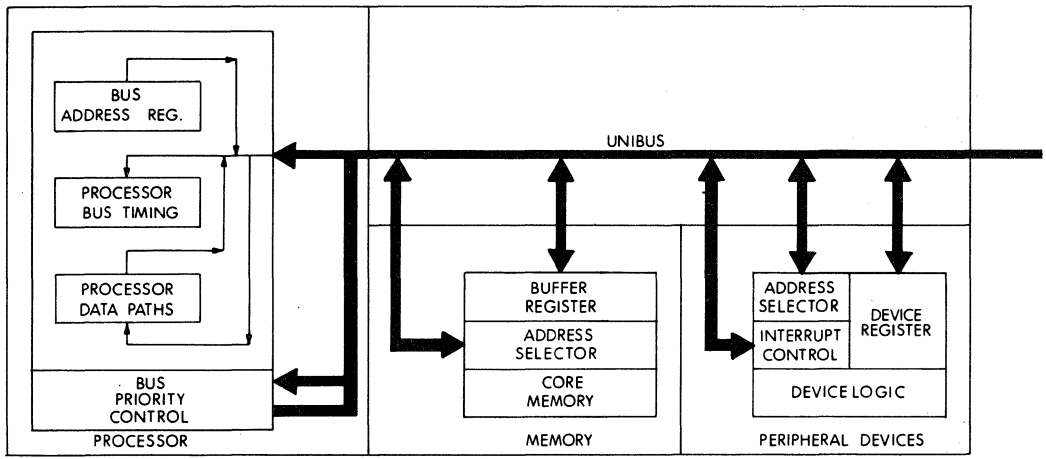


Figure 2-1 UNIBUS

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

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

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

BUS COMMUNICATION

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

USING THE BUS

A device uses the bus if it needs to:

- Request the processor. As a result, the processor stops what it is doing, enters an interrupt service routine, and services the device.
- Transfer a word or byte of data to or from another device, (usually memory), without involving the processor, an NPR (nonprocessor 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: nonprocessor requests (NPRs) or bus requests (BRs).

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

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

Bus Requests

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

Nonprocessor Requests

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

BUS BUSY SIGNAL

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

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

INTERRUPTS

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

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

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

Interrupt Servicing

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

PRIORITY CONTROL

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

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

Bus Request Level

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

Priority Levels

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

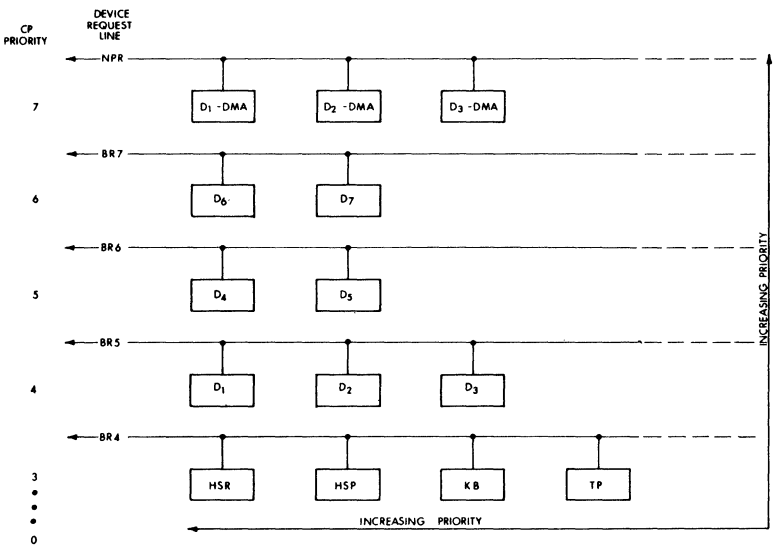


Figure 2-2 Priority Control

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

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

PRIORITY ASSIGNMENTS

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

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

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

CPU Priority Level

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

DATA TRANSACTIONS

There are four types of data transactions:

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

EXECUTION OF DATA TRANSACTIONS

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

- Obtain control of the bus via an NPR.

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

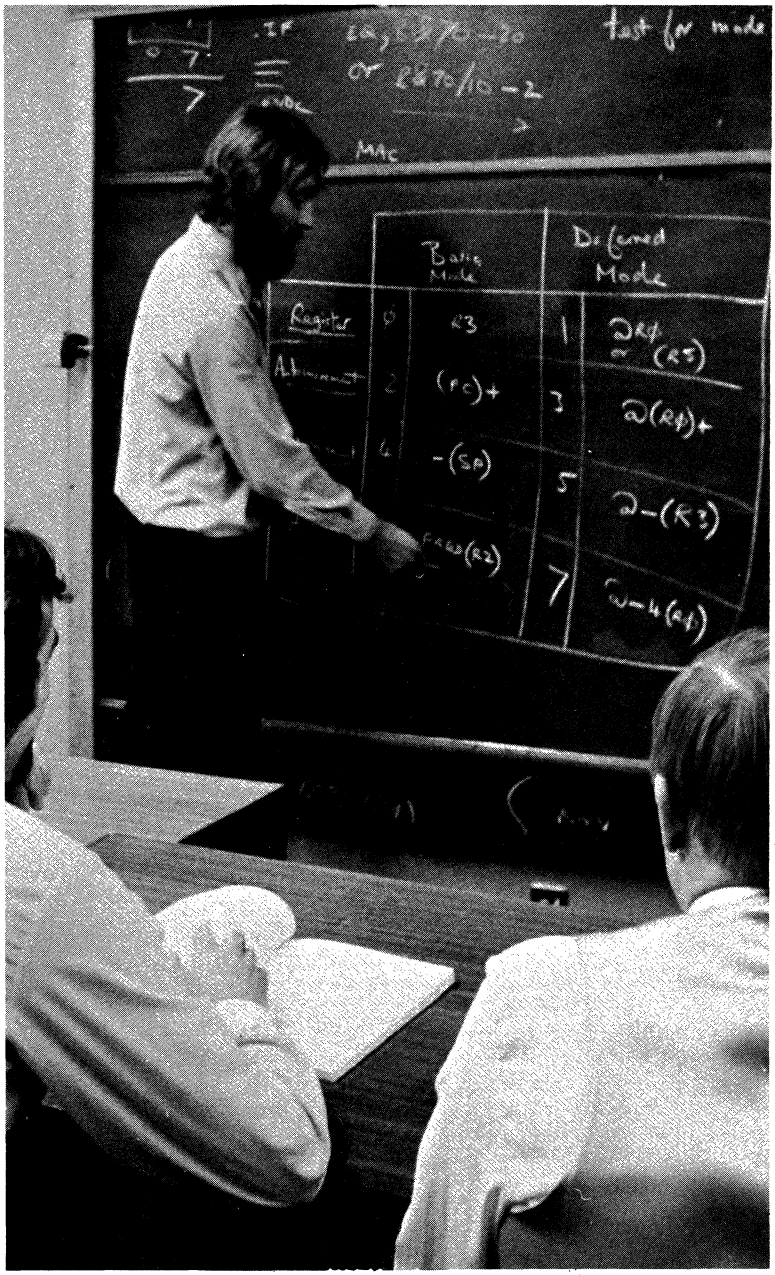
Data transactions between a master and a slave device are synchronized by master sync (MSYNC) and slave sync (SSYN) signals. Below is an example of how these signals are used during a typical DATI transaction:

1. Master selects the slave by addressing it, specifies the type of data transaction, and requests data by asserting MSYN. ("Give me data.")
2. Slave gathers the data and asserts SSYN when the data are 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	UNIBUS Control LOGIC	Asynchronous	Highest priority bus request
NPG	Non-processor Grant	CPU	Next bus master	Asynchronous	Transfers bus control
BR7 through BR4	Bus Request	Any device	UNIBUS Control LOGIC	Asynchronous	Requests bus control
BG7 through BG4	Bus Grant	Memory	Next bus master	After instruction	Transfers bus control
SACK	Selection Acknowledge	Next bus master	UNIBUS Control LOGIC	Response to NPG or BG	Acknowledges grant and inhibits further grants

Signal	Name	Source	Dest.	Timing	Function
BBSY	Bus Busy	Master	All devices	Asserted by bus master	Asserts control of the bus
INTR	Interrupt	Master	UNIBUS Control LOGIC	If control has been gained by a BR (not NPR), INTR asserted after BBSY	Transfers bus control to handling routine in the processor


Register
Deferred Mode
MAC

CHAPTER 3

ADDRESSING MODES

In the PDP-11 family, all operand addressing is accomplished through the eight general purpose registers. To specify the location of data (operand address) one of eight registers is selected with an accompanying addressing mode. Each instruction specifies the:

- Function to be performed (operation code)
- General purpose register to be used when locating the source operand 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. This combination of addressing modes and the instruction set provides the PDP-11 family with a unique number of capabilities. The PDP-11 is designed to handle structured data efficiently and with flexibility. The general purpose registers implement these functions in the following ways, by acting:

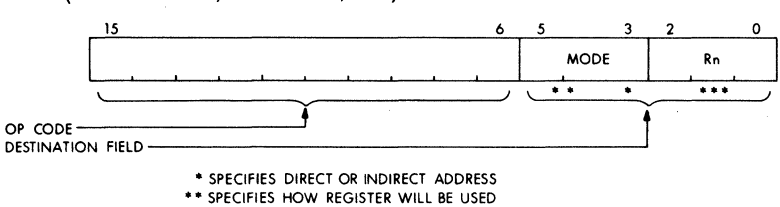
- As accumulators: holding the data to be manipulated
- As pointers: the contents of the register are the address of the operand, rather than the operand itself
- As index registers: the contents of the register are added to an additional word of the instruction to produce the address of the operand; this capability allows easy access to variable entries in a list

Using registers for both data manipulation and address calculation results in a variable length instruction format. If registers alone are used to specify the data source, only one memory word is required to hold the instruction. In certain modes, two or three words may be utilized to hold the basic instruction components. Special addressing mode combinations enable temporary data storage for convenient dynamic handling of frequently accessed data. This is known as **stack addressing**. For a discussion about using the stack, please refer to the Programming Techniques chapter in this handbook. 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. See the Instruction Set chapter for an explanation of the full instruction set and instruction formats.

To illustrate the use of the various addressing modes clearly, the following instructions are used in this chapter:

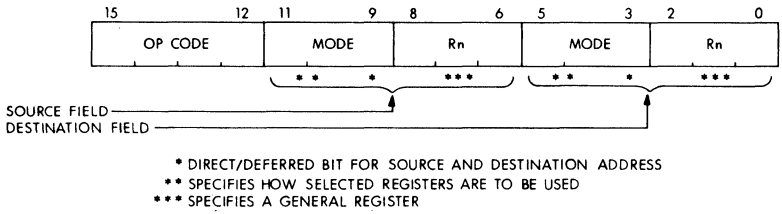
Mnemonic	Description	Octal Code
CLR	Clear (Zero the specified destination)	0050DD
CLRB	Clear Byte (Zero the byte in the specified destination)	1050DD
INC	Increment (Add 1 to contents of destination)	0052DD
INCB	Increment Byte (Add 1 to the contents of destination byte)	1052DD
COM	Complement (Replace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

DD = destination field (6 bits)
SS = source field (6 bits)
( ) = contents of
Single- and double-operand instructions use the following format.
The instruction format for the first word of all single-operand instructions (such as clear, increment, test) is:



Single-Operand Instruction Format

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



Double-Operand Instruction Format

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

The four basic addressing modes are:

- Register
- Autoincrement
- Autodecrement
- Index

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

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

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

The indirect addressing modes are:

- Register deferred
- Autoincrement deferred
- Autodecrement deferred
- Index deferred

Program counter (register 7) addressing modes are:

- Immediate
- Absolute

- Relative
- Relative deferred

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

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

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

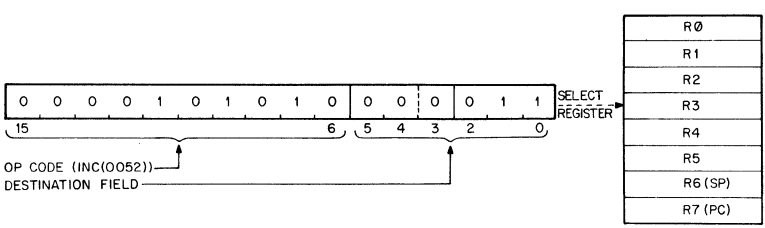
R0 = %0
R1 = %1
R2 = %2

% indicates register definition.

Register Mode Example

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

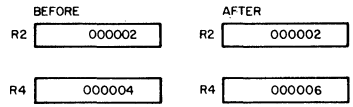
Represented as:



Register Mode Example

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

Represented as:



REGISTER DEFERRED MODE **MODE 1** **(Rn)**

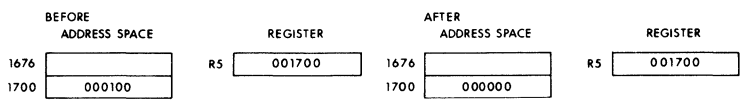
In register deferred mode, the address of the operand is stored in a general purpose register. The address contained in the general purpose register directs the CPU to the operand. The operand is located outside the CPU, either in memory, or in an I/O register.

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

Register Deferred Mode Example

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

Represented as:



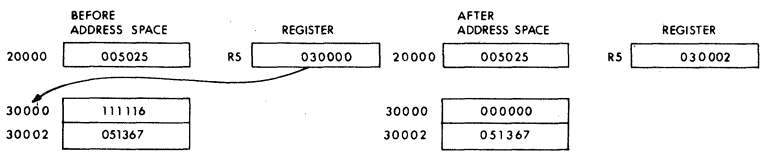
AUTOINCREMENT MODE **MODE 2** **(Rn)+**

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

Autoincrement Mode Example

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

Represented as:



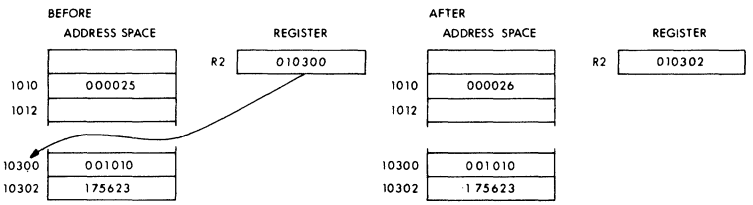
AUTOINCREMENT DEFERRED MODE MODE 3 @(Rn)+

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

Autoincrement Deferred Example

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

Represented as:



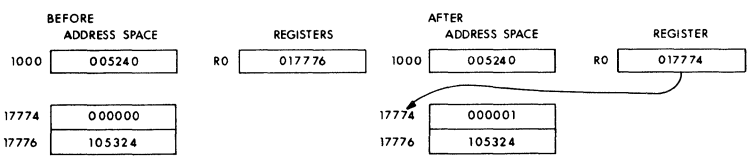
AUTODECREMENT MODE **MODE 4** **−(Rn)**

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

Autodecrement Mode Example

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

Represented as:



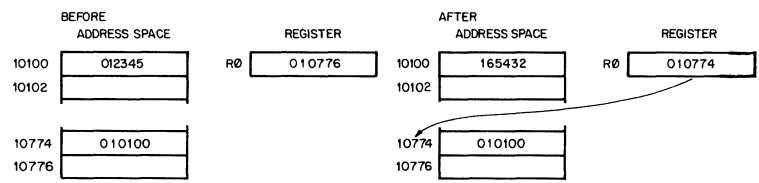
AUTODECREMENT DEFERRED MODE **MODE 5** **@−(Rn)**

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

Autodecrement Deferred Mode Example

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

Represented as:



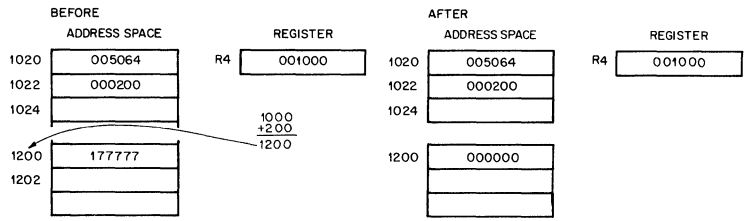
INDEX MODE **MODE 6** **X(Rn)**

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

Index Mode Example

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

Represented as:



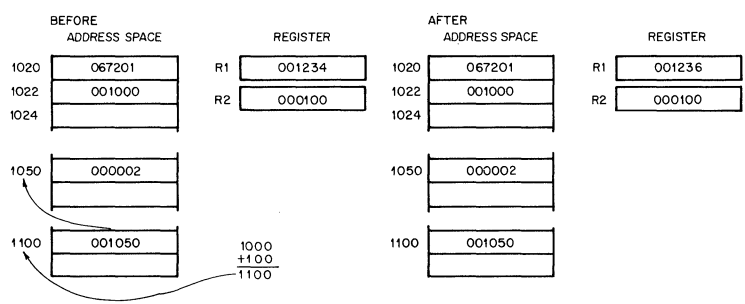
INDEX DEFERRED MODE **MODE 7** **@X(Rn)**

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

Index Deferred Mode Example

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

Represented as:



USE OF THE PC AS A GENERAL REGISTER

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

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

PC IMMEDIATE MODE — MODE 2 — #n

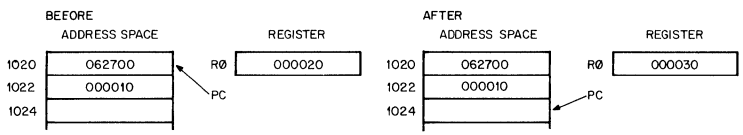
Immediate mode is equivalent to using the autoincrement mode with the PC. It provides time improvements for accessing constant operands by including the constant in the memory location immediately following the instruction word.

PC Immediate Mode Example

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

the operand (the second word of the instruction) before being incremented by two to point to the next instruction.

Represented as:



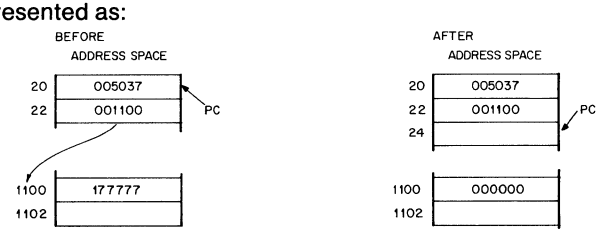
PC ABSOLUTE MODE **MODE 3** **@#A**

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

PC Absolute Mode Example

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

Represented as:



PC RELATIVE MODE **MODE 6** **A**

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

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

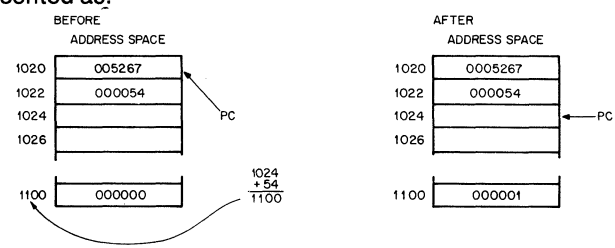
With the relative addressing mode, the address of the operand is always determined with respect to the updated PC. Therefore, when the instruction is relocated, the operand remains the same relative distance away.

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

PC Relative Mode Example

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

Represented as:



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

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

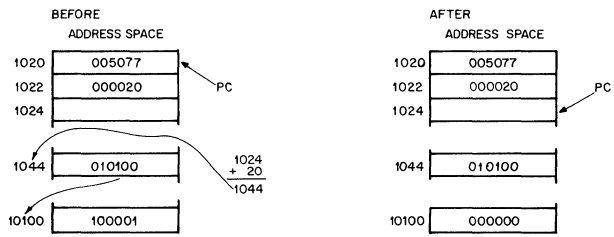
This mode is similar to the relative mode, except that it involves one additional level of addressing to obtain the operand. The sum of the offset and updated PC (PC+4) serves as a pointer to an address. When the address is retrieved, it can be used to locate the operand.

PC Relative Deferred Mode Example

Symbolic	Instruction Octal Code	Description
CLR @A	005077 000020	Adds the second word of the instruc-

tion to PC to produce the address of the address of the operand. Clears operand.

Represented as:



SUMMARY OF ADDRESSING MODES

Basic Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
000	0	Register	Rn	Register contains operand.
010	2	Autoincrement	(Rn)+	Register is used as a pointer to sequential data, then incremented. R0-R5 are incremented by 1 for byte and 2 for word instruction. R6-R7 are always incremented by 2.
100	4	Autodecrement	−(Rn)	Register is decremented and then used as a pointer to sequential data. R0-R5 are decremented by 1 for byte and by 2 for word instructions. R6-R7 are always decremented by 2.

110	6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) is modified. X, the index value, is always found in the next memory location and increments the PC.

Indirect Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
001	1	Register Deferred	@Rn or (Rn)	Register contains the address of the operand.
011	3	Autoincrement Deferred	@(Rn)+	Register is first used as a pointer to a word containing the address of the operand, then incremented (always by 2, even for byte instructions).
101	5	Autodecrement Deferred	@−(Rn)	Register is decremented (always by 2, even for byte instructions) and then used as a pointer to a word containing the address of the operand.

111	7	Index Deferred	@X(Rn)	Value X (the index is always found in the next memory location and increments the PC by 2) and (Rn) are added and the sum is used as a pointer to a word containing the address of the operand. Neither X nor (Rn) is modified.

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

PC Register Addressing Modes

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

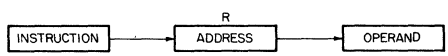
GRAPHIC SUMMARY OF PDP-11 ADDRESSING MODES

General Register Addressing Modes

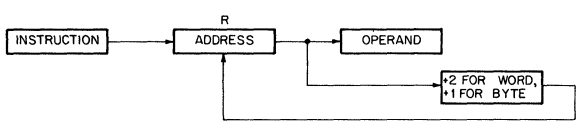
R is a general register, 0 to 7.
(R) is the contents of that register.

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

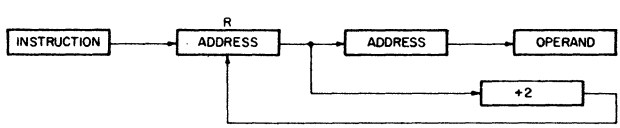
Mode 1 **Register deferred** OPR (R) R contains address.



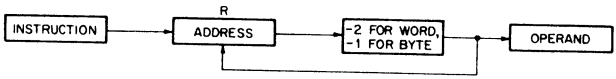
Mode 2 **Autoincrement** OPR (R)+ R contains address, then increment (R). Note that R6 and R7 are always incremented by 2.



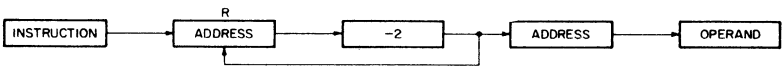
Mode 3 **Autoincrement deferred** OPR @(R)+ R contains address of address, then increment (R) by 2.



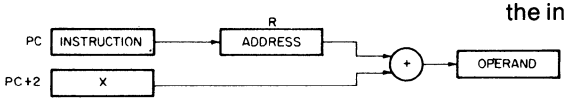
Mode 4 **Autodecrement** OPR −(R) Decrement (R), then R contains address. Note that R6 and R7 are always decremented by 2.



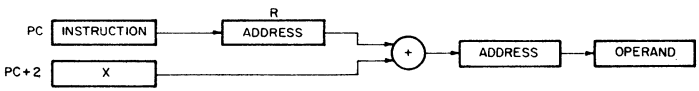
Mode 5 **Autodecrement deferred** OPR @−(R) Decrement (R) by 2, then R contains address of address.



Mode 6 **Index** OPR X(R) (R)+X is address. X is contained in the word following the instruction.



Mode 7 **Index deferred** OPR @X(R) (R)+X is address of address. X is contained in the word following the instruction.

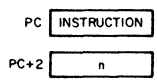


Program Counter Addressing Modes

Register = 7

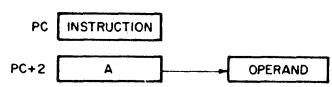
Mode 2 **Immediate** OPR #n

Literal operand n is contained in the word following the instruction.



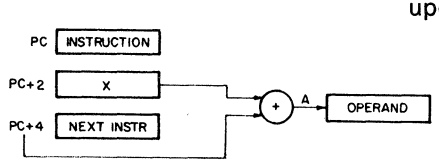
Mode 3 **Absolute** OPR @#A

Address A is contained in the word following the instruction.



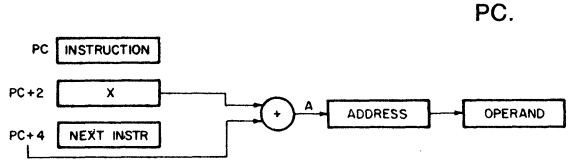
Mode 6 **Relative** OPR A

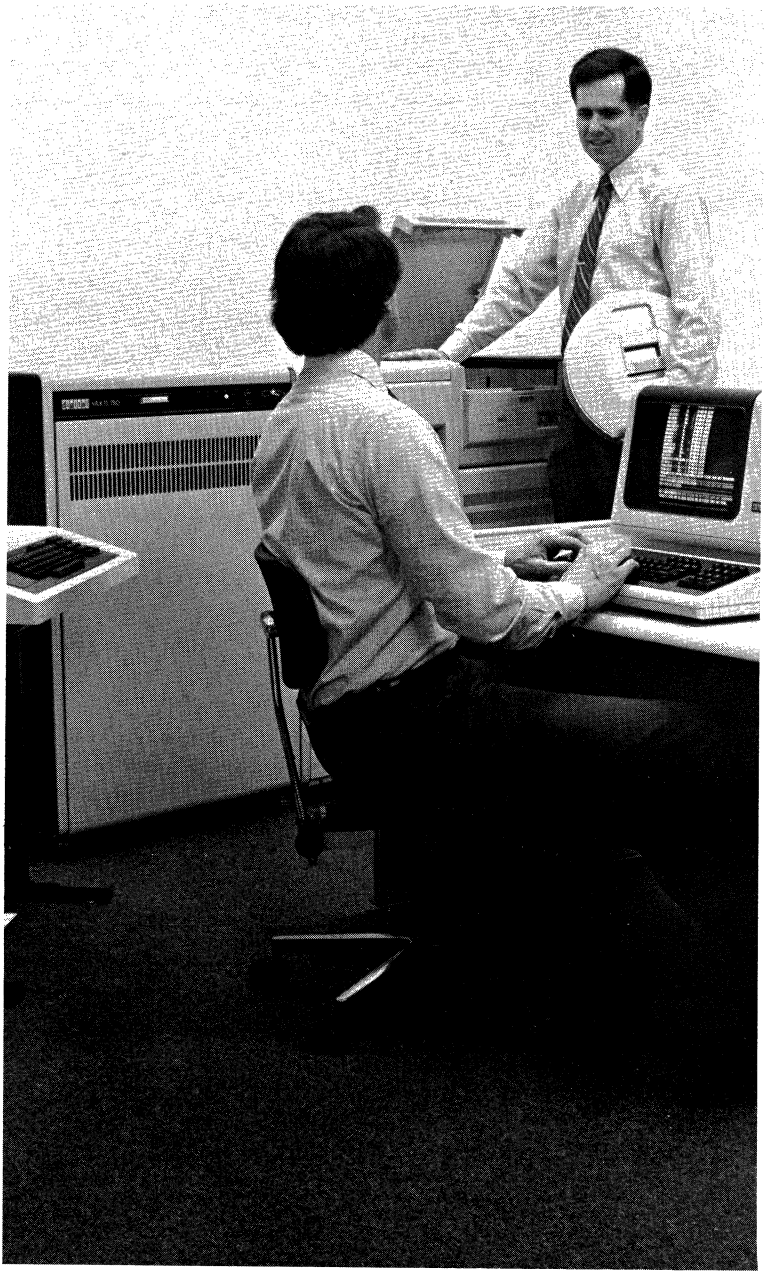
PC+4 + X is address. PC+4 is updated PC.



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 offers a wide selection of operations and addressing modes. To save memory space and to simplify the implementation of control and communications applications, the PDP-11 instructions allow byte and word addressing in both single- and double-operand formats. By using the double-operand instructions, you can 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 this way:

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

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

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

SINGLE-OPERAND INSTRUCTIONS

	Mnemonic	Instruction
General		
	CLR(B)	clear
	COM(B)	1's complement

	INC(B)	increment
	DEC(B)	decrement
	NEG(B)	2's complement (negate)
	TST(B)	test
	NOP	no operation
Shift & Rotate		
	ASR(B)	arithmetic shift right
	ASL(B)	arithmetic shift left
	ROR(B)	rotate right
	ROL(B)	rotate left
	SWAB	swap bytes
Multiple Precision		
	ADC(B)	add carry
	SBC(B)	subtract carry
	SXT	sign extend

Instruction Format

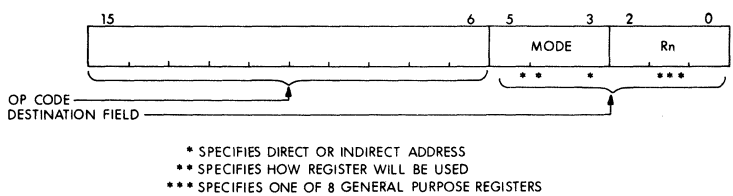


Figure 4-1 Single-Operand Instruction Format

The instruction format for single-operand instructions is:

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

DOUBLE-OPERAND INSTRUCTIONS

	Mnemonic	**Instruction**
General		
	MOV(B)	move source to destination
	ADD	add source to destination
	SUB	subtract source from destination

CMP(B)	compare source to destination
ASH	shift arithmetically
ASHC	arithmetic shift combined
MUL	multiply
DIV	divide

Logical

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

Instruction Format

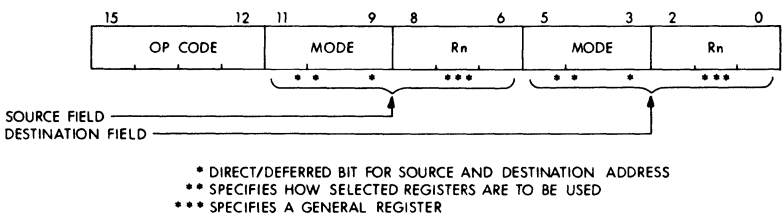


Figure 4-2 Double-Operand Instruction Format

The format of most double-operand instructions, though similar to that of single-operand instructions, has *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 opcode 6, in which case it indicates an ADD or SUBtract instruction.
- Bits 14-12 indicate the opcode, which specifies the operation to be done.
- Bits 11-6 indicate the 3-bit addressing mode field and the 3-bit general register field. These two fields are referred to as the **source** field.
- Bits 5-0 indicate the 3-bit addressing mode field and the 3-bit general register field. These two fields are referred to as the **destination** field.

- Some double-operand instructions (ASH, ASHC, MUL, DIV) must have the destination operand only in a register. Bits 15-9 specify the opcode. Bits 8-6 specify the destination register. Bits 5-0 contain the source field. XOR has a similar format, except that the source is in a register specified by bits 8-6, and the destination field is specified by bits 5-0.

Byte Instructions

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

BRANCH INSTRUCTIONS

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

Instruction Format

- The high byte (bits 15-8) of the instruction is an opcode specifying the conditions to be tested.
- The low byte (bits 7-0) of the instruction is the signed offset value in

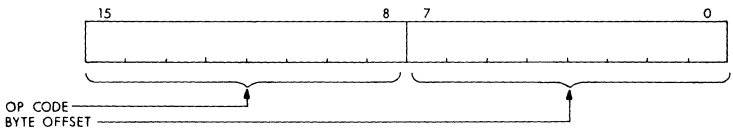


Figure 4-3 Branch Instruction Format

words that determines the new program location if the branch is taken. Thus, program control can be transferred within a range of −128 to +127 words from the updated PC.

JUMP AND SUBROUTINE INSTRUCTIONS

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

Instruction Format

JSR Format

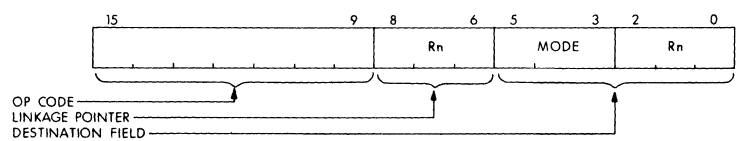


Figure 4-4 JSR Instruction Format

- Bits 15-9 are always octal 004, the opcode for JSR.
- Bits 8-6 specify the link register. Any general purpose register may be used in the link, except R6 (SP).
- Bits 5-0 designate the destination field that consists of addressing mode and general register fields. This specifies the starting address of the subroutine.
- Register R7 (the Program Counter) is frequently used for both the link and the destination. For example, you may use JSR R7, SUBR, which is coded 004767. R7 is the *only* register that can be used for both the link and destination, the other GPRs cannot. Thus, if the link is R5, any register except R5 can be used for one destination field.

RTS Format

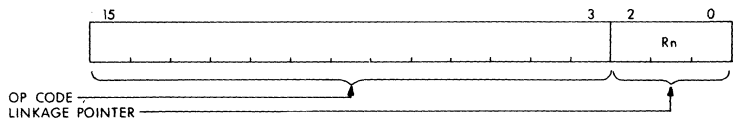


Figure 4-5 RTS Instruction Format

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

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

TRAPS AND INTERRUPTS

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

The three ways to leave a main program are:

- *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 each case, a jump to another program occurs. Once the latter 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
MFPT	move from processor type

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

CONDITION CODE OPERATION

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

The four condition code bits are:

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

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

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

- Adding two numbers equal in magnitude but different in sign
- Comparing two numbers of equal value
- Using the CLR or BIC instruction

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

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

carry of 0 clears the C bit. However, there are exceptions. For example:

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

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

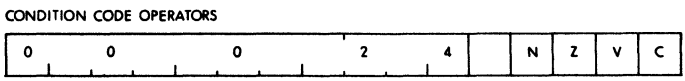


Figure 4-6 Condition Code Operators' Format

Instruction Format

The format of the condition code operators is:

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

corresponding condition code bit is set or cleared depending on the state of the "operator" (bit 4).

EXAMPLES

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

Single-Operand Instruction Example

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

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

Program Description

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

Double-Operand Instruction Example

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

```
INIT:   MOV #600, R0        ;set up source address
        MOV #prtbuf, R1     ;set up destination address
        MOV #76, R2         ;set up loop count
START:  MOVB (R0)+, (R1)+   ;move one character
                            ;and increment
                            ;both source and
```

```
                    ;destination addresses
DEC R2              ;decrement count by one
BNE START           ;loop back if
HALT                ;decremented counter is not
                    ;equal to zero
```

Program Description

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

Branch Instruction Example

NOTE

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

A payroll program has set up a series of words to identify each employee by his badge number. The high byte of the word contains the employee's badge number, the low byte contains an octal number ranging from 0 to 13 which represents his salary. These numbers represent steps within three wage classes to identify which employees are paid weekly, monthly, or quarterly. It is time to make out weekly paychecks. Unfortunately, employee information has been stored in a random order. The problem is to extract the names of only those employees who receive a weekly paycheck. Employee payroll numbers are assigned as follows: 0 to 3 — Wage Class I (weekly), 4 to 7 — Wage Class II (monthly), 10 to 13 — Wage Class III (quarterly).

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

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

START:   CMPB(R0)+,#3

         BHI CONT

STACK:   MOVB (R0),-(R1)

CONT:    INC R0

         CMP #1264, R0

         BHIS START
```

Program Description

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

INSTRUCTION SET

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

SPECIAL SYMBOLS

You will find that a number of special symbols are used to describe

certain features of individual instructions. The commonly used symbols are explained below.

Symbol	Meaning
MN	Maintenance instruction
SO	Single-operand instruction
DO	Double-operand instruction
PC	Program control instruction
MS	Miscellaneous instruction
CC	Condition Code
(x)	Contents of memory location whose address is x
src	Source address
dst	Destination address
tmp	Contents of temporary internal register
←	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	Contents of register
Rv1	Contents of register R if an odd-numbered register is specified. Contents of the register following R if R is an even-numbered register
R, Rv1	32-bit quantity obtained by concatenating R and Rv1
B	Byte
M.P.I.	Most Positive Integer—077777 (word) or 177 (byte)
M.N.I.	Most Negative Integer—100000 (word) or 200 (byte)

NOTE

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

SUMMARY OF PDP-11 INSTRUCTION SET

Basic PDP-11 Instruction Set

ADC	BIT	COM	ROL
ADCB	BITB	COMB	ROLB
ADD	BLE	DEC	ROR
ASL	BLO	DECB	RORB
ASLB	BLOS	EMT	RTI
ASR	BLT	HALT	RTS
ASRB	BMI	INC	RTT
BCC	BNE	INCB	SBC
BCS	BPL	IOT	SBCB
BEQ	BPT	JMP	SCC, SEN, SEZ, SEV, SEC
BGE	BR	JSR	SOB
BGT	BVC	MARK	SUB
BHI	BVS	MOV	SXT
BHIS	CLR	MOVB	SWAB
BIC	CLRB	NEG	TRAP
BICB	CCC, CLN, CLZ, CLV, CLC	NEGBB	TST
BIS	CMP	NOP	TSTB
BISB	CMPB	RESET	XOR
			WAIT

The basic PDP-11 instructions are standard on:

- LSI-11
- LSI-11/2
- PDP-11/03
- PDP-11/23
- PDP-11/24
- PDP-11/34A
- PDP-11/44
- PDP-11/70

The PDP-11/04 implements all basic instructions except for MARK, RTT, SOB, SXT, and XOR.

Extended Integer Instructions (EIS)
ASH
ASHC
DIV
MUL

EIS is standard on:

- PDP-11/23
- PDP-11/24
- PDP-11/34A
- PDP-11/44
- PDP-11/70

EIS is also available as an option on:

- LSI-11
- LSI-11/2
- PDP-11/03

MFPD, MFPI, MTPD, MTPI
Available on PDP-11/23, PDP-11/24, PDP-11/34A, PDP-11/44, PDP-11/70.

MFPS
Available on LSI-11, LSI-11/2, PDP-11/03, PDP-11/23, PDP-11/24, PDP-11/34A.

SPL
Available only on PDP-11/44, PDP-11/70.

CSM
Available on PDP-11/44 only.

MFPT
Available on PDP-11/23, PDP-11/24, PDP-11/44.

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

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

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

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

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BGT Branch if Greater than	PC	003000 PLUS 8- bit offset	PC ← PC + (2 × offset) if Zv(N v V) = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if Z is clear and N equals V. Thus, BGT never branches following an operation that added two negative numbers, even if overflow occurred. In particular, BGT never causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BGT always causes a branch when it follows a CMP instruction operating on a positive source and negative destination. BGT does not cause a branch if the result of the previous operation was 0 (without overflow).
BHI Branch if Higher	PC	101000 PLUS 8- bit offset	PC ← PC + (2 × offset) if C = 0 and Z = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the previous operation causes neither a carry nor a 0 result. This will happen in comparision (CMP) operations as

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				C: unaffected	source nor destination operands are affected. The BIT instruction may be used to test whether any of the corresponding bits that are set in the destination are clear in the source.
BLE Branch if Less than or Equal to	PC	003400 PLUS 8-bit offset	PC ← PC + (2 × offset) if Zv(N v V) = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if Z is set or if N does not equal V. Thus, BLE always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BLE always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLE never causes a branch when it follows a CMP instruction operating on a positive source and negative destination. BLE always causes a

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					ther, BLT never causes a branch when if follows a CMP instruction operating on a positive source and negative destination. BLT does not cause a branch if the result of the previous operation was 0 (without overflow).
BMI Branch if Minus	PC	100400 PLUS 8-bit offset	PC ← PC + (2 × offset) if N = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the N bit and causes a branch if N is set. Used to test the sign (most significant bit) of the result of the previous operation.
BNE Branch if Not Equal	PC	001000 PLUS 8-bit offset	PC ← PC + (2 × offset) if Z = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the Z bit and causes a branch if the Z bit is clear. BNE is the complementary operation to BEQ. It is used to test inequality following a CMP, to test that some bits set in the destination were also set in the source, following a BIT, and generally, to test that

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BVS Branch if V bit Set	PC	102400 PLUS 8- bit offset	PC ← PC + (2 × offset) if V = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of V bit and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.
CLR CLRB Clear	SO	0050DD 1050DD	(dst) ← 0	N: cleared Z: set V: cleared C: cleared	Contents of specified destination are replaced with zeros.
C Clear Selected Condition Code Bits	CC	000240 PLUS 4- bit mask	Clear condition code bits. Selectable combinations of these bits may be cleared together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified. Clears the bit specified by the mask; i.e., bit 0, 1, 2, or 3. Bit 4 is a 0. **Operation:** PSW <3:0> ← PSW <3:0>Λ[~mask <3:0>]		
CCC Clear all Condition	CC	00257	N, Z, V, C ← 0		

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

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

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				V: set if (src) = 0 or if quotient cannot be represented as a 16-bit 2's complement number. R, Rv1 are unpredictable if V is set and C is clear. C: set if divide by 0 is attempted	
EMT Emulator Trap	PC	104000 to 104377	−(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 word (PS) is taken from the word at address 32.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				C: loaded from trap vector	is transmitted in the low byte.
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 and a trap through location 4. (Program control cannot be transferred to a register.) Register

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			subroutine address)		saved in a re-entrant manner on the R6 stack, execution of a subroutine may be interrupted, and the same subroutine re-entered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called nesting) can proceed to any level. JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general purpose registers. JSR, with the PC as the linkage register, saves the use of an extra register. **Note:** If the register specified in the first operand register is autoincremented or autodecremented in the second operand (dst) evaluation,

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
PDP-11/03, and PDP-11/04)					
MFPS Move Byte from PSW (Not available on PDP-11/04, PDP-11/44, and PDP-11/70)	MS	1067DD	(dst) ← PS<7:0> dst lower 8 bits	N: set if PS bit 7 = 1 Z: set if PS <7:0> = 0 V: cleared C: not affected	The 8-bit contents of the PS are moved to the effective destination. If destination is mode 0, PS bit 7 is sign extended through the upper byte of the register. The destination operand is treated as a byte address.
MFPT Move From Processor (Available on	MS	000007	R0<7:0> ← processor model code R0<15:8> ←	N: unaffected Z: unaffected V: unaffected C: unaffected	No source operands are used. The MFPT instructions returns in the low byte of R0 a processor model code (1 on the PDP-11/44, 3 on the

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Previous Data space MTPI Move To Previous Instruction space (Not available on LSI-11, LSI-11/2, PDP-11/03, and PDP-11/04)				V: cleared C: unaffected	bits <15:14> and stores that word into an address in previous space determined by PS bits <13:12>. The destination address is computed using the current registers and memory map. MTPD is identical to MTPI on the PDP-11/24 and PDP-11/34A, and on the PDP-11/44 and PDP-11/70 when D-space is disabled.
MTPS Move Byte To PSW (Available on PDP-	MS	1064SS	PS ← (src)	N: set according to effective src operand 0-3 Z: same as above V: same as above	The eight bits of the effective operand replace the current contents of the PS <7:0>. The source operand address is treated as a byte address. Note that PS bit 4 cannot be

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				C: unaffected	
RESET	MS	000005		N: unaffected Z: unaffected V: unaffected C: unaffected	Sends INIT on the UNIBUS for 10 ms. All devices on the unit are reset to their state at power-up.
ROL ROLB Rotate Left	SO	0061DD 1061DD	(dst) ← (dst) rotate left one place	N: set if the high-order bit of the result word is set (result < 0). Z: set if all bits of the result = 0 V: loaded with the exclusive OR of the N bit and C bit (as set by the completion of the rotate operation). C: set if the high-order bit of the destination was set prior to instruction execution.	Rotates all bits of the destination left one place. The high-order bit is loaded into the C bit of the status word and the previous contents of the C bit are loaded into the low-order bit of the destination.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
RTS Return from Subroutine	PC	00020R	PC ← (reg) (reg) ← (SP)+	N: unaffected Z: unaffected V: unaffected C: unaffected	Loads contents of register into PC and pops the top element of the R6 stack into the specified register. Return from a non-re-entrant subroutine is 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 Return from Interrupt	MS	000006	PC ← (SP)+ PS ← (SP)+	N: loaded from current R6 stack Z: loaded from current R6 stack V: loaded from current R6 stack C: loaded from current R6 stack	This is the same as the RTI instruction (used to exit from an interrupt or trap service routine), the PC and PS are restored (popped) from the processor stack; if the RTI sets the T bit in the PS, a trace trap will occur prior to executing the next instruction) except that it inhibits a

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

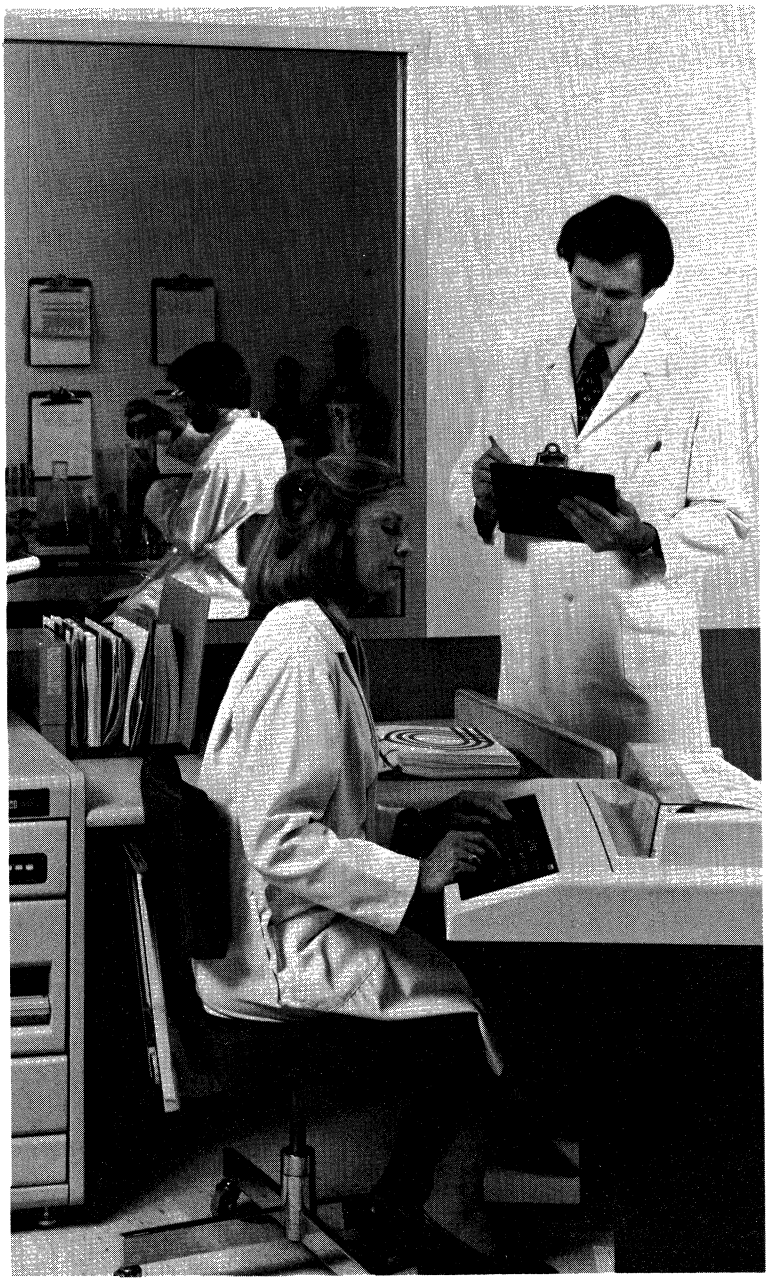
Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Condition Codes			**Operation:** PSW <3:0> ← PSW <3:0> v mask <3:0>		
SCC Set all Condition Codes	CC	000277	N, Z, V, C ← 1		
SEC Set C	CC	000261	C ← 1		
SEN Set N	CC	000270	N ← 1		
SEV Set V	CC	000262	V ← 1		
SEZ Set Z	CC	000264	Z ← 1		
SOB	PC	077R00	R ← R−1	N: unaffected	The register is decremented. If it is

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			(dst)+ ~ (src)+1	arithmetic overflow as a result of the operation, i.e., if the operands were of opposite signs and the sign of the source is the same as the sign of the result. C: set if there is a borrow into the most significant bit of the result, i.e., if (dst) + ~ (src)+1 was less than 2^{16}.	dress. 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 Bytes	SO	0003DD	tmp ← (dst) <7:0> (dst) <7:0> ← (dst) <15:8> (dst) <15:8> ←	N: set if high-order bit of low-order byte (bit 7) of result is set Z: set if low-order byte of result = 0	Exchanges high-order byte and low-order byte of the destination word (destination must be a word address).

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
WAIT Wait for Interrupt	MS	000001		N: unaffected Z: unaffected V: unaffected C: unaffected	Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt. Having been given a WAIT command, the processor will not compete for the bus by fetching instructions or operands from memory. This permits higher transfer rates between device and memory, since no processor-induced latencies will be encountered by bus requests from the device. In WAIT, as in all instructions, the PC points to the next instruction following the WAIT operation. Thus, when an interrupt causes the PC and PS to be pushed onto the stack, the address of the next instruction following the WAIT is saved. The exit from the interrupt routine (i.e., execution of



CHAPTER 5
PROGRAMMING TECHNIQUES

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

POSITION-INDEPENDENT CODE

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

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

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

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

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

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

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

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

A	PC relative mode
@A	PC relative deferred mode

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

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

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

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

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

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

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

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

#N	immediate mode

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

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

```
@#A        absolute mode
```

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

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

EXAMPLES

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

Position-Dependent Code

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

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

Position-Independent Code

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

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

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

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

STACKS

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

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

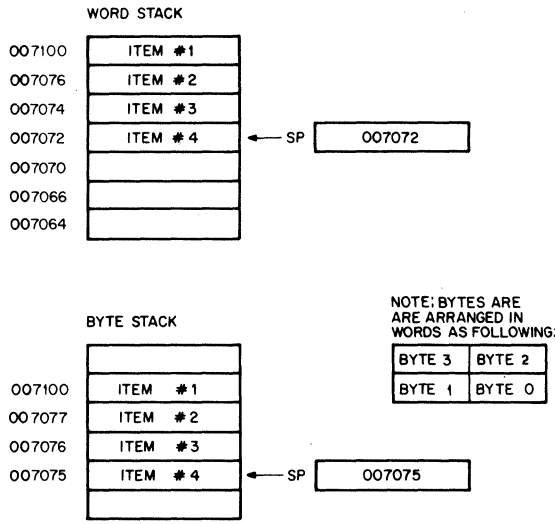


Figure 5-1 Word and Byte Stacks

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

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

Data is thus PUSHed onto the stack.

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

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

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

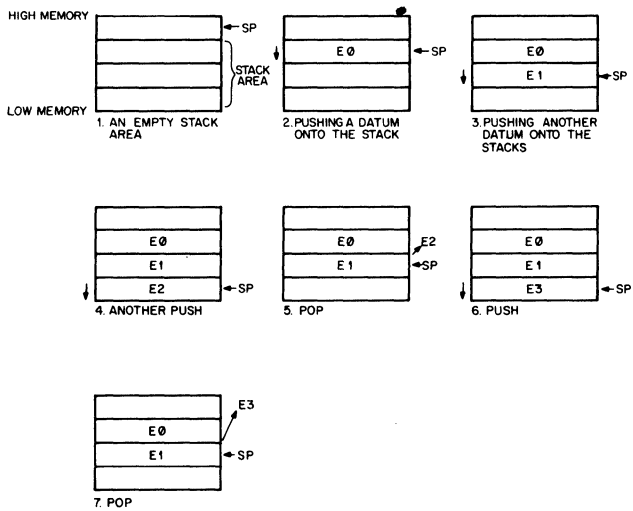


Figure 5-2 Illustration of Push and Pop Operations

Uses for the stack

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

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

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

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

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

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

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

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

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

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

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

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

* Index Constants

OR: Using the Stack

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

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

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

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

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

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

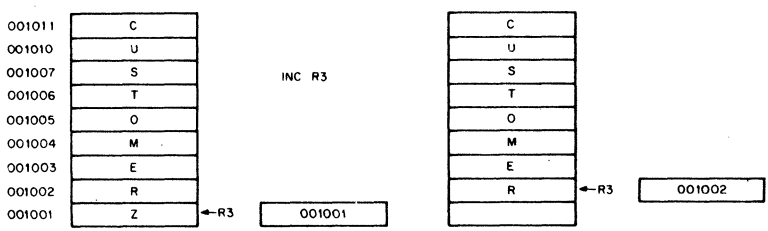


Figure 5-3 Byte Stack Used as a Character Buffer

NOTE

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

DELETING ITEMS FROM A STACK

To delete one item:

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

To delete two items:

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

To delete fifty items from a word stack:

ADD #100.,SP

SUBROUTINE LINKAGE

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

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

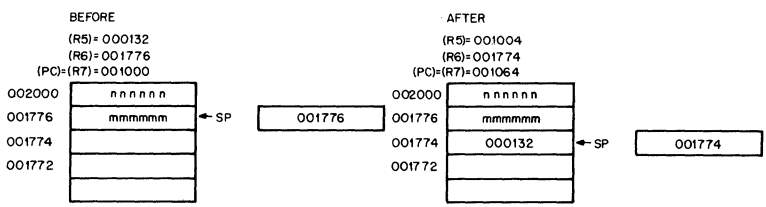


Figure 5-4 JSR

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

Return from a Subroutine

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

Subroutine Advantages

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

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

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

INTERRUPTS

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

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

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

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

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

had been executed.

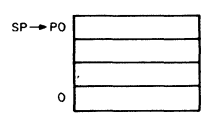
The new contents of the PC and PS are loaded from two preassigned consecutive memory locations which are called "vector addresses." The first word contains the interrupt service routine entry address (the address of the service routine program sequence), and the second word contains the new PS which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

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

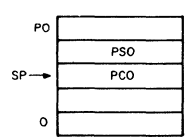
Nesting

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

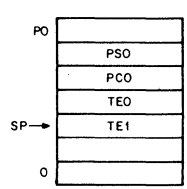
1. Process 0 is running; SP is pointing to location P0.



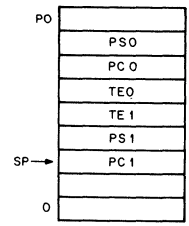
2. Interrupt stops process 0 with PC = PC0, and status = PS0; starts process 1.



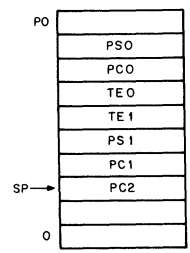
3. Process 1 uses stack for temporary storage (TE0, TE1).



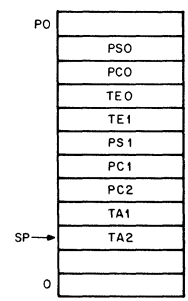
4. Process 1 interrupted with PC = PC1 and status = PS1; process 2 is started.



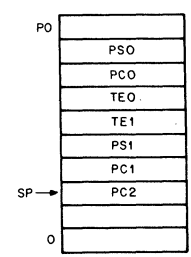
5. Process 2 is running and does a JSR R7,A to subroutine A with PC = PC2.



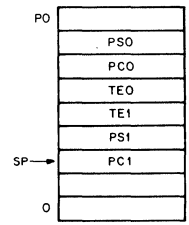
6. Subroutine A is running and uses stack for temporary storage.



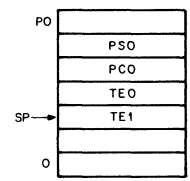
7. Subroutine A releases the temporary storage holding TA1 and TA2.



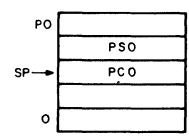
8. Subroutine A returns control to process 2 with an RTS R7; PC is reset to PC2.



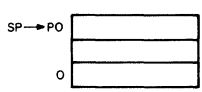
9. Process 2 completes with an RTI instructions (dismisses interrupt) PC is reset to PC1 and status is reset to PS1; process 1 resumes.



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



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



Nested Interrupt Service Routines and Subroutines

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

REENTRANCY

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

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

(ART)

PDP-11 Approach

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

(ART)

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:

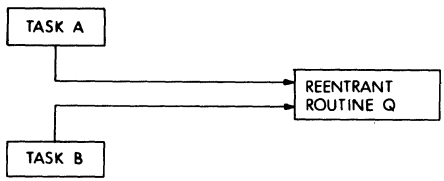


Figure 5-7 Sharing Control of a Routine

- 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 (GPRs) to be saved and replaced by the appropriate values for the task being entered. Therefore, reentrant code should use the GPRs and the stack for any counters, pointers, or data that must be modified or manipulated in the routine.

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

The following should be observed when writing reentrant code:

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

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

Use of Reentrant Code

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

COROUTINES

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

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

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

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

Coroutine Calls

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

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

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

For example:

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

Figure 5-8 Coroutine Example

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

Coroutines Versus Subroutines

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

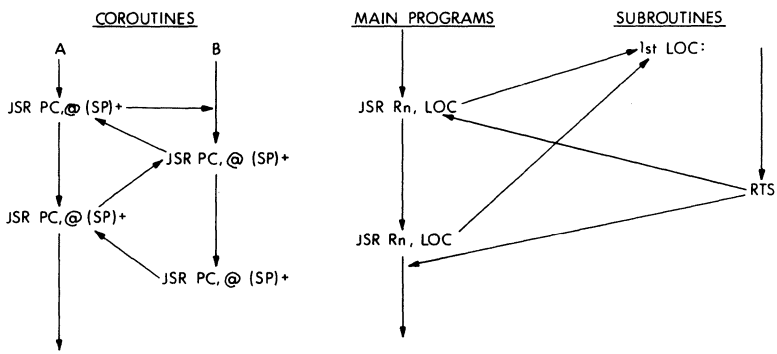


Figure 5-9 Coroutines vs. Subroutines

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

  JSR PC,@(SP)+

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

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

Using Coroutines

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

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

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

Examples

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

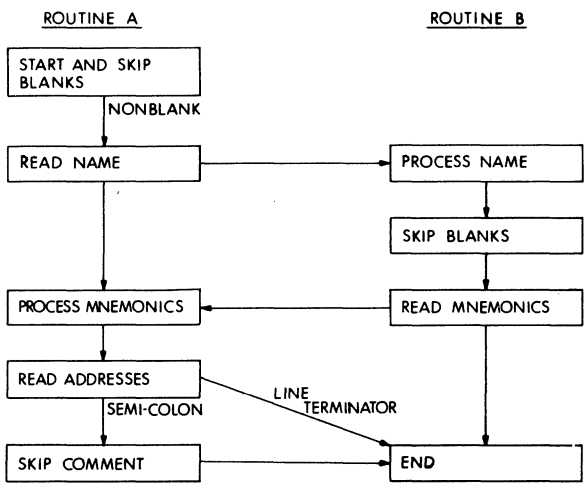


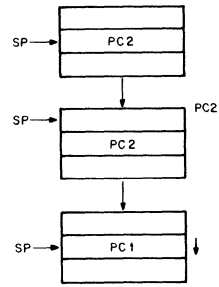
Figure 5-10 Coroutine Path

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

Write 01
Read 11 concurrently
Process 12

then

Write 02
Read 12 concurrently
Process 11



Routine #1 is operating, it then executes:

MOV #PC2,−(R6)
JSR PC,@(R6)+

with the following results:

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

Routine #2 is operating, it then executes:

JSR PC,@(R6)+

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

Figure 5-11 Coroutine Interaction

RECURSION

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

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

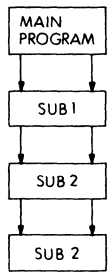


Figure 5-12 Recursive Routine Flow

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

Example:

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

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

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

Uses of Recursion

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

where:

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

and:

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

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

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

Recursion should terminate when N = 1.

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

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

PROCESSOR TRAPS

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

Power Failure

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

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

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

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

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

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

Vector Address and Trap Errors

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

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

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

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

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

Uses of Trap Handlers

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

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

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

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

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

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

and CM3 is defined as

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

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

Summary of PDP-11 Processor Trap Vectors:

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

CONVERSION ROUTINES

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

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

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

Division is typically approached in one of two ways.

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

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

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

   Therefore, to divide 7 by 2:

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

STACK
011             counter
000             quotient
```

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

At the conclusion of the routine:

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

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

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

```
  221
-1000
```

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

```
  221
 -100

  121   count for 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 $10^1 = 2$.

```
  121
```

No lower power, so remainder is 1.

Answer = 022_{10}, remainder 1.

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

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

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

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

The following conditions exist for 6 times 3:

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

After the routine is executed:

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

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

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

```
ASL R0
ASL R0
RETURN
```

If R0 contains 7:

R0 = 111

After execution;

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

ASCII CONVERSIONS

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

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

PROGRAMMING EXAMPLES

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

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

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

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

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

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

                .=500

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

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


VALUES:         .BLK 20.
                0
                .END

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

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

```
            .=500

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

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

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

            .END


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

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

          .TITLE ECHO

          .=1000
```

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

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

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

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

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

        .END
```

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

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


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

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


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

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

        =3000

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


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

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

LOOPING TECHNIQUES

PROGRAM SEGMENTS BELOW USED TO CLEAR A 50.WORD TABLE

1. AUTOINCREMENT (POINTER ADDRESS IN GPR)

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

2. AUTODECREMENT (POINTER AND LIMIT VALUES IN GPR)

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

3. COUNTER (DECREMENTING A GPR CONTAINING COUNT)

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

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

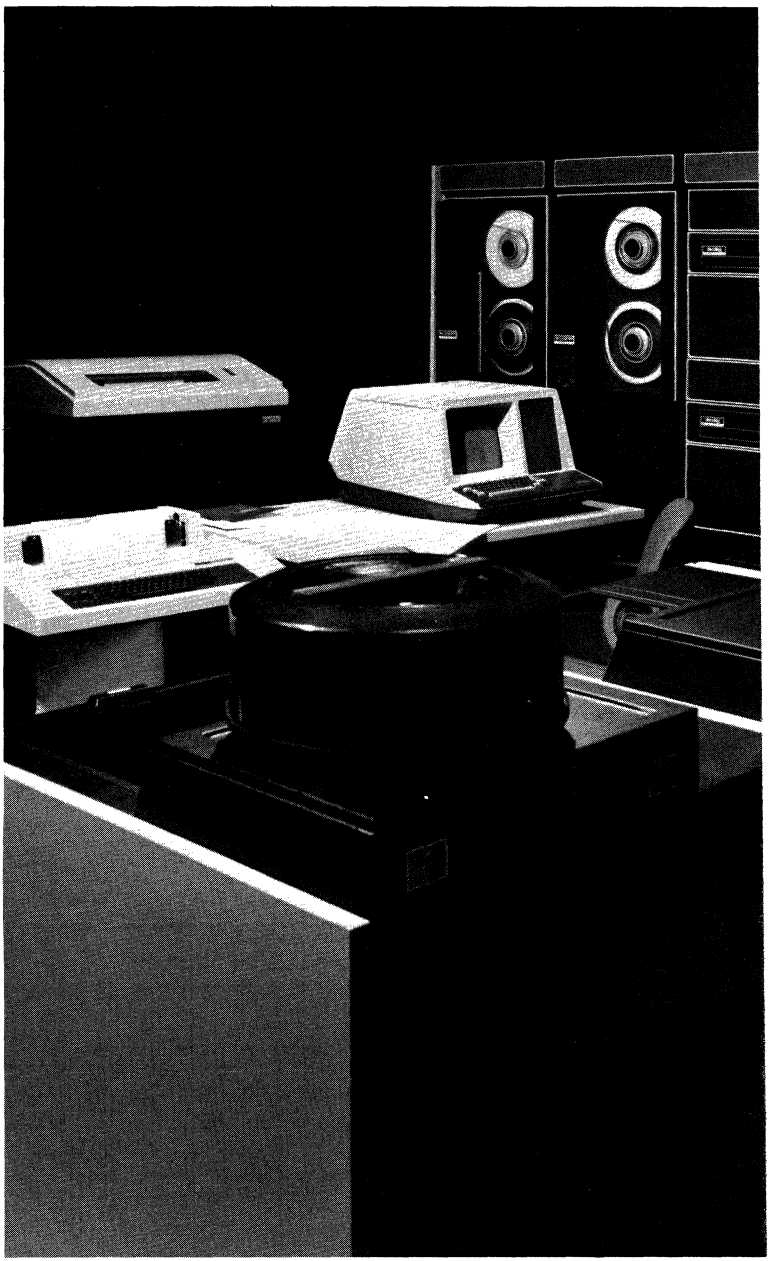
```
                R0=%0
                CLR R0
LOOP:           CLR TBL (R0)
                ADD #2,R0
                CMP R0,#100.
                BNE LOOP
```

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

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

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

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



CHAPTER 6

MEMORY MANAGEMENT

INTRODUCTION

During the execution of user programs, various system resources are required at different times. There is only one CPU, and only one program can fetch and execute instructions at one time; however, other operations such as I/O may occur simultaneously. Frequently, a program may use the CPU for only a short amount of processing time and then wait for system resources to become available (e.g., memory, peripherals, etc.) or for feedback from concurrently active programs. During this processor idle time, another program could make use of the CPU. This concept is known as multiprogramming. Therefore, to most efficiently utilize the speed and power of the PDP-11 system, it is essential that several programs reside in main memory simultaneously.

The task of the executive (monitor or supervisory program) is to control the execution of the various user programs, manage the allocation of memory and peripheral device resources, and safeguard the integrity of the system by careful control of each user program.

CONCEPTS

Before describing the memory management schemes incorporated by the family of PDP-11 processors, it is important to review several related concepts.

Virtual Address Space

Virtual address space is that set of addresses restricting the size of a user's program. For instance, a program written for a PDP-11 processor is restricted to a 16-bit address space. The PC (Program Counter) is a 16-bit register. Therefore, each user program can reference only the range of addresses between 0 and 177777 octal. This range of 32K Words or 64K Bytes (200000 octal bytes) is known as the program's virtual address space. Each program's virtual address space begins with address 0 and can extend upward to a maximum of 64K Bytes. Figure 6-1 illustrates several user programs and their associated virtual address space.

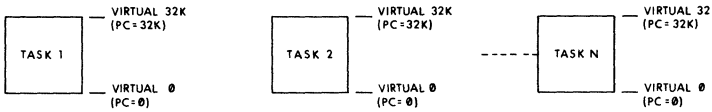


Figure 6-1 Program Virtual Address Space

Physical Address Space

Physical address space is a contiguous series of word addressable hardware locations used to define main memory and peripheral device registers. Three magnitudes of physical address space are utilized by the PDP-11 family of processors; 16-bit, 18-bit, and 22-bit. The 16-bit space yields a total of 64K Bytes and the 18-bit space yields a total of 256K Bytes. Since devices on the UNIBUS are addressable via an 18-bit address, it is clear, that in both of the above cases (16- and 18-bit), main memory may be physically attached to the UNIBUS. The 22-bit space yields a total of 4096K Bytes. In this case however, the physical address range (22 bits) exceeds that of the UNIBUS (18 bits); and main memory must be located on a separate memory bus.

Peripheral Device Register Addressing

Up to this point, virtual and physical address space have been viewed as the series of locations available to the programmer as program space. However, some provisions must be made to address peripheral device registers; a function necessary in performing I/O operations. The top 8K Bytes of physical address space have been assigned the addresses of peripheral device registers. Therefore, any reference to an address contained within the top 8K Bytes of virtual address space causes a reference to the corresponding address within the I/O page (top 8K Bytes) of the particular physical address space. The diagram in Figure 6-2 illustrates physical address space including main memory and UNIBUS peripheral device register (I/O page) space.

The diagram in Figure 6-2 will be explained more fully during the discussion of 16-, 18-, and 22-bit mapping of processor addresses.

Address Relocation

Very often a program is loaded into main memory at a starting address other than zero. This situation occurs when more than one program is loaded into main memory. When any one program is executing, it is accessed by the processor as if it were located in the set of main memory addresses beginning at zero. When the processor accesses program location 0 (PC = 0), a constant called the base address, is added to the virtual address in the PC by the memory management

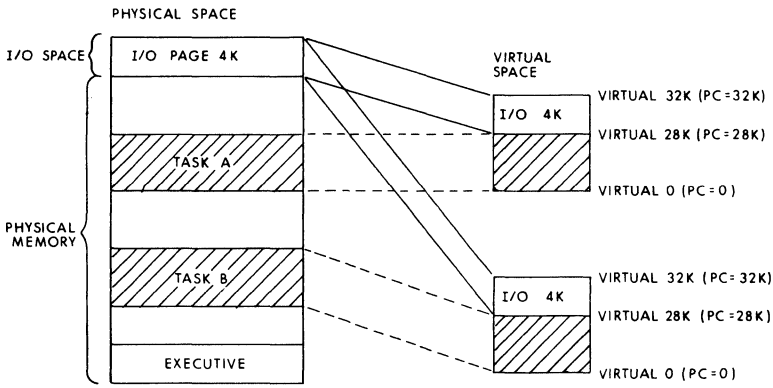


Figure 6-2 Physical Address Space

hardware. Thus, the relocated or actual memory address of program location 0 is accessed. This process is known as address relocation or address translation. This same base address is added to all references while the same program is executing. A different base address is used for each program in main memory. (The previous two statements however, assume that the executing program resides in a contiguous area of main memory. Later in this chapter we will see that a program can also be loaded into main memory in non-contiguous segments known as pages. When this situation occurs, each individual page is assigned a different relocation constant.)

To illustrate this point, lets look at a simplified memory relocation example. In Figure 6-3, 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. However, when Program B is executing, the relocation constant is changed to 100000_8. Then Program B virtual addresses are relocated by the relocation constant 100000_8.

MEMORY MANAGEMENT

Memory management is the hardware that translates (relocates) the program's 16-bit virtual address into either an 18-bit or 22-bit (processor dependent) physical address. The hardware consists of an adder, a number of registers that perform the actual address translation, and an overall internal system protection scheme.

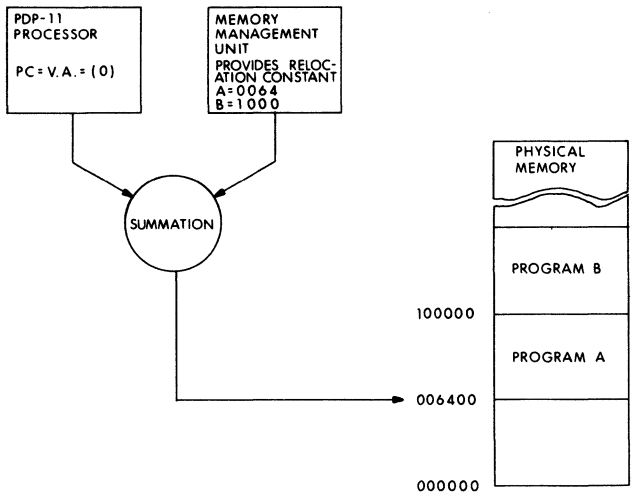


Figure 6-3 Simplified Memory Relocation Example

The basic function of memory management is to perform memory relocation and provide extended memory addressing capability for systems with greater than 28K words of physical memory. In order to perform this basic function, however, the memory management system utilizes a series of Active Page Registers (APRs). The APRs are actually a set of hardware relocation registers that permit several user's programs, each starting at virtual address 0, to simultaneously reside in physical memory.

In the PDP-11 system, a program is mapped (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 x 32) words, and the maximum number of pages in the program is 8 (4096 words x 8 = 32K). Memory management contains 8 APRs for mapping virtual pages into physical memory. Using all of the eight available active pages 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 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 6-4 illustrates several points about memory relocation. These points are:

1. Although the program appears to be in contiguous address space to the processor, the 32K-word virtual 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. The physical memory space need not be contiguous.
2. Pages may be relocated to higher or lower physical addresses, with respect to their virtual address ranges. In the example of Figure 6-4, 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).
3. All of the pages shown in the example start on 32-word boundaries.
4. 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 6-4, 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.

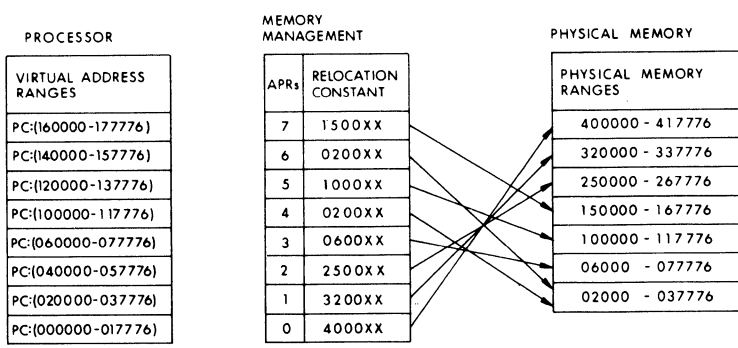


Figure 6-4 Relocation of a 32K Word Program into 124K Word Physical Memory

NOTE

The manipulation of Instruction and Data space (I and D space) is an advanced programming technique that effectively doubles the user's virtual address range from 32K to 64K words. The concept of I and D space will be discussed in this chapter.

An important function of memory management is to keep track of and to control memory allocation as well as monitor memory access violation attempts. The reason for this statistical and control hardware is to pass system parameters to an intelligent software program to effectively manage physical memory resources. This intelligent software is known as the Kernel, monitor, executive, operating system, etc.

A key goal of the memory management scheme is to protect the operating system software from the user community as well as to protect individual programs from one another. PDP-11 memory management provides the hardware facilities to implement all of the following types of memory protection:

- 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

Memory management divides memory into individual sections called pages. Each page has a protection or access key associated with it that defines the type of access allowed on that particular page. For example, a page can be labeled memory resident Read/Write, memory resident Read-only, or non-resident. To more fully understand these access control types, lets look at the memory requirements of a typical application program. If the application program can be contained within three pages of virtual space (24KB), then only three pages of main memory need be allocated by memory management as resident for that program, all other pages are assigned non-resident status. Therefore, the non-resident access key can be used to allocate physical memory efficiently. If the kernel contains an area that could be of use to a user but must be non-modifiable, then that area is designated as read-only. However, there might be a data base or a common data area in the users space that must be updated constant-

ly, i.e., a data base of digital data or A/D conversion data. In this case, the data base or common data area must be designated as read-write.

Kernel, Supervisor, and User Mode

The PDP-11 processor family offers either two or three (dependent upon processor model) modes of execution, Kernel, Supervisor, and User. Their use is to enhance the memory protection scheme and to increase the flexibility and functionality of timesharing and multi-programming environments.

Kernel mode is the most privileged of the three modes and allows execution of any instruction. In an operating system featuring multi-programming, the ultimate control of the system is implemented in code that executes in kernel mode. Typically, this includes; control of physical I/O operations, job scheduling and resource management. Memory management mapping and protection allows these executive elements to be protected from inadvertant or malicious tampering by programs executing in the less privileged processor modes. If the I/O page is only mapped in kernel mode, then only the kernel has access to the memory management registers to re-map or modify the protection. This is because the memory management registers themselves exist in the I/O page.

In order for a user program to have sensitive functions performed in its behalf, a request must be made of the executive program, typically in the form of a software trap that vectors the processor into kernel mode. Thus the executive code remains in control and can verify that the function requested is consistent with the operation of the system as a whole.

The supervisor mode is the next most privileged mode, and may be used to provide for the mapping and execution of programs shareable by users but still requiring protection from them. This might include command interpreters, logical I/O processors, or runtime systems.

User mode is the least privileged mode and prohibits the execution of instructions such as HALT and RESET as does Supervisor mode. A multi-programming operating system will typically restrict execution of user programs to user mode to prevent a single user from having a negative effect on the system as a whole. The user's virtual address space is set up such that the only areas of memory that can be written are those that belong to that user. Areas shared among users are protected as read-only, execute-only, or for both read and execute access.

Interrupt Conditions Under Memory Management Control

The memory management unit relocates all addresses. Thus, when it

is enabled, all trap, abort, and interrupt vectors are considered to be in Kernel mode virtual address space. When a vectored transfer occurs, control is transferred according to a new Program Counter (PC) and Processor Status Word (PSW) contained in a two word vector relocated through the Kernel page address register set. Relocation of trap addresses means that the hardware is capable of recovering from a failure in the first physical bank of memmory.

When a trap, abort, or interrupt occurs, the "push" of the old PC and old PSW is to the User/Supervisor/Kernel R6 stack specified by CPU mode bits 15, 14 of the new PS in the vector. (00 = Kernel, 01 = Supervisor, 11 = User.) The CPU mode bits also determine the new page address register set. Thus, it is possible for a Kernel mode program to have complete control over service assignments for all interrupt conditions, since the interrupt vector is located in Kernel space. The Kernel program may assign the service of some of these conditions to a Supervisor or User mode program by simply setting the CPU mode bits of the new PSW in the vector to return control to the appropriate mode.

Instruction and Data Space

The memory management unit in some processor models can relocate data and instruction references with separate base address values; thus, it is possible to have a user program of 64K words consisting of 32K of pure instructions or procedure code and 32K of data; all within a program's virtual address range.

The user can enable the I and D space mode of operation (under program control) by setting the appropriate bit in memory management register 3. (Memory management registers will be explained at the end of this chapter.)

Eight I space APRs accommodate up to 32K instruction words and eight D space APRs accommodate up to 32K data words. By using the separate I and D space APRs, a maximum 64K word program capacity is possible. The following rules apply to any separate I and D space programs:

1. I space can contain only instructions, immediate operands (Mode 2, Register 7), absolute addresses (Mode 3, Register 7), and index words (Modes 6 and 7). This restriction is reflected in Table 6-1.
2. The stack page must be mapped into both I and D space if the Mark instruction is used (standard PDP-11 subroutine calling sequence), because it is executed off the stack.
3. I space-only pages cannot contain subroutine parameters, which are data. Therefore, any page that contains standard PDP-11 call-

ing sequences for example cannot be mapped into an I space page.

4. The trap catcher technique of putting .+2 in the trap vector (TV) followed by a Halt must be mapped into both I and D space.

Table 6-1 illustrates the separation of I and D references for all address modes and all registers. Note that all registers (R0-R7) are in both spaces.

ACTIVE PAGE REGISTERS (APRS)

The memory management unit uses two or more sets of eight 32-bit Active Page Registers (APRs). 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 locate the currently active memory pages.

One set of APRs is dedicated for Instruction space, and one for Data space for each supported mode of operation. Figure 6-5 illustrates the selection of an APR (PAR/PDR) register set. The Current Mode bits, <15:14>, of the PSW select the mode of execution. (Once again, some members of the PDP-11 family do not utilize Supervisor mode.) When the memory management unit is turned on, the upper three bits, <15:13>, of the virtual address generated by the processor (PC) are used to select one of the 8 PAR/PDR relocation register sets. And finally, bits <2:0> of Memory Management Status Register 3 are used to select I space only, or the combined use of I and D space for each memory management mode independently. (If I space alone is selected, then both instructions and data reside in I space.)

Page Address Register (PAR)

The page address register (PAR), illustrated in Figure 6-6, contains the page address field (PAF) specifing the starting (base) address of the page as a block number in physical memory. The PDP-11/34A PAF contains 12 bits while the PDP-11/24, 11/44, and 11/70 PAF contains 16 bits.

The PAR 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 PAR as a relocation tool.

Page Descriptor Register (PDR)

The Page Descriptor Register (PDR), illustrated in Figure 6-7, contains information relative to page expansion, page length, and access control.

Table 6-1 Addressing Mode I and D References

Mode	Register	Name		
000	X	Register	INSTRUCTION	I space
001	X	Register Deferred	INSTRUCTION	I space
			DATA	D space
010	0-6	Autoincrement	INSTRUCTION	I space
			DATA	D space
	7	Immediate	INSTRUCTION	I space
			IMMEDIATE DATA	I space
011	0-6	Autoincrement		
		Deferred	INSTRUCTION	I space
			INDIRECT	D space
			DATA	D space
	7	Absolute	INSTRUCTION	I space
			ABSOLUTE	
			ADDRESS	I space
			DATA	D space
100	0-6	Autodecrement	INSTRUCTION	I space
			DATA	D space
	7	DO NOT USE THIS CONSTRUCTION		
101	0-6	Autodecrement		
		Deferred	INSTRUCTION	I space
			INDIRECT	D space
			DATA	D space
	7	DO NOT USE THIS CONSTRUCTION		
110	X	Index	INSTRUCTION	I space
			INDEX	I space
			DATA	D space
111	X	Index Deferred	INSTRUCTION	I space
			INDEX	I space
			INDIRECT	D space
			DATA	D space

Note that when D space is not enabled for a mode by setting the proper bit in SR3, all memory references are relocated and protected by the I space set of PAR/PDR registers.

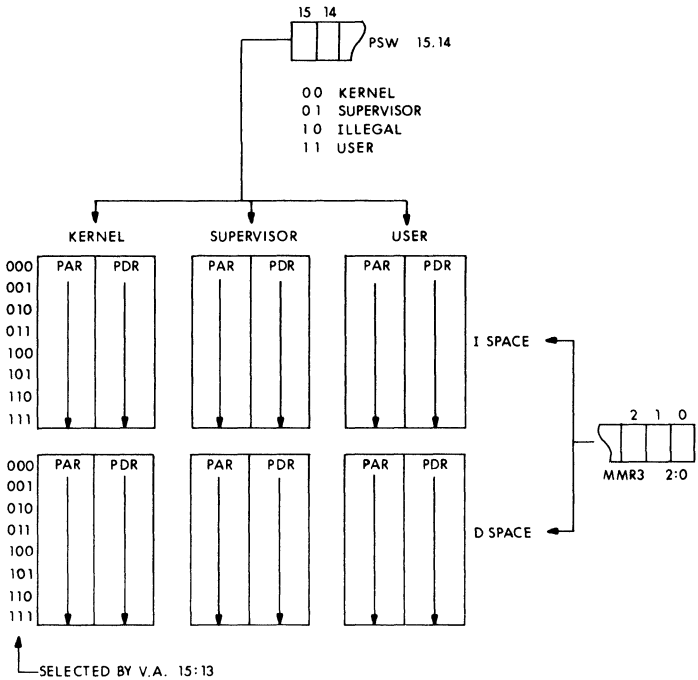


Figure 6-5 (PAR/PDR) Register Set

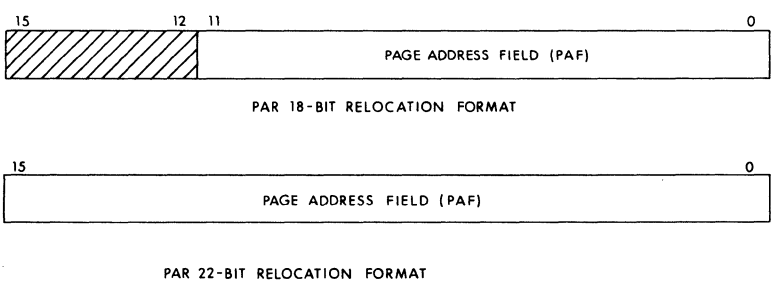


Figure 6-6 The Page Address Register

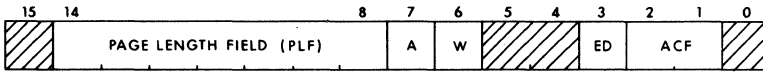


Figure 6-7 The Page Descriptor Register

Access Control Field (ACF) — Bits <2:0> of the PDR, contains the access rights to this particular page. The access codes (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.

Expansion Direction (ED) — During the execution of a program, additional memory space is frequently required by a page. Bit <3> of the PDR indicates in which direction the page expands. A logic zero in this bit (ED = 0) indicates that the page expands upward from relative zero (page base address). A logic 1 in this bit (ED = 1) indicates that the page expands downward toward relative zero (page base address). When expansion is upward, the page length is increased by adding blocks with higher relative addresses. Upward expansion is usually specified for program or data pages so that more program or table space can be made available. Figure 6-8 illustrates an example of upward page expansion.

When expansion is downward,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. Figure 6-9 illustrates an example of downward page expansion.

Access Information Bits — Bit <6> of the PDR, the Writen Into (W) bit, indicates whether the page has been written into since it was loaded in memory. A logical 1 in bit <6> indicates a modified page. The W bit is automatically cleared when the PAR or PDR of that page is written into.

In disk swapping and memory overlay applications, the W bit 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 modified (logical 0 in bit <6>) need not be saved and can be overlayed with new pages if necessary.

Bit <7> of the PDR, the Attention (A) bit, indicates whether any memory page accesses caused memory management trap conditions to be true. A logical 1 in bit <7> indicates a memory management trap condition. Trap conditions are specified by the ACF bits of the PDR. The following conditions will set the A bit;

1. ACF = 001 and read reference
2. ACF = 100 and read or write reference
3. ACF = 101 and write reference

The A bit (PDP-11/70) is used in the process of gathering memory management statistics for the purpose of optimizing memory use. The A bit is automatically cleared when the PAR or PDR of the page is written into.

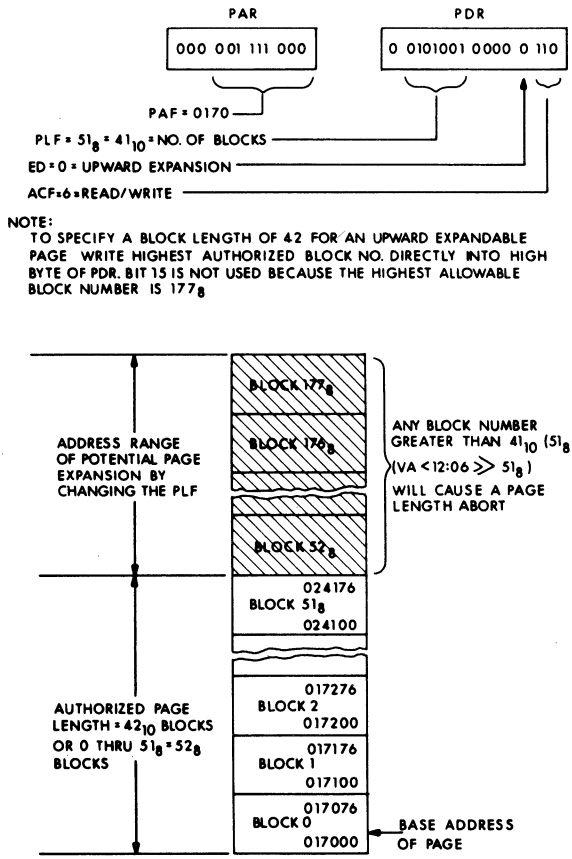


Figure 6-8 Upward Page Expansion

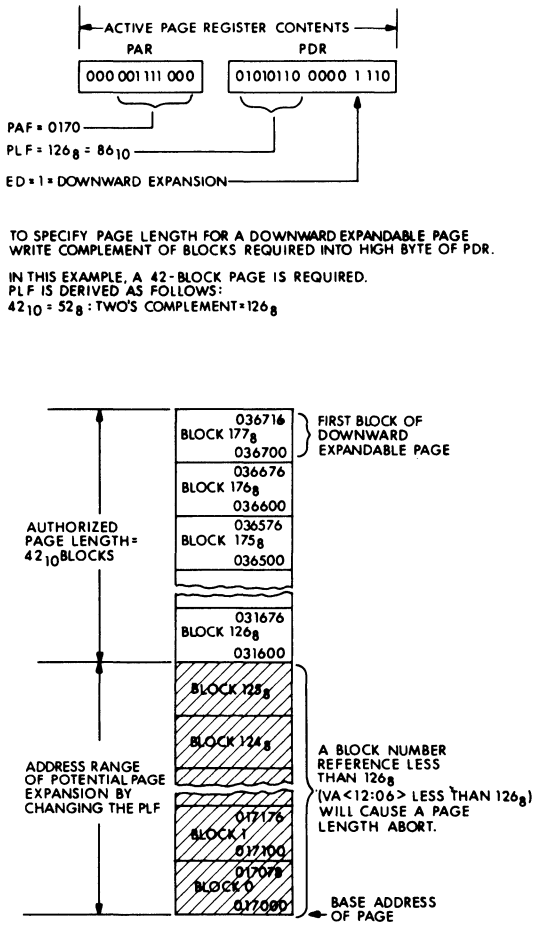


Figure 6-9 Downward Page Expansion

PHYSICAL ADDRESS CONSTRUCTION

When the memory management unit is turned off, the 16-bit virtual address generated by the processor is interpreted as a direct physical address (PA). Therefore, the total physical address space accessible to a system without memory management is 28K of main memory and 4K of I/O. However, when the memory management unit is enabled, the normal 16-bit virtual address generated by the processor is no longer interpreted as a direct physical address, but as a virtual address containing information to be used in constructing a new physical address. The information contained in the virtual address is combined with relocation information contained in the PAR to yield a physical address. Via the MMU, memory is dynamically allocated in pages. The starting physical address for each page is an integral multiple of 32 words, each page contains a maximum of 4,096 words.

Virtual Bus Address (VBA) and APRs

As stated in the last paragraph, the basic information needed to construct a physical address is derived from the virtual address and the appropriate PAR. The VA is illustrated in Figure 6-10.

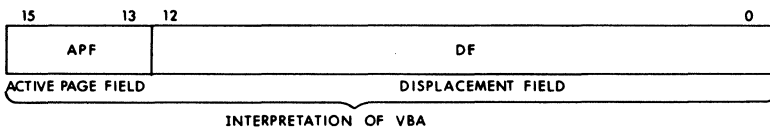


Figure 6-10 Interpretation of a Virtual Address with Memory Management enabled

The 16-bit virtual address is interpreted as having the following two fields:

- The Active Page Field (APF). The APF is a 3-bit field, <15:13>, used to determine which of 8 active page registers (PAR0-PAR7) will be used to form the physical address.
- The Displacement Field (DF). The DF is a 13-bit field, <12:0>, containing an address relative to the beginning of a page. This permits page lengths up to 4K words. The displacement field is further subdivided into two fields as illustrated in Figure 6-11.

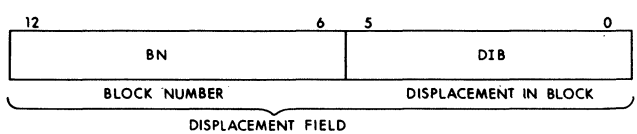


Figure 6-11 Interpretation of Displacement Field

The displacement field (DF) consists of:

- The physical memory Block Number (BN). This 7-bit field, <12:06>, is interpreted as the block number (0-127) within the current page.
- The Displacement in the Block (DIB). This 6-bit field, <5:0>, contains the displacement within the block (0-31 words) refered to by the block number (BN).

The remaining information needed to construct the physical address, i.e., the relocation constant (base address), comes from the PAR. As illustrated in Figure 6-6, the PAR contains a field known as the page address field (PAF). It is this field that specifies the starting address or relocation constant of the currently active memory page.

Before illustrating specific 18- and 22-bit relocation examples, let's summarize the procedure for constructing any physical address. The logical sequence involved is as follows:

1. Select a set of APRs, depending on the space being referenced (I or D). ((Refer to Figure 6-5.)
2. The APF of the VBA is used to select a PAR (PAR0-PAR7). (Refer to Figure 6-10).
3. The PAF of the selected PAR contains the starting address of the currently active page as a block number in physical memory. (Refer to Figure 6-6.)
4. The Block Number (BN) from the VBA is added to the PAF to yield the number of the physical block in memory which will contain the PA being constructed.
5. The Displacement in Block (DIB) from the Displacement Field (DF) of the VBA is joined to the physical block number to yield the physical address.

This sequence is illustrated in Figure 6-12.

At this point, lets look at several virtual to physical address translations. In the first example, a 16-bit virtual address will be translated into an 18-bit physical address. The address to be relocated is 157746_8 virtual. In order to perform this example however, we must make one assumption; that the PAF of the PAR already contains a main memory relocation constant. In this example, the relocation constant is 5460_8. The actual flow of translation is illustrated in Figure 6-13.

In the next example, a 16-bit virtual address will be translated into a 22-bit physical address. In this case, the address to be relocated is

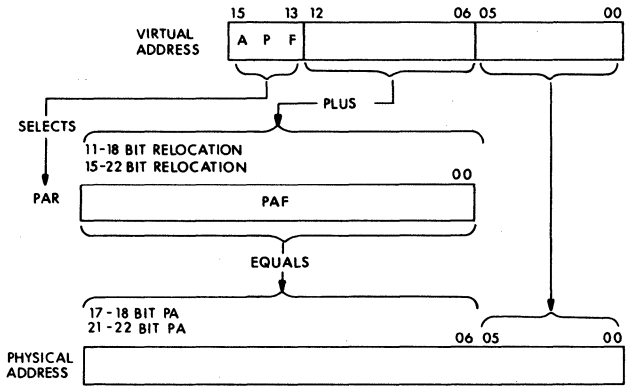


Figure 6-12 Virtual to Physical Address Translation

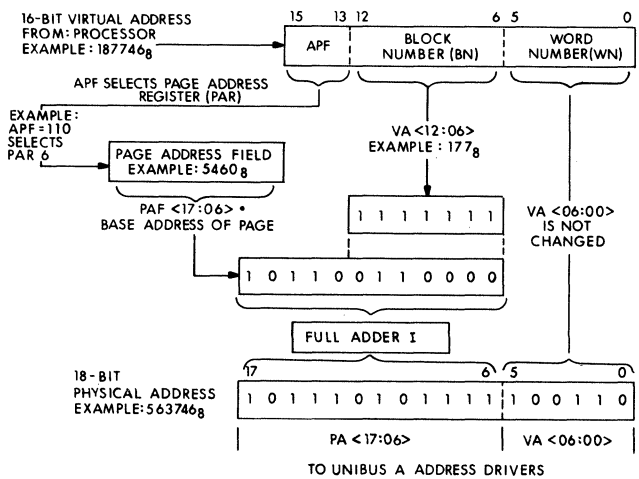


Figure 6-13 16-Bit Virtual to 18-Bit Physical Address Translation

157746_8. Once again, to perform the translation, we will assume that the PAF of the PAR already contains a main memory relocation constant. In this example, the value in the PAF is 135460_8. (Please note that the only difference between the 18- and 22-bit examples is the length of the PAF. Refer to Figure 6-6.) The actual flow of translation is illustrated in Figure 6-14.

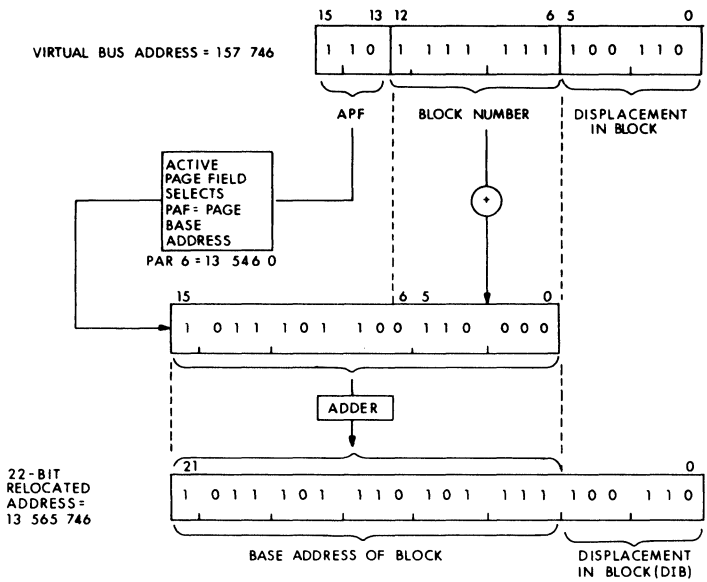


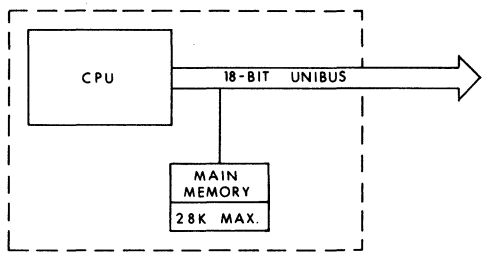
Figure 6-14 16-Bit Virtual to 22-Bit Physical Address Translation

MAPPING

Mapping is the process of converting the virtual address generated by the program to a physical memory address, or to a UNIBUS address, or the process of converting a UNIBUS address to a physical memory address. The virtual address is mapped by the memory management hardware and the UNIBUS address is mapped by the UNIBUS map hardware. Memory management and the UNIBUS map are separate pieces of hardware; one may be enabled independently of the other. (Note here, that only processors supporting a 22-bit physical address space use the UNIBUS map.) Before introducing specific mapping diagrams, let's look at a functional block diagram of each of the processors described within this handbook, the physical address space supported by each.

Figure 6-15 illustrates the PDP-11/04 processor. This processor does not contain either a memory management unit nor a UNIBUS map. This CPU can access a maximum of 28K words of main memory and the 4K word I/O page.

The physical address space supported by the PDP-11/04 is illustrated in Figure 6-16. Main memory is physically attached to the UNIBUS.



OVERALL BLOCK DIAGRAM OF PDP-11/04

Figure 6-15 Overall Block Diagram of the PDP-11/04

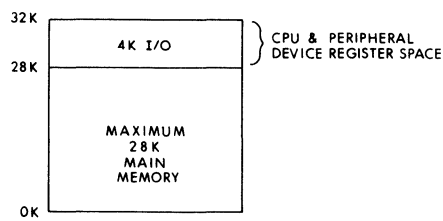


Figure 6-16 PDP-11/04 Physical Address Space

Figure 6-17 illustrates the PDP-11/34A processor. This processor contains a memory management unit that, when enabled, translates the user's 16-bit virtual addresses into 18-bit UNIBUS (physical) addresses. With memory management enabled, the PDP-11/34A has the ability to access a maximum of 124K words of main memory in addition to the 4K word I/O page.

The physical address space supported by the PDP-11/34A is illustrated in Figure 6-18. Main memory is physically attached to the UNIBUS.

This next section describes memory management for the PDP-11/24, 11/44, and the 11/70 processors. These processors contain both memory management hardware and a UNIBUS map (PDP-11/24 option). Although processor architecture is slightly different for each CPU, memory management functionality is the same regardless of CPU. Figure 6-19 illustrates a typical PDP-11/70 processor simplified block diagram. The memory management hardware translates the user's 16-bit virtual addresses into 22-bit physical addresses. The UN-

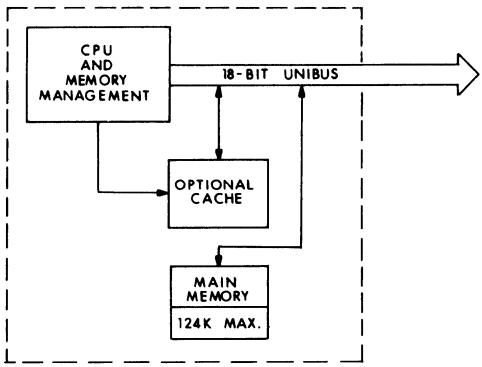


Figure 6-17 Overall Block Diagram of the PDP-11/34A

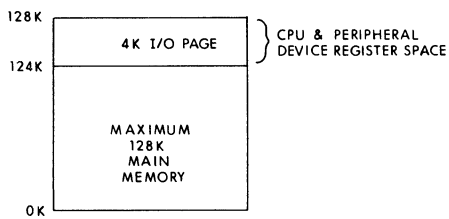


Figure 6-18 PDP-11/34A Physical Address Space

IBUS map performs the address conversion that allows devices on the UNIBUS to communicate with physical memory by means of Non-Processor Requests (NPRs), i.e., Direct Memory Access (DMA) transfers. 18-bit UNIBUS addresses are converted to 22-bit physical addresses via the UNIBUS map hardware.

The physical address space supported by the PDP-11/24, -11/44, and -11/70 CPUs is illustrated in Figure 6-20.

Referring to Figure 6-20, the following points can be observed:

1. UNIBUS references include 128K physical addresses, 17 000 000 - 17 777 777, which correspond to UNIBUS addresses 000 000 - 777 777. The UNIBUS reference in turn includes the following:
   a. The Peripheral Page, which is reserved for UNIBUS device registers; it consists of 4K physical addresses, 17 760 000 - 17 777 777 (UNIBUS addresses 760 000 - 777 777).

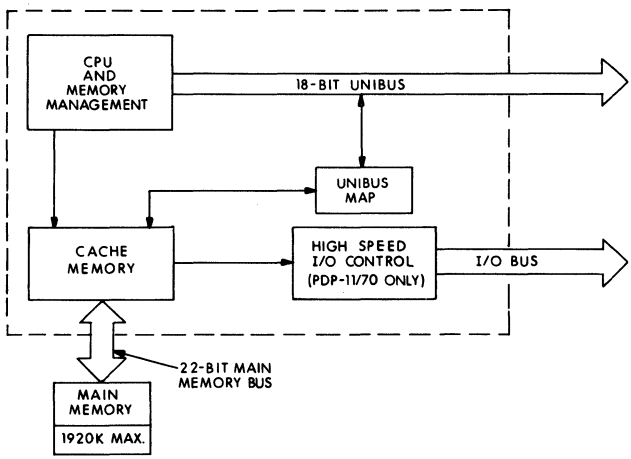


Figure 6-19 Overall Block Diagram of the PDP-11/70

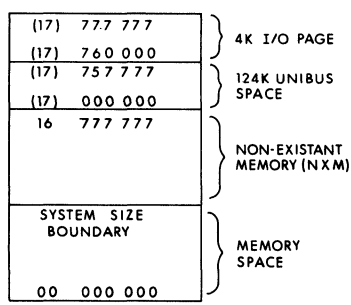


Figure 6-20 PDP-11/24, -11/44, and -11/70 Physical Address Space

b. The remaining 124K physical addresses, 17 000 000 - 17 757 777 (UNIBUS addresses 000 000 - 757 777) may be used by UNIBUS devices to access memory.

2. Memory reference includes physical addresses from 00 000 000 through the system size boundary, which is the highest address available in the system main memory. There may be no discontinuity in main memory, i.e., available memory locations must be numbered sequentially—from 00 000 000 through the system size boundary. The highest possible address is 16 777 777. Maximum possible memory is 1920K words ($2^{21}-2^{17}$ = 1,966,080, or 2048K−128K = 1920K).

3. Non-Existent Memory or NXM includes the Physical addresses from the system size boundary plus 1 - 16 777 777.

Another approach to understanding the 22-bit relocation scheme is to look at the address space bus configuration illustrated in Figure 6-21.

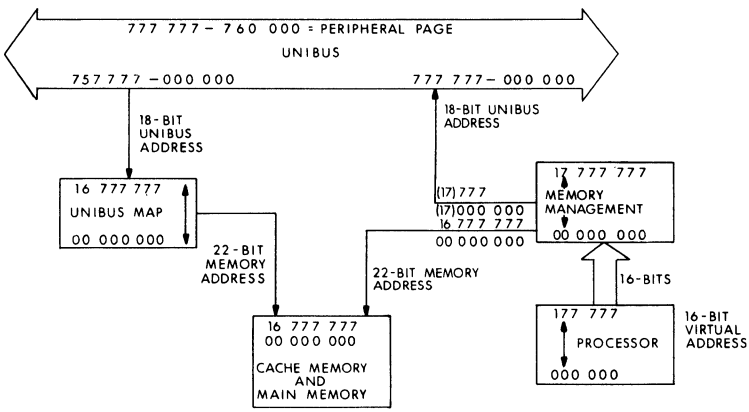


Figure 6-21 22-bit Address Space Bus Configuration

All PDP-11s generate virtual addresses in the range of 000 000 - 177 777. However, in order to access the UNIBUS, which requires an 18-bit address or main memory requiring a 22-bit address, the virtual address must be relocated. In the same manner, UNIBUS devices generate an 18-bit address, which must be expanded to 22-bits in order to access main memory. By observing Figure 6-21, it is seen that the memory management unit translates a 16-bit virtual address into a 22-bit physical address. It was also seen from Figure 6-20, that addresses between the range of 00 000 000 through 16 777 777 referenced main memory and addresses between the range of 17 000 000 through 17 777 777 referenced UNIBUS space. Therefore all addresses within the range of 00 000 000 and 16 777 777 are directed to cache and main memory. All other addresses (those between 17 000 000 and 17 777 777) are directed to the UNIBUS. UNIBUS addresses are those 22-bit addresses whose most significant 4 bits are all set to 1. Therefore, after the hardware strips off the most significant 4 bits (17_8), we are left with the familiar 18-bit (128K word) UNIBUS space (000 000 - 777 777).

The UNIBUS map performs a function very similar to that of the memory management hardware, it expands presently existing UNIBUS ad-

dresses to 22-bit physical addresses. This function is also known as mapping. The UNIBUS map accepts UNIBUS addresses in the range of 000 000 - 757 777 and relocates them within the physical address space of 00 000 000 - 16 777 777. (Note in this case that only the UNIBUS addresses are relocated and that the upper 4K I/O page is not touched.)

At this point, let's look at several specific memory management mapping structures regarding 16-bit, 18-bit, and 22-bit physical address spaces.

16-Bit Physical Address Space

Figure 6-22 illustrates the 16-bit mapping scheme for processors such as the PDP-11/04 and -11/34A. This illustration shows fixed relocation mapping from virtual to physical addresses. The lowest 28K of virtual addresses are treated as corresponding to the same lower 28K of physical addresses. With the PDP-11/24, -11/44, and -11/70 in 16-bit mode, the lower 28K of virtual addresses address main memory (not attached to the UNIBUS). The top 4K virtual addresses however always cause UNIBUS cycles to address the top 4K physical addresses no matter what size the physical address space might be. In this example, the top 4K virtual addresses reference physical addresses 124K - 128K.

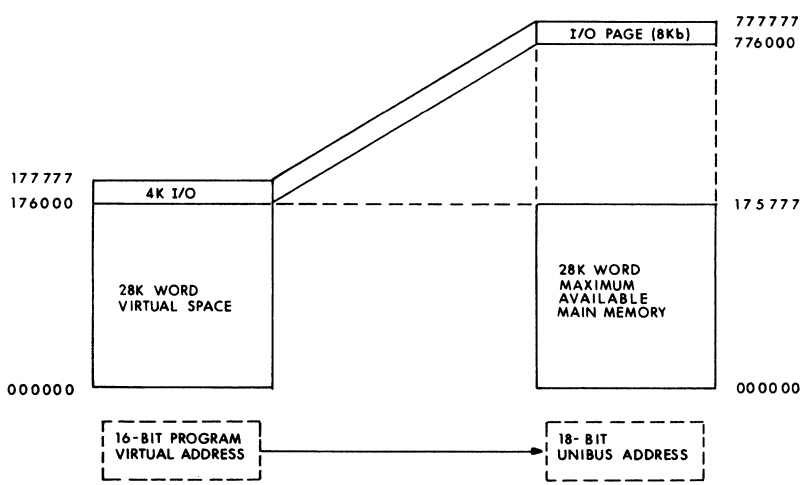


Figure 6-22 16-Bit Mapping within 18-Bit Physical Address Space

18-Bit Physical Address Space

Figure 6-23 illustrates the 18-bit mapping scheme for processors such as the PDP-11/34A with memory constraints of 124K words. Figure 6-23 depicts the fact that with memory management enabled, the user's virtual address space of 28K words can be relocated anywhere in available main memory (in 4K word pages—if necessary, refer back to Figure 6-4 and the discussion entitled **MEMORY MANAGEMENT**). However, if memory management hardware is not enabled, (under program control), the resulting mapping structure is identical to Figure 6-22. With the PDP-11/24, -11/44, and -11/70 in 18-bit memory management mode, the lower 28K of virtual addresses address main memory (not attached to the UNIBUS) using relocation.

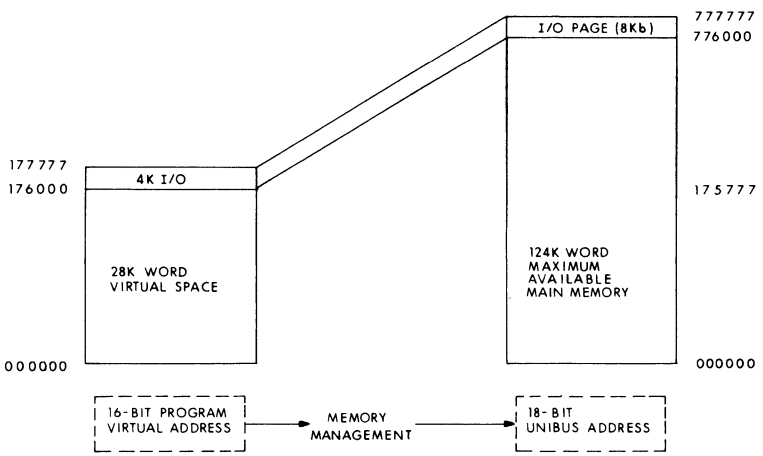


Figure 6-23 18-Bit Mapping within 18-Bit Physical Address Space

22-Bit Physical Address Space

The next series of figures illustrates 16-bit, 18-bit, and 22-bit mapping structures within a 22-bit physical address space. If the PDP-11/24, -11/44 or -11/70 system contains only 124K words of main memory, then the 16-bit mapping scheme (memory management disabled) is illustrated in Figure 6-24. And if 18-bit memory management is enabled, the mapping scheme is illustrated in Figure 6-25. The 22-bit mapping structure is illustrated in Figure 6-26. The solid arrow lines in Figure 6-26 represent a one-to-one correspondence between physical address and physical location.

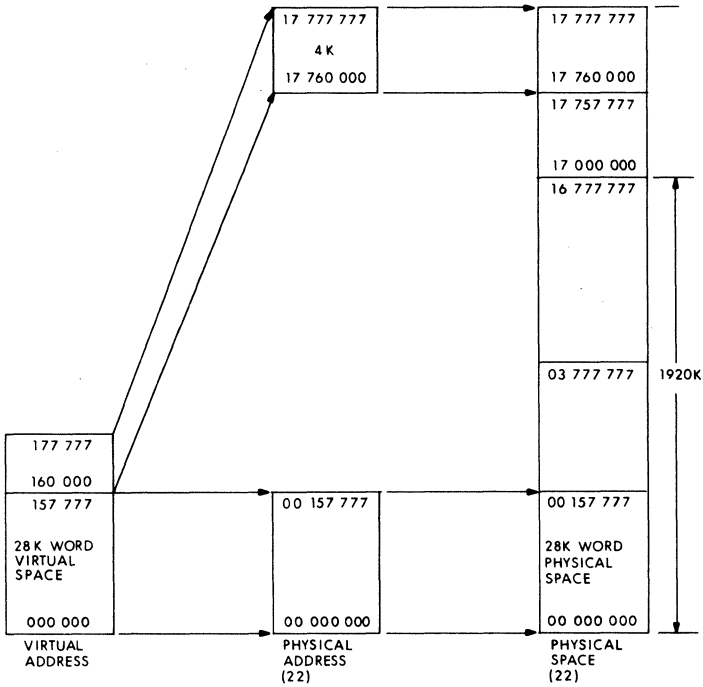


Figure 6-24 16-Bit Mapping Structure for 22-Bit Physical Address Space

The UNIBUS Map

The UNIBUS map is the interface to memory from the UNIBUS. The UNIBUS map can be in either of two operational modes; relocation enabled or relocation disabled. If the UNIBUS map relocation is not enabled, an incoming 18-bit UNIBUS address has 4 leading zeros (bits <21:18>)added for referencing a 22-bit physical address. The lower 18-bits of the UNIBUS address are identical to the 18-bit physical address, i.e., no other translation is performed.

However, when the UNIBUS map is enabled, the UNIBUS map utilizes a total of 31 mapping registers for address relocation. Similar to the memory management scheme, each map register is composed of a double 16-bit PDP-11 word (in consecutive locations) that holds the 22-bit base address (right justified).

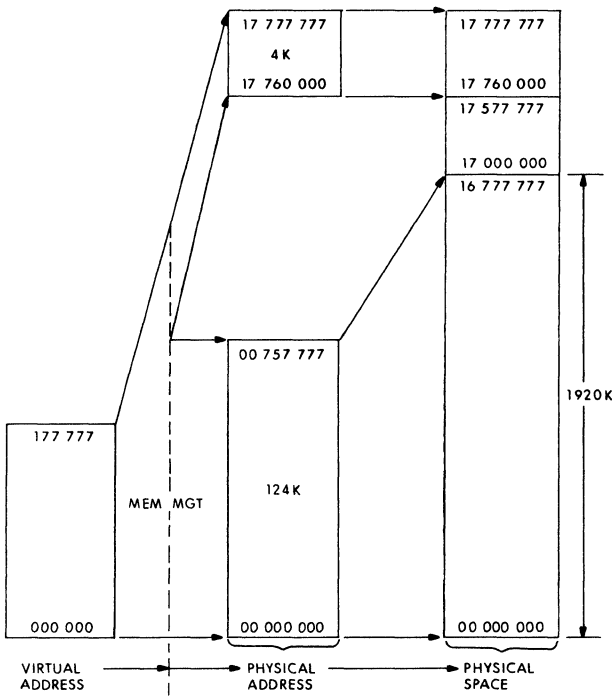


Figure 6-25 18-Bit Mapping Structure for 22-Bit Physical Address Space

If UNIBUS map relocation is enabled, the 5 high order bits of the UNIBUS address are used to select one of the 31 mapping registers. The low order 13-bits of the incoming address are used as an offset from the base address contained in the 22-bit mapping register. To form the physical address, the 13 low order bits of the UNIBUS address are added to 22 bits of the selected mapping register to produce the 22-bit physical address. The lowest order bit of all mapping registers is always a zero, since relocation is always on word boundaries. The functionality of the UNIBUS map is illustrated in Figure 6-27. Figure 6-28 illustrates the construction of a physical address from a UNIBUS address.

FAULT RECOVERY (STATUS) REGISTERS

Aborts and traps generated by the Memory Management hardware

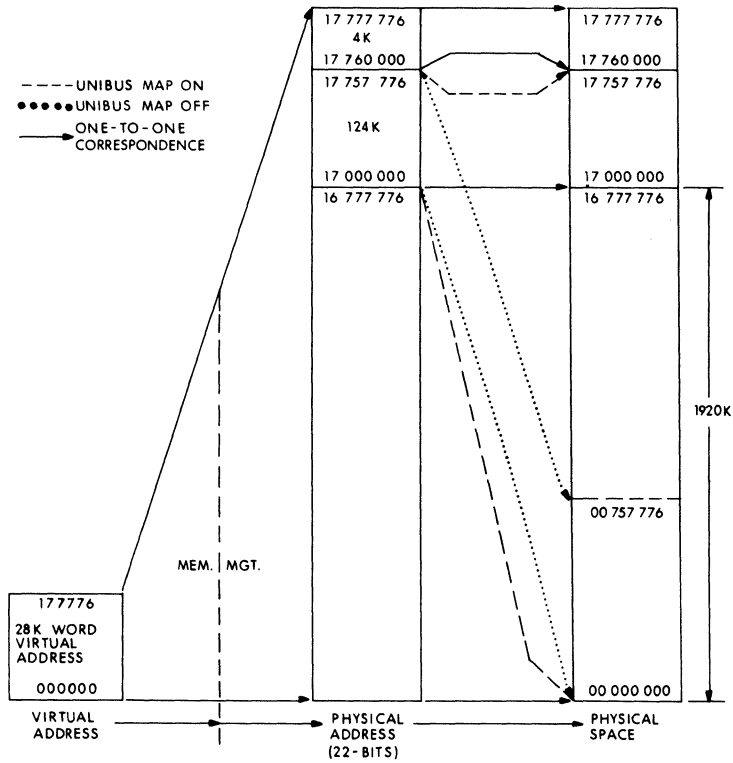


Figure 6-26 22-Bit Mapping Structure for 22-Bit Physical Address Space

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

Memory Management Register #0 (MMR0)

MMR0 contains error flags, the page number whose reference caused the abort, and various other status flags. This register is illustrated in Figure 6-29.

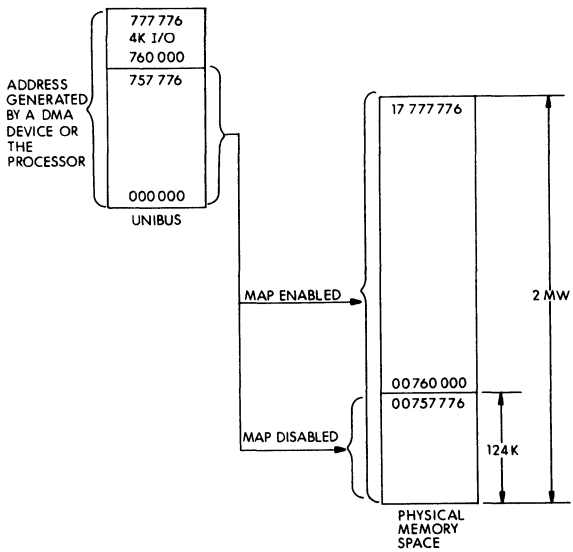


Figure 6-27 UNIBUS Map Functionality

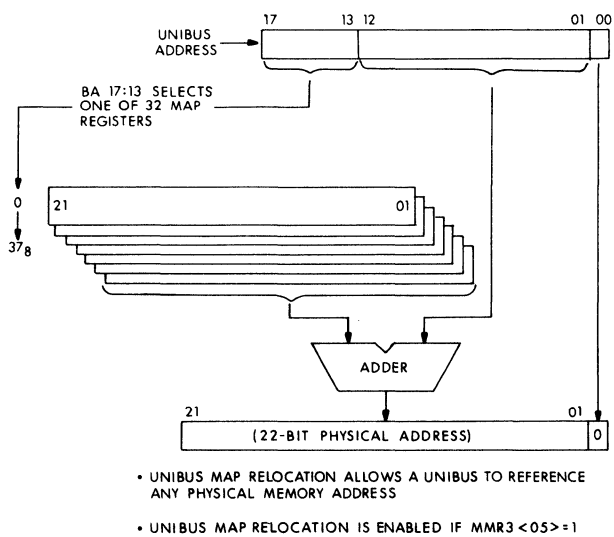


Figure 6-28 UNIBUS Map Physical Address Construction

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

Bits <15:12> are the error flags. They may be considered to be in a priority queue in that flags to the right are less significant and should be ignored. That is, a non-resident fault-service routine would ignore length, access control, and memory management flags. A page length service routine would ignore access control and memory management faults, etc.

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

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

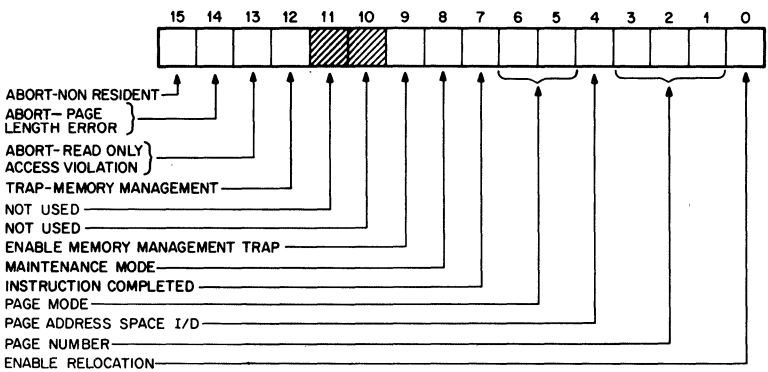


Figure 6-29 Memory Management Register #0 (MMR0)

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

Abort—Page Length Bit 14 — Bit <14> is the Abort Page Length bit. It is set by attempting to access a location in a page with a block number (Virtual Address bits <12:6>) that is outside the area authorized by the Page Length Field of the Page Descriptor Register for that page. Bits <15:14> may be set simultaneously by the same access

attempt. Bit <14> is also set by attempting to use Memory Relocation with a processor mode of 2.

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

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

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

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

NOTE

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

Maintenance/Destination Mode Bit 8 (not used by PDP-11/24)
Bit <8> specifies that only destination mode references will be relocated using Memory Management. This mode is used only for maintenance purposes.

Instruction Completed Bit 7 (PDP-11 /70 only)
Bit <7> indicates that the current instruction has been completed. It will be set to zero during T bit, Parity, Odd Address, and Timeout traps and interrupts. This provides error handling 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>.

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

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

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

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

The information contained in MMR1 is necessary to accomplish an effective recovery from an error resulting in an abort. The low order byte is written first and it is not possible for a PDP-11 instruction to autoincrement/decrement more than two general purpose registers per instruction before an abort-causing reference. Register numbers are recorded MOD 8; thus it is up to the software to determine which set of registers (User/Supervisor/Kernel—General Set 0/General Set 1) was modified, by determining the CPU and Register modes as contained in the PS at the time of the abort.The 6-bit displacement of R6 (SP) that can be caused by the MARK instruction cannot occur if the instruction is aborted. MMR1 is illustrated in Figure 6-30.

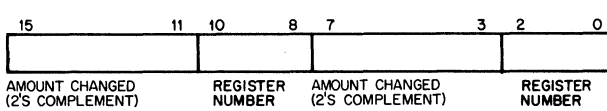


Figure 6-30 Memory Management Register #1 (MMR1)

NOTE

For the PDP-11/24, this register is not mechanized. When explicitly addressed, it reads out as a word containing all zeros, but cannot be written into. This register is included for compatibility with PDP-11 software.

Memory Management Register #2 (MMR2)

MMr2 is loaded with the 16-bit Virtual Address (VA) at the beginning of each instruction fetch, or with the address Trap Vector at the beginning of an interrupt, T Bit trap, Parity, Odd Address, and Timeout aborts and parity traps. Note that MMR2 does not get the Trap Vector on EMT, TRAP, BPT, and IOT instructions. MMR2 is read-only; it cannot be written. MMR2 is the Virtual Address Program Counter.

Memory Management Register #3 (MMR3)(PDP-11/44 and 11/70)

Memory Management Register #3 (MMR3) enables or disables the use of the D space PARs and PDRs, 22-bit mapping and UNIBUS mapping. When D space is disabled, all references use the I space registers; when D space is enabled, both the I space and D space registers are used. Bit <0> refers to the User's registers, bit <1> to the Supervisor's and bit <2> to the Kernel's. When the appropriate bits are set, D space is enabled; when clear, it is disabled. Bit <3> is used to enable the change to Supervisor mode (CSM) instruction in the 11/44. It is reserved for future use. Bit <4> enables 22-bit mapping. If Memory Management is not enabled, bit <4> is ignored and 16-bit mapping is used.

If bit <4> is clear and Memory Management is enabled (bit <0> of MMR0 is set), the computer uses 18-bit mapping. If bit <4> is set and Memory Management is enabled, the computer uses 22-bit mapping. Bit <5> is set to enable relocation in the UNIBUS map; the bit is cleared to disable relocation. Bits <15:6> are unused. On initialization, this register is set to 0 and only I space is in use. MMR3 is illustrated in Figure 6-31.

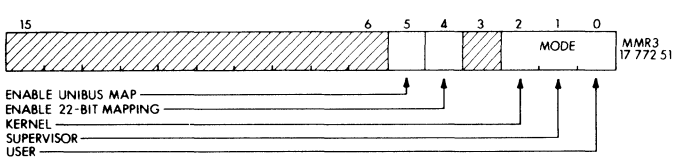


Figure 6-31 Memory Management Register #3 (MMR3)

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

NOTE

The PDP-11/24 utilizes only bits <4:5>.

Instruction Back-Up/Restart Recovery

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

1. Performing the appropriate memory management tasks to alleviate the cause of the abort (e.g., loading a missing page).
2. Restoring the general purpose registers indicated in MMR1 to their original contents at the start of the instruction by subtracting the modify value specified in MMR1.
3. Restoring the PC to the abort time PC by loading R7 with the content of MMR2, which contains the value of the Virtual PC at the time the abort generating instruction was fetched.

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

Clearing Status Registers Following Trap/abort

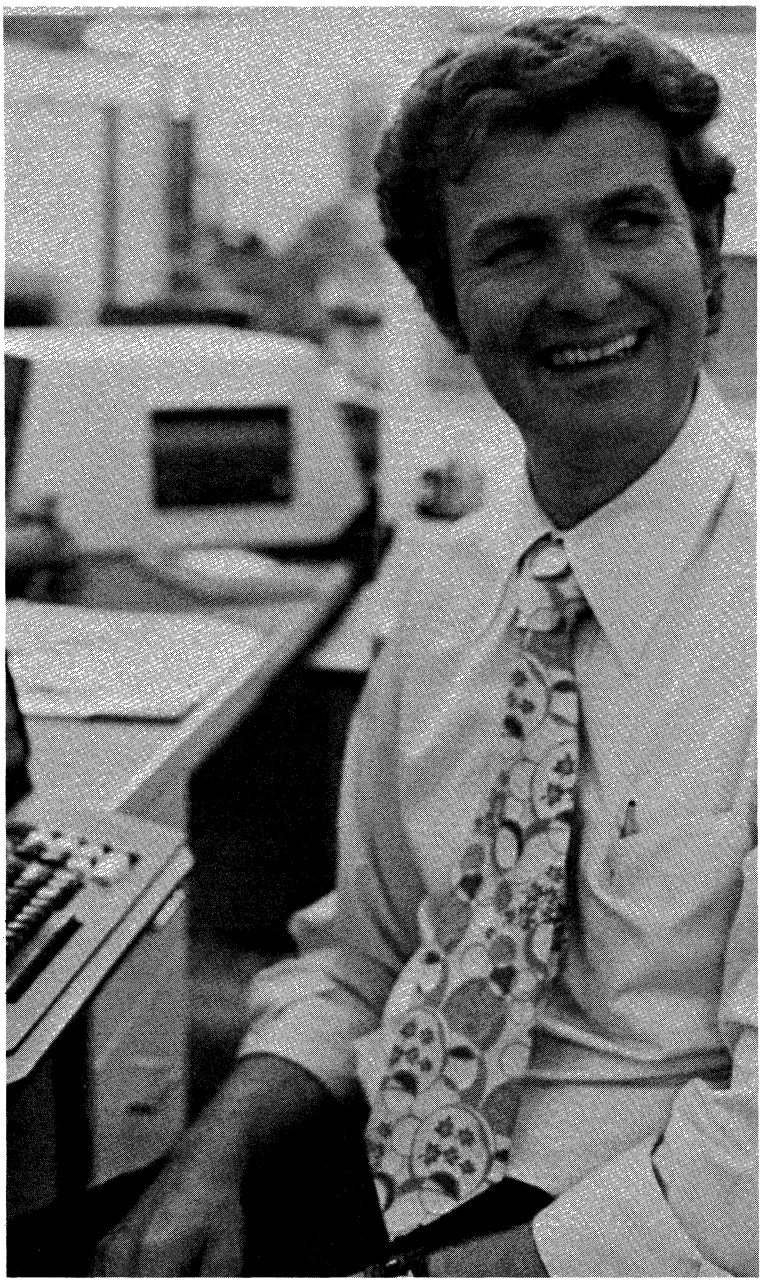
At the end of a fault service routine, bits <15:12> of MMR0 must be cleared (set to 0) to resume error checking. On the next memory reference following the clearing of these bits, the various registers will resume monitoring the status of the addressing operations. MMR2 will be loaded with the next instruction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

change information and MMR0 will log Memory Management status information.

Multiple Faults

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

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



CHAPTER 7

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

The PDP-11/04 and PDP-11/34A have similar architecture, capabilities and features. The PDP-11/04 is a cost-effective version of the high-performance PDP-11/34A processor.

The PDP-11/04 is optimized for compactness; the entire CPU logic is confined to one circuit board. This allows extra chassis space for system expansion. By offering up to 56K bytes of core or MOS memory, the PDP-11/04 offers such flexibility that you can tailor both package and price to each application. It is offered in minimum hardware configurations which allow room to grow.

The PDP-11/34A contains hardware multiply/divide instructions, Memory Management, an enhanced data path and the capability of adding hardware Floating Point and Cache Memory options. These extra features are each contained on one module. The PDP-11/34A is physically similar to the PDP-11/04. Yet it has 2½ times the power of the PDP-11/04. It also has memory expansion to 248K bytes, setting a standard of upward compatibility for PDP-11/04-based systems.

FEATURES

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

- Self-test diagnostic routines which are automatically executed every time the processor is powered up, the console emulator routine is initiated, or the bootstrap routine is initiated. These allow system faults to be detected early to avoid catastrophic failure during the running of the application program.
- Operator front panel with built-in CPU console emulator allows control from any ASCII terminal without the need for the conventional front panel with display lights and switches.
- Automatic bootstrap loader allows system restart from a variety of peripheral devices without manual switch toggling or keypad operations.
- MOS memory, with parity memory optional, expandable from a minimum of 16K bytes on the PDP-11/04 and 32K bytes on the PDP-11/34A to as much as 56K bytes on the PDP-11/04, and 248K bytes on the PDP-11/34A. This choice gives exceptional configuring flexibility, and allows tailoring of memory size to precisely fit application requirements.
- Slot-independent backplane with power and space available for significant expansion within the 5¼" or 10½" chassis. This provides

easier system configuring than the single mounting chassis most systems have.

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

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

MEMORY

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

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

The control and status register of the parity logic captures the high-order address bits of a memory location with a parity error.

Memory Capacity

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

Memory Management

Memory Management is a hardware feature in the PDP-11/34A. It serves two functions: it extends memory addresses to 18 bits (248K bytes), and provides protection and relocation features for multiuser applications. The processor can be operated in either of two modes: Kernel and User. In Kernel mode, the program has complete control and can execute all instructions. Monitors and executive programs would be executed in this mode. In User mode, the program is prevented from executing certain instructions that could modify the Kernel program, halt the computer, gain access outside the assigned memory, or issue a restart.

Battery Backup

Since MOS memory is volatile, meaning it depends on electricity to store information, a power loss or shutdown would erase its contents. To prevent this loss from occurring, a battery backup unit (BBU) has been designed to temporarily preserve the contents in memory. The BBU is an auxiliary power unit. It is charged by main ac power when the computer system is operating normally. Under normal operation, the battery backup has no effect on MOS memory. When power is interrupted, voltage sensing circuitry within the battery backup automatically causes the MOS to be refreshed by this auxiliary source, allowing for retention of memory contents.

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

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

Cache Memory

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

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

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

FLOATING POINT OPTION

The Floating Point Processor is a hardware option that enables the PDP-11/34A central processor to execute floating point arithmetic operations. It performs high-speed numerical data handling much faster and more effectively than software floating point routines. Floating point representation permits a greater range of number values than is possible with the conventional integer mode, and system speed is increased by the elimination of complex arithmetic coding routines that consume valuable CPU time. The option features both single- and double-precision (32- or 64-bit) capability and floating point modes.

The floating point processor operates using the same address modes and Memory Management facilities as the central processor. Floating Point Processor instructions can reference the floating point accumulators, the central processor's general registers, or any location in memory. The floating point processor operates in serial with CPU operations.

Floating Point Instruction Set Features

- 46 additional instructions, compatible with the floating point instruction set available on the PDP-11/23, PDP-11/44 and PDP-11/70
- 32-bit (single-precision) and 64-bit (double-precision) data modes
- Addressing modes compatible with existing PDP-11 addressing modes
- Special instructions that can improve input/output routines and mathematical subroutines
- Allows execution of in-line code (i.e., floating point instructions and other instructions can appear in any sequence desired)
- Multiple accumulators for ease of data handling
- Can convert 32- or 64-bit floating point numbers to 16- or 32-bit integers during the Store instructions
- Can convert 32-bit floating point numbers to 64-bit floating point numbers and vice-versa during the Load or Store instructions
- Is a required option when using the FORTRAN IV PLUS compiler

CONSOLES

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

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

Power	OFF	DC power to the computer is off.
	ON	Power is applied to the computer (and the system).
	STNDY	Standby; no dc power to the computer, but dc power is applied to MOS memory (to retain data). In the 5¼" box, the fans remain on.
CONT/ HALT	CONT	The program is allowed to continue.
	HALT	The program is halted.
BOOT/ INIT	INIT	The switch is spring-returned to the BOOT position. When the switch is depressed to INITialize and then returned to BOOT, the operation depends on the setting of the CONT/HALT switch.
		If the switch is set to HALT, then only the processor is initialized and no "UNIBUS INIT" is generated. Upon lifting the CONT/ HALT switch, the M9312 routine is executed, allowing examination of system peripherals without clearing their contents with "UNIBUS INIT."
		If the switch is set to CONT, then initialization and execution of the M9312 program begins.

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

Console Emulation

A ROM-resident virtual console routine permits control of the processor from any ASCII terminal. This routine emulates the functions traditionally performed by the "lights and switches" on the programmer's console.

Summary of Console Emulator Functions

LOAD ADDR	Loads into the system the address to be manipulated
EXAMINE	Allows the operator to examine the contents of the address that was loaded
DEPOSIT	Allows the operator to write into the address that was loaded and/or examined

START	Initializes the system and starts execution of the program at the address loaded
BOOT	Allows the booting of a device specified by a two-character code and optional unit number

Entry into the Console Emulator

There are four ways of entering the console emulator:

- Move the power switch to the ON position
- Depress the BOOT switch
- Enter automatically on return from a power failure
- Load the address manually

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

Example—a typical printout on power up:

XXXXXXXX *(R0)*	XXXXXXXX *(R4)*	XXXXXXXX *(R6 or Stack Pointer)*	XXXXXXXX *(R7 or Program Counter)*

@ *(Prompt Character)*

NOTE

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

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

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

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

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

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

Boot Module (M9312)

The Boot Module provides the following four functions:

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

Diagnostics

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

Bootstrap Loader

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

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

BACKPLANE CONFIGURATIONS

The processor backplane consists of a double system unit (SU) comprising nine hex slots. All PDP-11/04 and PDP-11/34A systems contain the CPU, M9312 Bootstrap/Terminator, and M9302 Terminator (or a UNIBUS jumper to the next SU). Figures 7-1 and 7-2 illustrate the implementation of MS11-JP or MS11-FP memory on the PDP-11/04 backplane. M7850 parity control is included when MS11-JP memory (32 Kb MOS) is used as shown in Figure 7-3. MS11-LB (128 Kb MOS) or MS11-LD (256 Kb MOS) has parity control included on the board (Figure 7-4).

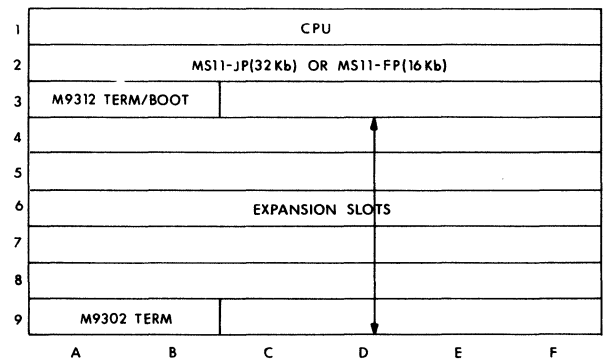


Figure 7-1 PDP-11/04 Processor Backplane Configuration with MS11-JP or MS11-FP Memory

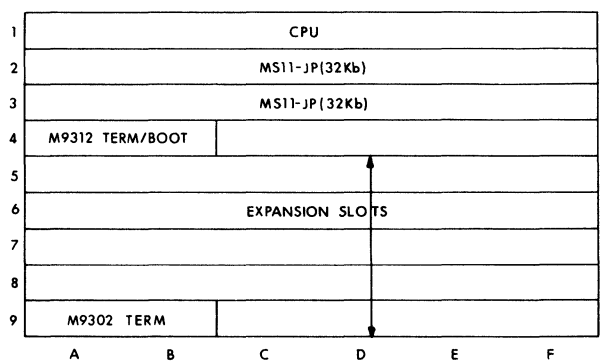


Figure 7-2 PDP-11/04 Processor Configuration with MS11-JP Memory

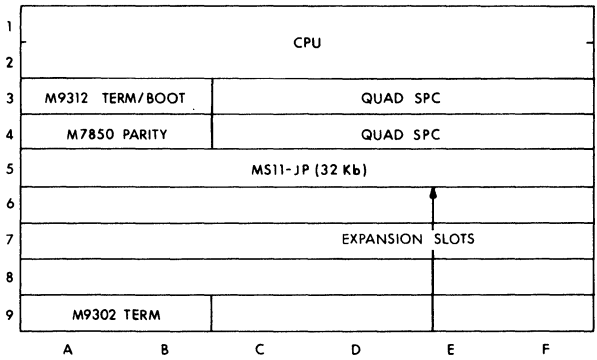


Figure 7-3 PDP-11/34A Backplane Configuration with MS11-JP Memory

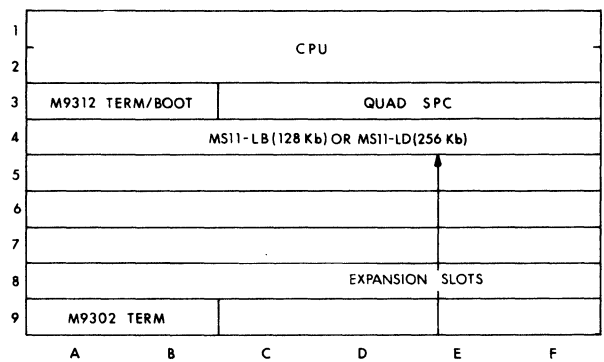


Figure 7-4 PDP-11/34A Backplane Configuration with MS11-L Memory

Additional memory or quad and hex SPC options (DL11-W, RX21 controller, etc.) may be added to the processor backplane as space allows.

The 5¼-inch chassis has space for one 9-slot backplane (2 SUs). The 10½-inch chassis has space for two 9-slot backplanes and one 4-slot backplane (5 SUs). Expansion space is obtained by adding expander backplanes and/or expansion boxes.

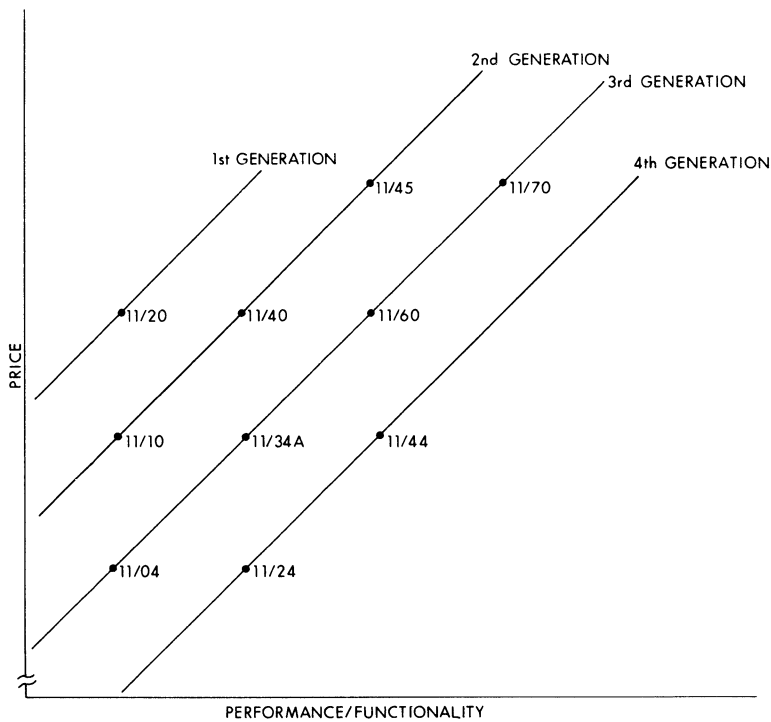
SPECIFICATIONS
PDP-11/04 and PDP-11/34A

Environment

Operating Temperature:	5°-50° C, (41°-122° F)
Relative Humidity:	10% or less to 95%; with maximum wet bulb of 32° C (90° F), maximum dew point of 2° C (36° F)

Mechanical

5¼-inch Chassis	10½-inch Chassis
Weight: 20 kg (45 lbs.)	50 kg (110 lbs.)
Height: 13.3 cm (5.25 in.)	26.7 cm (10.5 in.)
Width: 48.3 cm (19 in.)	48.3 cm (19 in.)
Depth: 63.5 cm (25 in.)	63.5 cm (25 in.)



PDP-11/24 PRODUCT POSITION

CHAPTER 8
PDP-11/24

The PDP-11/24 joins the PDP-11/44 as a new member of the fourth generation PDP-11 processor family. Designed as a low-cost, single-Hex module UNIBUS processor, the PDP-11/24 provides performance and functionality similar to the PDP-11/34A, but at lower cost. An extended 22-bit memory addressing capability makes the PDP-11/24 the lowest cost systems-oriented CPU from DIGITAL that can address up to a full megabyte of memory. The PDP-11/24 floating point unit and commercial instruction set provide programming compatibility with the PDP-11/44.

FEATURES

Integral to the PDP-11/24 central processor unit are these hardware features and expansion capabilities:

- Real-time clock which provides KW11-L compatible line-frequency clock
- DL11-W capability provides a serial port (SLU1) for the console terminal
- Ability to access up to 248Kb (1 million optional) of main memory (1 byte = 8 bits) provides ample memory space for application programs
- Second general purpose serial line (SLU2), which may be used to interface with TU58 Dec tape II or any ASCII device
- Extended Instruction Set (EIS) for better integer arithmetic throughput
- Memory management for relocation and protection in multiuser, multitask environments
- ASCII console with which the user can operate the computer without requiring access to the front panel
- 256 Kbyte Parity MOS main memory modules provide high-density, low-cost memories
- Optional Floating Point Unit with advanced features, operating with 32-bit and 64-bit numbers for better FORTRAN or BASIC throughput
- Optional Commercial Instruction Set for improved COBOL throughput

RELIABILITY, AVAILABILITY, MAINTAINABILITY (RAMP)

RAMP is a special program designed to provide Reliability, Availability and Maintainability for the PDP-11 processor family. Reliability means minimizing failures. Availability and Maintainability mean planning for ease of maintenance and spending less time isolating faults and making repairs.

The PDP-11/24 contains these RAMP features:

- Self-diagnostic bootstrap module to test the viability of the instruction set processor, memory, and console terminal interface
- Error logging to provide maximum diagnostic information to the user and Field Service
- Improved electrical integrity including implementation of Serial Line Unit Static and front panel Filters, insulated key switch and 15 Kv static immunity
- Optional Battery Backup Unit (BBU) to retain memory data during power outages

SYSTEM ARCHITECTURE

The PDP-11/24 is a minicomputer designed for multi-task and dedicated applications. A block diagram of the computer is shown in Figure 8-1.

The central processor performs all arithmetic and logical operations required in the system. Memory Management is standard with the basic computer, allowing expanded memory addressing, relocation, and protection. The UNIBUS map, which translates UNIBUS addresses to physical memory address, is an optional feature for the PDP-11/24. The UNIBUS remains the primary control path in the PDP-11/24 system. Memory addresses are passed on a separate 22-bit wide bus. This bus provides reduced memory access times. It is conceptually identical with previous PDP-11 systems; the memory in the system still appears to be on the UNIBUS to all UNIBUS devices, through the UNIBUS map.

CENTRAL PROCESSOR

The PDP-11/24 processor is the arbitration unit for UNIBUS control. It regulates bus requests and transfers control of the bus to the request device with the highest priority.

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

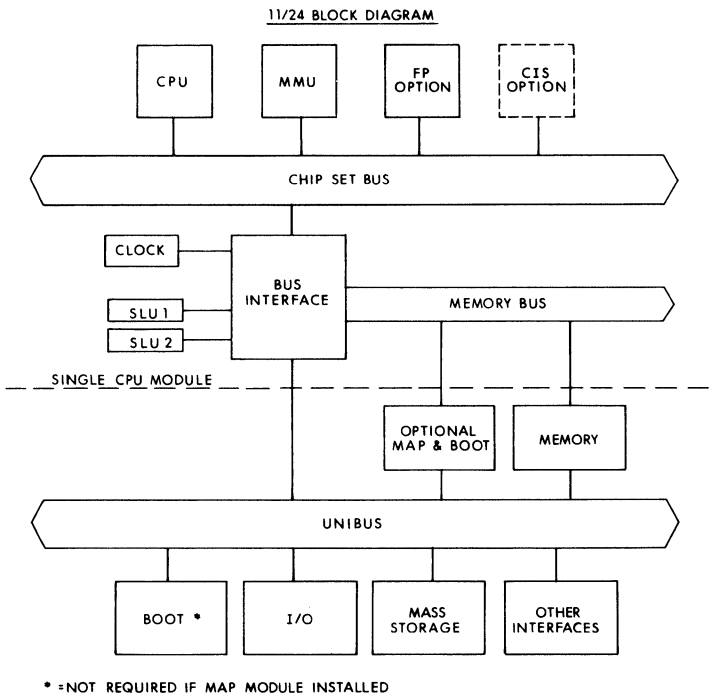


Figure 8-1 PDP-11/24 Block Diagram

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

The PDP-11/24 processor is implemented using three chips. Two MOS/LSI chips, data and control, implement the basic processor. The memory management unit (MMU), the third chip, provides a PDP-11/34 software-compatible memory management scheme.

The data chip (DC302) performs all arithmetic and logical functions, handles data and address transfers with the external world, and coordinates most interchip communication. The control chip (DC303) does microprogram sequencing for PDP-11 instruction decoding and con-

tains the control store ROM. The data and control chips are both contained in one 40-pin package. The MMU chip (DC304) contains the registers for 18 or 22-bit memory addressing and also includes the FP11 floating point registers and accumulators.

Data Chip

The data chip contains the PDP-11 general registers, the processor status word (PS), several working registers, the arithmetic and logic unit (ALU), and conditional branching logic. The data chip does the following.

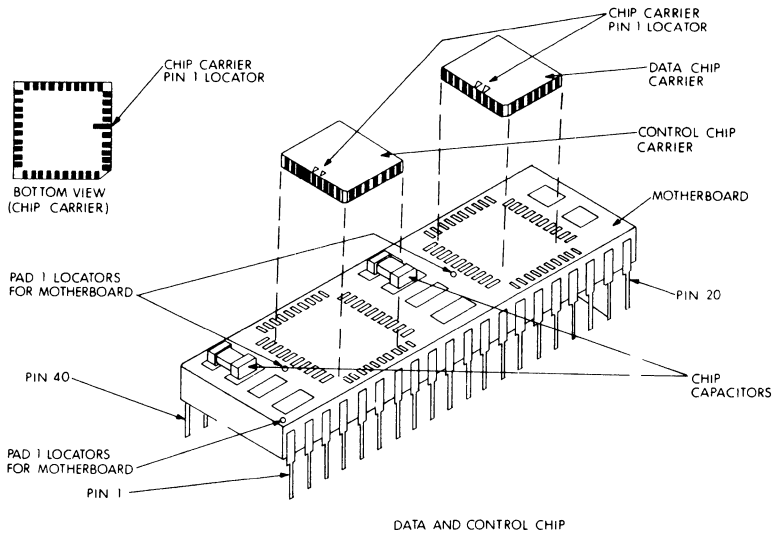
1. Performs all arithmetic and logical functions.
2. Handles all data and address transfers on the systems bus (except relocation, which is handled by the MMU).
3. Generates most of the signals used for interchip communication and external system control.

Control Chip

The control chip contains the microprogram sequence logic and 552 words of microprogram storage in programmable logic arrays (PLA) and read-only memory (ROM) arrays.

During the course of a normal microinstruction cycle, the control chip accesses the appropriate microinstruction in the PLA and ROM, sends it along the microinstruction bus (MIB) to the data and MMU chips for execution, and then generates the address for the next microinstruction to be accessed. The next address is constructed from either a next address field associated with the current microinstruction or, if a microprogrammed branch is to be executed, the target address contained within the microinstruction itself. The control chip operation is pipelined for better performance so that the next microinstruction is being accessed while the current one is being executed. The next address is then used in conjunction with various internal status and external service inputs to determine the microprogram sequence. The control chip accesses only its local storage. Additional control chips can be cascaded with external buffering to provide additional microstore.

Chip Select (CSEL) — CSEL is an open collector line which is routed to all MOS chips on the board except the MMU. The active control chip holds the line low. If a nonexistent control chip is selected by the mirocode, the line is pulled high. This causes a control chip error and a trap to location 10_8.



PDP-11/24 Data and Control Chip

MMU Chip

The MMU chip serves two purposes: it provides the memory management function, and provides storage for the FP11 floatng point accumulators and status registers. This chip provides dual mode (user and kernel) address reloaction of 18 or 22 bits. Sixteen-bit virtual addresses are received from the data chip via the data address lines (DAL), relocated to the appropriate 18- or 22-bit physical address, and then sent on the DAL to replace the original virtual address for transmission to the external system bus. The MMU chip contains the status registers and active page registers (PAR/PDR register pairs), as well as access protection and error detection capability. The MMU chip also provides the thirty-six 16-bit registers needed for operand storage, scratchpad areas, and status information storage during floating point operations.

The MMU chip is controlled information received on the MIB from both the data chip and the control chip, and by several discrete control inputs.

REGISTERS

The central processor contains nine registers which can be used as accumulators, index registers or stack pointers for temporary storage

of data. Six of these registers, R0-R5, are general registers which increase the speed of real-time data handling and facilitate multiprogramming. They can be used as accumulators or index registers for a real-time task or device. Another register, R7, is the PDP-11/24's program counter (PC). It is normally used for addressing purposes, not as an accumulator for arithmetic operations. This register contains the address of the next instruction to be executed. Two other registers (R6), are Processor Stack Pointers (SP). They maintain their respective hardware stacks, Kernel and User. Additional registers are reserved for internal machine use. Chapter 3 on addressing describes the functions and operations of the registers in more detail.

Stacks are used for nesting programs, creating re-entrant coding and temporary storage when a Last-In/First-Out structure is designed. For more information on programming uses of stacks, please refer to Chapter 5.

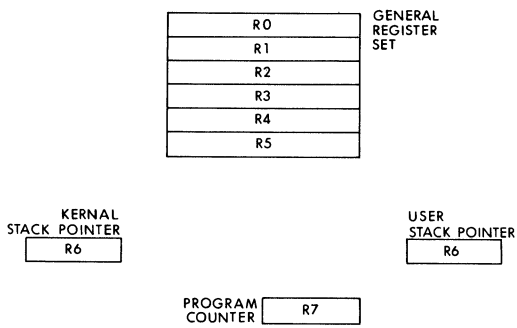


Figure 8-2 The General Registers

Processor Status Word

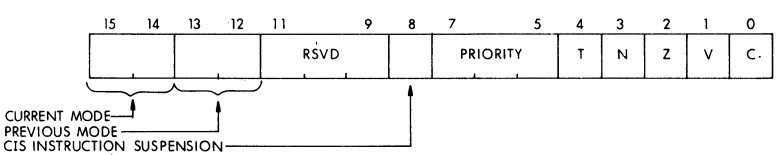


Figure 8-3 Processor Status Word

The Processor Status Word, located at 17 777 776, contains information on the current status of the PDP-11. This information includes

current processor priority; current and previous operational modes; condition codes describing the results of the last instruction; an indicator for detecting the execution of an instruction to be trapped during program debugging; and an indicator that is used to show that a CIS instruction was suspended by an interrupt.

Modes

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

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

When in User mode, a program is inhibited from executing a HALT instruction and the processor will trap through location 4 if an attempt is made to execute this instruction. Furthermore, the processor will ignore the RESET instruction, and execute No Operation. In Kernel mode, the processor will execute all instructions.

A program operating in Kernel mode can map users' programs anywhere in memory and thus explicitly protect key areas (including device registers and the Processor Status Word) from the User operating environment.

Processor Priority

The central processor operates at any of eight levels of priority, 0-7. When the CPU is operating at level 7, an external device cannot interrupt it with a request for service. The central processor must be operating at a lower priority than the priority of the external device's request 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.

Condition Codes

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

Trace Trap

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

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

Stack Limit

The PDP-11/24 has a Kernel Stack Overflow Boundary at location 400.

Once the Kernel stack goes below this boundary, the processor will complete the current instruction and then trap to location 4 (Stack Overflow).

MEMORY SYSTEM

Since MOS memory is volatile, meaning it depends on electricity to store information, a power loss or shutdown would cause data loss. To prevent this loss from occurring, a Battery Backup Unit (BBU) has been designed to temporarily preserve the contents in memory. This unit is available as an option on the PDP-11/24.

Generally, the incidence of ac line power loss varies inversely with the severity of loss. That is, there are an extremely small number of complete failures of ac power, and a relatively larger number of short-term failures or drops in voltage. Battery backup units are not intended to preserve data overnight or over weekends, but rather to overcome infrequent, short-term failures of ac power.

Memory Management

The Memory Management hardware is standard with the PDP-11/24 computer. This hardware relocation and protection facility can convert the 16-bit program virtual addresses to 22-bit addresses. The unit may be enabled or disabled under program control. There is a small speed advantage when in the 16-bit mode.

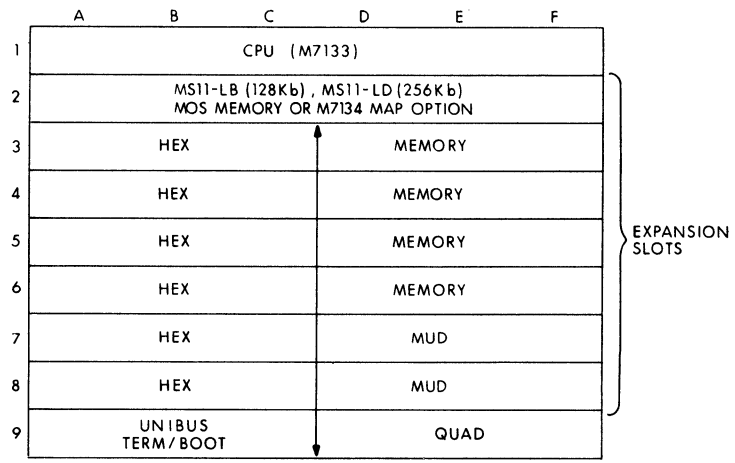
UNIBUS Map

The optional UNIBUS map responds as memory on the UNIBUS. It is the hardware relocation facility for converting the 18-bit UNIBUS ad-

dresses to 22-bit addresses. The relocation mapping may be enabled or disabled under program control.

PDP-11/24 BACKPLANE EXAMPLE

The PDP-11/24 backplane consists of nine slots. Slot 1 is reserved for the M7133 CPU module. Slot 2 can contain MS11-L memory or the UNIBUS MAP module. Additional memory can be configured in Slots 3-6. (In the 5.25 in. box, the total number of MS11-L memory modules cannot exceed three). The UNIBUS MAP is required for configurations with more than 256 kb of memory. Slot 9 contains either the M9312 bootstrap/terminator, the M9302 terminator (if the UNIBUS MAP option is installed), or the UNIBUS Cable.



The PDP-11/24 uses MS11-LB (128kb) or MS11-LD (256 kb) MOS parity memory. These have the following characteristics:

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

ASCII CONSOLE

The PDP-11/24 serial console is a standard feature which replaces the "lights and switches" programmer's console of earlier processors with logic that interprets ASCII characters to perform equivalent panel functions.

Physically, the I/O port used for the serial console function is shared with the standard system terminal (also called the "system console"), and is mode (or state) switchable by typing ASCII characters on the system terminal (the LA120 or equivalent which serves as the system console/programmer console).

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

NOTE

The console state can be entered only when the key switch is in the LOCAL position.

Console State

The processor is halted and the console state entered by pressing the BREAK key on the console terminal, or by placing the front panel HALT/CONTINUE/BOOT switch in the HALT position. The console state is also entered when the CPU executes the HALT instruction, provided Kernal mode HALTS are enabled. It can be accessed when the front panel key switch is in the LOCAL position. The BREAK is not passed to a running program, and console state is entered after printing the current output character, if any. While in the Console state, all input characters are interpreted as programmer's console commands.

The Console state is exited to the Program I/O state by typing a specific console command such as PROCEED or GO, or by Booting the processor.

Program I/O State

The Program I/O state is entered from the console state by typing the PROCEED or GO commands. Any ASCII character may be output by the program, and any ASCII character, may be input to the program. Any abnormal input character which causes a framing error, i.e., BREAK, will cause the processor to halt and re-enter the console state. Character echoing is the responsibility of the CPU software in Program I/O state.

The Program I/O state is exited to the Console state by typing the Console Break character, or by placing the HALT/CONTINUE/BOOT Switch in the HALT position.

Console defaults

Address defaults:	An 18-bit octal physical address is always assumed. Leading zeroes need not be typed.
Data defaults:	All transfers are 16-bit octal word transfers. Leading zeroes need not be typed.

CONSOLE ODT

The processor's microcode operates the serial line interface in half-duplex mode. Program I/O techniques are used rather than interrupts. When the console ODT microcode is busy printing characters using the transmit side of the interface, the microcode is not monitoring the receive side for incoming characters. Any characters coming in at this time are lost. The interface may post overrun errors, but the microcode does not check for any error bit in the interface. Therefore users should not type ahead to ODT because those characters are not recognized. In addition, if another processor is at the other end of the interface, it must obey half-duplex operation. No input characters should be sent until console ODT has finished outputting.

Console ODT Entry Conditions

1. Execution of HALT instruction in kernel mode, provided the HALT TRAP jumper is not installed.
2. Setting the 'HALT/CONTINUE/BOOT' switch to the HALT position while the CPU is running.
3. ODT is entered upon power-up if the HALT/CONTINUE/BOOT switch is in the HALT position and the keyswitch is in the LOCAL position.
4. Depressing 'BREAK' on the console terminal while the keyswitch is in the 'LOCAL' position.

Console ODT Input Sequence

Upon entry to console ODT, the RBUF register is read using a DATI and the character present in the buffer is ignored. This is done so that erroneous characters or user program characters are not interpreted by console ODT as a command, especially when a program is halted. The input sequence for console ODT is as follows:

1. Read and ignore character in RBUF.
2. Output a <CR><LF> to terminal.
3. Output contents of PC (program counter R7) in six digits to terminal.

4. Output a <CR><LF> to terminal.
5. Output the prompt character, @, to terminal.
6. Enter a wait loop for terminal input. The Done flag, bit 7 in RCSR, is tested using a DATI. If it is 0, the test continues.
7. If RCSR bit 7 is a 1, then low byte of RBUF is read using a DATI.

Console ODT Output Sequence

The output sequence for ODT is as follows:

1. Test XCSR byte 7 (Done flag) using a DATI and if a 0, continue testing.
2. If XCSR bit 7 is 1, write character to low byte of XBUF using a DATO (high byte is ignored by interface).

CONSOLE ODT COMMAND SET

The console ODT command set, listed below, is described in the following paragraphs. The commands are a subset of ODT-11 and use the same command characters. Console ODT has ten internal states. For each state only specific characters are recognized as valid inputs; other inputs invoke a "?" response. These states decrease the livelihood that an incorrect command will be permitted to damage user data.

Console ODT Commands

Command	Symbol	Use
Slash	/	Prints the contents of a specified location.
Carriage Return	<CR>	Closes an open location.
Line Feed	<LF>	Closes an open location and then opens the next contiguous location.
Internal Register Designator	$ or R	Opens a specific processor register.
Processor Status Word Designator	S	Opens the PS—must follow $ or R command.
Go	G	Starts program execution.

Proceed	P	Resumes execution of a program.
Binary Dump	Control-S	Manufacturing use only.
Toggle Halt	H	Allows the processor to be single-stepped.

The parity bit (bit 7) on all input characters is ignored (i.e., not stripped) by console ODT, and if the input character is echoed, the state of the parity bit is copied to the output buffer (XBUF). Output characters internally generated (e.g., <CR>) by ODT have the parity bit equal to 0. All commands are echoed except for <LF>. Where applicable, uppercase and lowercase command characters are recognized (with the exception of 'H', which must be uppercase).

In order to describe the use of a command, other commands are mentioned before they have been defined. For the novice user, these paragraphs should be scanned first for familiarization and then reread for detail. The word "location," as used in this paragraph, refers to a bus address, processor register, or processor status word (PS).

NOTE

In the following examples, the user's entry is in bold faced (dark) type, while the response from the processor is not.

/(ASCII 057) Slash

This command is used to open a bus address, processor register, or processor status word and is normally preceded by other characters which specify a location. In response to /, console ODT prints the contents of the location (i.e., six characters) and then a space (ASCII 40). After printing is complete, console ODT waits for either new data for that location or a valid close command. The space character is issued so that the location's contents and possible new contents entered by the user are legible on the terminal.

Example: @**001000/**012525<SPACE>

where:

@ = console ODT prompt character.

001000 = octal location in the bus address space desired by the user (leading 0s are not required).

= command to open and print contents of location.

012525 = contents of octal location 1000.

<SPACE> = space character generated by console ODT.

The / command can be used without a location specifier to verify the data just entered into a previously opened location. The / is recognized only if it is entered immediately after a prompt character. A / issued immediately after the processor enters ODT mode causes a ?<CR><LF> to be printed because a location has not yet been opened.

Example: @**1000/**012525<SPACE> 1234 **<CR>**<CR><LF>
@**/**001234<SPACE>

where:

first line = new data of 1234 entered into location 1000 and location closed with <CR>

second line = a / was entered without a location specifier and the previous location was opened to reveal that the new contents were correctly entered into memory.

<CR>(ASCII 15) Carriage Return

This command is used to close an open location. If a location's contents are to be changed, the user should precede the <CR> with the new data. If no change is desired, <CR> closes the location without altering its contents.

Example: @**R1/**004321<SPACE> **<CR>** <CR><LF>
@

Processor register R1 was opened and no change was desired so the user issued <CR>. In response to the <CR>, console ODT printed <CR> <LF>@.

Example: @**R1/**004321<SPACE> **1234 <CR>** <CR><LF>
@

In this case the user desired to change R1, so new data, 1234, were entered before issuing the <CR>. Console ODT deposited the new data in the open location and then printed <CR><LF>@.

Console ODT echoes the <CR> entered by the user and then prints an additional <CR>, followed by a <LF>, and @.

<LF> (ASCII 12) Line Feed
This command is used to close an open location and then open the next contiguous location. Bus addresses and processor registers are incremented by 2 and 1 respectively. If the PS is open when a <LF> is issued, it is closed and a <CR><LF>@ is printed; no new location is opened. If the open location's contents are to be changed, the new data should precede the <LF>. If no data are entered, the location is closed without being altered.

Example: @**R2/**123456<SPACE> **<LF>** <CR><LF>
R3/054321<SPACE>

In this case, the user entered <LF> with no data preceding it. In response, console ODT closed R2 and then opened R3. When a user has the last register, R7, open, and issues <LF>, console ODT opens the beginning register, R0. When the user has the last bus address open of a 32K-word segment and issues <LF>, console ODT opens the first location of that same segment. If the user wishes to cross the 32K-word boundary, he must re-enter the address for the desired 32K-word segment (i.e., console ODT is module 32K-word). This operation is the same as that found on all other PDP-11 consoles.

Example: @**R7/**000000<SPACE> **<LF>** <CR><LF>
R0/123456<SPACE>

or

@**577776/**000001<SPACE> **<LF>** <CR><LF>
400000/125252<SPACE>

Unlike other commands, console ODT does not echo the <LF>. Instead it prints <CR>, then <LF> so the printing terminals opeate properly. In order to make this easier to decode, console ODT does not echo ASCII 0, 2, or 10, either, but responds to these three characters with ?<CR><LF>@.

$ (ASCII 044) or R (ASCII 122) Internal Register Designator
Either character when followed by a register number, 0 to 7, or PS designator, S, will open that specific processor register.

The $ character is recognized to be compatible with ODT-11. The R character was introduced for the convenience of one key stroke and because it is representative of what it does.

Example: @**$0/**000123<SPACE>

or

@**R7/**000123<SPACE> **<LF>**
RO/054321<SPACE>

If more than one character is typed (digit or S) after the R or $, console ODT uses the last character as the register designator. There is an exception, however: if the last three digits equal 077 or 477, ODT interprets it to mean the PS rather than R7.

S (ASCII 123) Processor Status Word

This designator is for opening the PS (processor status word) and must be employed after the user has entered an R or $ register designator.

Example: @**RS/**100377<SPACE> 0 **<CR>** <CR><LF>
@/000020<SPACE>

Note the trace bit (bit 4) of the PS cannot be modified by the user. This is done so that PDP-11 program debug utilities (e.g., ODT-11), which use the T bit for single-stepping, are not accidentally harmed by the user.

If the user issues a <LF> while the PS is open, the PS is closed and ODT prints a <CR><LF>@. No new location is opened in this case.

G (ASCII 107) Go

This command is used to start program execution at a location entered immediately before the G. This function is equivalent to the LOAD ADDRESS and START switch sequence on other PDP-11 consoles.

Example: @**200 G**<NULL><NULL>

The console ODT sequence for a G, after echoing the command character, is as follows:

1. Print two nulls (ASCII 0) so the bus initialization that follows does not flush the G character from the double-buffered UART chip in the serial line interface.
2. Load R7 (PC) with the entered data. If no data are entered, 0 is used. (In the above example, R7 is set equal to 200 and that is where program execution begins.)
3. The PS and floating point status register contained in the MMU are cleared to 0.
4. The bus is initialized.

5. The service state is entered by the processor. If there is anything to be serviced, it is processed. If the HALT signal is asserted, the processor re-enters the console ODT state. This feature is used to initialize a system without starting a program (R7 is altered). If the user wants to single-step a program, it can be executed by issuing a G and then successive P commands, all done with the HALT signal asserted, (either by the HALT switch or via the 'H' command).

P (ASCII 120) Proceed

This command is used to resume execution of a program and corresponds to the CONTINUE switch on other PDP-11 consoles. No programmer-visible machine state is altered using this command.

Example: @**P**

The PDP-11 processor is started immediately after the transmission of the P to the terminal console has begun. If a RESET instruction is executed while the P is transmitting, the echo of the P may be lost.

Program execution resumes at the address pointed to by R7. After the P is echoed, the console ODT state is left and the processor immediately fetches the next instruction. If the HALT signal is asserted, it is recognized at the end of the instruction (during the service state) and the processor enters the console ODT state. Upon entry, the content of the PC (R7) is printed. In this fashion, a user can single-instruction step through a program and get a PC "trace" displayed on his terminal.

Control-S (ASCII 23) Binary Dump

This command is used for manufacturing test purposes and is not a normal user command. It is described here to explain the machine's response if accidentally invoked. It is intended to more efficiently display a portion of memory compared to using the / and <LF> commands. The protocol is as follows:

1. After a prompt character, console ODT receives a control-S command and echoes it.
2. The host system at the other end of the serial line must send two 8-bit bytes which console ODT interprets as a 16-bit starting address. These two bytes are not echoed.

   The first byte specifies starting address <15:8> and the second byte specifies starting address <7:0>. Bus address bits <17:16> are always forced to be 0; the dump command is restricted to the first 32K words of address space.
3. After the second address byte has been received, console ODT

outputs 10 bytes to the serial line starting at the address previously specified. When the output is finished, console ODT prints <CR><LF>@.

If a user accidentally enters this command, it is recommended, in order to exit from the command, that two @ characters (ASCII 100) be entered as a starting address. After the binary dump, an @ prompt character is printed.

H(ASCII110)Toggle HALT Flip-Flop

Programs are often debugged by single-instruction stepping them. In the PDP-11/24, this may be accomplished by setting the HALT/BOOT switch to the HALT position and using the "P" command to single-instruction-cycle the PDP-11/24. The same result as setting the HALT switch may be realized by typing "H". An internal flip-flop is set, simulating the action of the HALT switch. "P" will now single-cycle the PDP-11/24. After the debugging execution is completed, "H" must be typed once more. This will clear the internal flip-flop, and the next "P" will cause the PDP-11/24 to run at full speed.

ADDRESS SPECIFICATION

All addresses must be entered by users with all 18 bits specified, regardless of whether the MMU is present or not. For example, if a user desires to open the RCSR of Serial Line Unit 1, the user must enter 777560, not 177560. Eighteen-bit addresses must also be used to access memory greater than 32K words. Leading zeroes need not be typed.

Processor I/O Addresses

Certain processor and MMU registers have I/O addresses assigned to them for programming purposes. If referenced in console ODT, the PS responds to its bus address, 777776. Processor registers R0 through R7 do not respond to bus addresses 777700 through 777707 if referenced in console ODT (i.e., time-out occurs).

The MMU contains status registers and PAR/PDR pairs. Any of these registers can be accessed from console ODT by entering its bus address.

Example: @**777572**/000001<SPACE>

In this case, memory management status register 0 is opened and the memory management enable is seen to be set. The internal display register (777570) can not be accessed with ODT because the register is Write-only.

Stack Pointer Selection

Accessing Kernel and User stack pointer registers is accomplished in the following way: Whenever R6 is referenced in ODT, it accesses the stack pointer specified by the PS current mode bits (PS<15:14>). This is done for convenience. If a program operating in Kernel mode (PS<15:14>=00) is halted and R6 is opened, the Kernel stack pointer is accessed.

Similarly, if a program is operating in User mode, R6 accesses the user stack pointer. If a specific stack pointer is desired, PS<15:14> must be set by the user to the appropriate value and then the R6 command can be used. If an operating program has been halted, the original value of PS<15:14> must be restored in order to continue execution.

Example: PS = 140000
@**R6/**123456<SPACE>

The user mode stack pointer has been opened.

@**RS/**140000<SPACE> **0 <CR>** <CR><LF> (switch to Kernel mode)
@**R6/**123456<SPACE> **<CR>** <CR><LF> (read the Kernel Stack Pointer)

@**RS/**000000<SPACE> **140000<CR>** <CR><LF> (return to User mode)

@**P**

In this case, the Kernel mode stack pointer was desired. The PS was opened and PS<15:14> was set to 00 (Kernel mode). Then R6 (Kernel Stack Pointer) was examined and closed. The original value of PS<15:14> was restored and then the program was continued using the P command.

If PS<15:14> is set to 01, another unique register exists in the processor, but is reserved for future DIGITAL use.

The floating point accumulators, which are also in the MMU chip, cannot be accessed from console ODT. Only floating point instructions can access these registers.

Entering Octal Digits

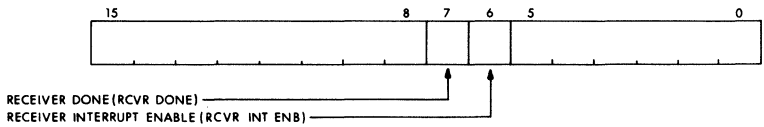
When the user is specifying an address of data, console ODT will use the last six octal digits if more than six have been entered. The user need not enter leading 0s for either address or data; console ODT forces 0s as the default. If an odd address is entered, the low-order bit is ignored and a full 16-bit word is displayed.

ODT Time-Out
If the user specifies a nonexistent address, console ODT responds to the error by printing ?<CR><LF>@.

Invalid Characters
Console ODT will recognize, with the exception of "H", uppercase and lowercase characters as commmands. Any character that console ODT does not recognize during a particular sequence is echoed (with the exception of ASCII 0, 2, 10, or 12 as noted earlier) and console ODT prints a ?<CR><LF>@. Console ODT has ten internal states, each of which has its own set of valid input characters. When in a particular state, only commands specific to that state are valid. This was done to lower the probability of a user unintentionally destroying a program by pressing the wrong key.

TERMINAL SERIAL LINE UNIT REGISTERS (SLU 1)

Receiver Status Register (TERM RCSR) 17 777 560



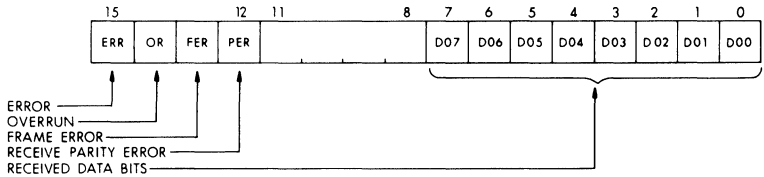
Bit: 15:8
Function: Unused

Bit: 7 (Read-only)
Name: RECEIVER DONE
Function: Set when an entire character has been received and is ready for transfer to the UNIBUS. Cleared by addressing (READ or WRITE) RBUF or by INIT. Starts an interrupt sequence when RECEIVER INTERRUPT ENABLE (bit 6) is also set.

Bit: 6 (Read/write)
Name: RECEIVER INTERRUPT ENABLE
Function: Cleared by INIT. Starts an interrupt sequence when RECEIVER DONE is set.

Bit: 5:0
Function: Unused

Receiver Data Buffer (TERM RBUF) 17 777 562



Bit: 15 (Read-only)
Name: ERROR
Function: Logical OR of OVERRUN ERROR, FRAMING ERROR and PARITY ERROR. Cleared by removing the error conditions. ERROR is not tied to the interrupt logic, but RECEIVER DONE is.

Bit: 14 (Read-only)
Name: OVERRUN
Function: Set if previously received character is not read (RECEIVER DONE not reset) before the present character is received. This indicates that the previous character(s) have been lost.

Bit: 13 (Read-only)
Name: FRAMING ERROR
Function: Set if the character read has no valid stop bit. Also used to detect break.

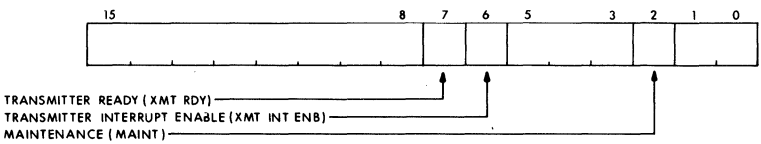
Bit: 12 (Read-only)
Name: RECEIVE PARITY ERROR
Function: Set if received parity does not agree with the expected parity. Always 0 if no parity is selected.

NOTE: Error conditions remain present until the next character is received, at which time the error bits are updated. INIT does not necessarily clear the error bits.

Bit: 11:8
Function: Unused

Bit: 7:0 (Read-only)
Name: RECEIVED DATA
Function: These bits contain the character just read. If less than eight bits are selected, the buffer will be right-justified with the unused bits read as 0s. Not cleared by INIT.

Transmitter Status Register (TERM XCSR) 17 777 564



Bit: 15:8
Function: Unused

Bit: 7 (Read-only)
Name: TRANSMITTER READY
Function: Set by INIT. Cleared when XBUF is loaded; set when XBUF can accept another character. When set, it will start an interrupt sequence if TRANSMITTER INTERRUPT ENABLE is also set.

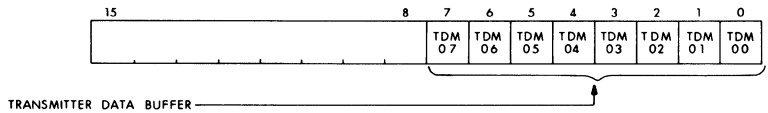
Bit: 6 (Read/write)
Name: TRANSMITTER INTERRUPT ENABLE
Function: Cleared by INIT. When set it will start an interrupt sequence if TRANSMITTER READY is also set.

Bit: 5:3
Function: Unused

Bit: 2 (Read/write)
Name: MAINTENANCE
Function: Cleared by INIT. When set, it disables the external serial line input to the RECEIVER and selects instead the serial output of the TRANSMITTER. This allows diagnostic programs to exercise the serial line.

Bit: 1:0
Function: Unused

Transmitter Data Buffer (TERM XBUF) 17 777 566

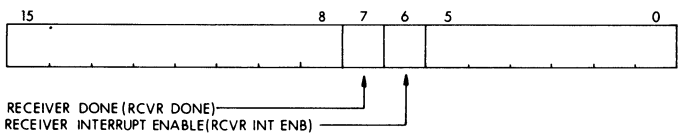


Bit: 15:8
Function: Unused

Bit: 7:0 (Write-only)
Name: TRANSMITTER DATA BUFFER
Function: If less than eight bits are selected, then the character must be right-justified. The character to be transmitted is written into this register.

SECOND SERIAL LINE UNIT REGISTERS (SLU 2)

Receiver Control/Status Register (RCSR) 17 776 500



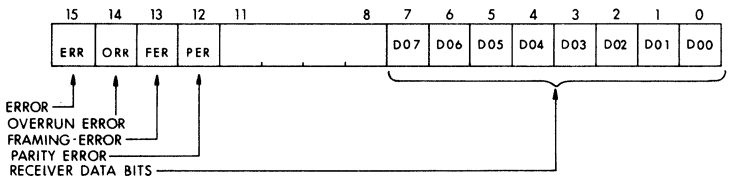
Bit: 15:8
Function: Unused

Bit: 7 (Read-only)
Name: RECEIVER DONE
Function: Set when a complete character is contained in the RBUF. Cleared when the RBUF is addressed or when an INITIALIZE operation occurs. Initiates the interrupt sequence when the RECEIVER INTERRUPT ENABLE (bit 6) is set.

Bit: 6 (Read/Write)
Name: RECEIVER INTERRUPT ENABLE
Function: Set by program to allow the interrupt sequence to be initiated by the RECEIVER DONE bit, 7.

Bit: 5:0
Function: Unused

Receiver Buffer Register (RBUF) 17 776 502



Bit: 15 (Read-only)
Name: ERROR
Function: Set when the OR ERROR (bit 14), FR ERROR (bit 13) or the PAR ERROR (bit 12) is set. Cleared by the reception of new and correct data.

Bit: 14 (Read-only)
Name: OVERRUN ERROR
Function: Set if the character in the RBUF has not been read before another character is received. Cleared by an INITIALIZE operation or when the RBUF is emptied.

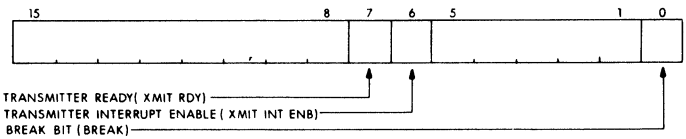
Bit: 13 (Read-only)
Name: FRAMING ERROR
Function: Set when the character read in RBUF does not have a valid stop bit(s). Cleared when a valid character is received. This bit may indicate an error in transmission or the reception of a BREAK character.

Bit: 12 (Read-only)
Name: PARITY ERROR
Function: Set when the parity of the character read in the RBUF is incorrect relative to the parity mode selected. Cleared when the parity of the next character is validated.

Bit: 11:8
Function: Unused

Bit: 7:0 (Read-only)
Name: RECEIVED DATA
Function: These bits are the data character received from the SLU 2.

Transmitter Control/Status Register (XCSR) 17 776 504



Bit: 15:8
Function: Unused

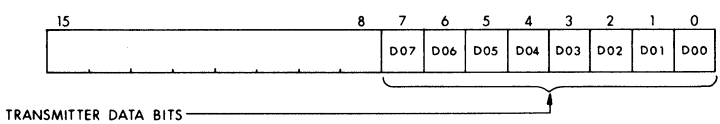
Bit: 7 (Read-only)
Name: TRANSMITTER READY
Function: Set when the XBUF is ready to accept a character or when an initialize operation occurs. Setting the bit indicates an interrupt sequence if the TRANSMITTER INTERRUPT ENABLE (bit 6) is set. Cleared when a character is written into the XBUF.

Bit: 6 (Read/write)
Name: TRANSMITTER INTERRUPT ENABLE
Function: Set by program. Enables the interrupt sequence to be initiated if the TRANSMITTER READY (bit 7) is set. Cleared by the program.

Bit: 5:1
Function: Unused

Bit: 0 (Read/write)
Name: BREAK
Function: Set by the program. Causes a space to be continuously transmitted to the SLU2. If maintained long enough, the SLU at the 'far' end will detect a framing error and interpret this as 'BREAK'. Cleared by the program.

Transmitter Data Buffer Register (XBUF) 17 776 506

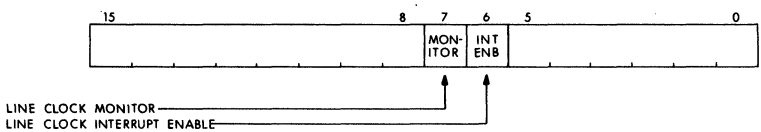


Bit: 15:8
Function: Unused

Bit: 7:0 (Write-only)

Name: TRANSMITTER DATA
Function: These bits are the data character to be transferred to the SLU 2.

Clock Status Register (LKS) 17 777 546



Bit: 15:8
Function: Unused

Bit: 7 (Read/clear)
Name: LINE CLOCK MONITOR
Function: Set only by the line frequency clock signal and cleared only by the program or the Line Clock interrupt sequence. Set by INIT.

Bit: 6 (Read/write)
Name: LINE CLOCK INTERRUPT ENABLE
Function: Cleared by INIT. When set, starts an interrupt sequence if LINE CLOCK MONITOR is also set.

Bit: 5:0
Function: Unused

ADDRESS AND VECTOR ASSIGNMENTS
Integral to the PDP-11/24 CPU are two serial line ports and a real-time clock. The serial line units follow the same address and vector assignments as the KL11, DL11-A, B, C, D and W. The addresses and vectors are fixed for all three devices.

	Address	Vector	Priority
Console	17 777 560	60/64	BR4
(SLU #1)	17 777 562		
	17 777 564		
	17 777 566		
	17 776 500	300/304	BR4
(SLU #2)	17 776 502		
	17 776 504		
	17 776 506		
Line Clock	17 777 546	100	BR6

TIMING CONSIDERATIONS

Receiver

The RECEIVER DONE flag sets when the UART has assembled a full character, which occurs at the middle of the first stop bit.

Since the UART is double buffered, data remain valid until the next character is received and assembled. This allows one full character time for servicing the RECEIVER DONE flag.

NOTE

The UART (Universal Asynchronous Receiver/Transmitter) is an asynchronous subsystem. The transmitter accepts parallel characters and converts them to a serial asynchronous output. The receiver accepts asynchronous serial characters and converts them to a parallel output.

Transmitter

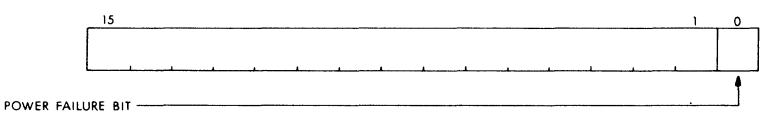
The UART's transmitter section is also double buffered. After initialization, the TRANSMITTER READY flag is set. When the buffer is loaded with the first character, the flag clears but sets again within a fraction of a bit time. A second character can then be loaded, clearing the flag again. The flag then remains clear for nearly a full character time.

Break Generation

Setting the break bit causes the transmission of a continuous space. Since the TRANSMITTER READY flag continues to function normally, the duration of break can be timed by the "pseudo-transmission" of a number of characters. However, since the transmitter is double buffered, a null character (all zeros) should precede transmission of break to insure the previous character clears the line. Likewise, the last "pseudo-transmitted" character under break should be null.

REGISTERS

CPU Error Register 17 777 766

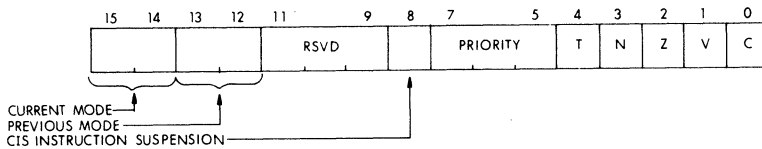


This register is available only when the optional UNIBUS map is installed. The CPU Error Register contains one bit, bit 0. This bit, when

set, indicates that one or more power supply voltages have exceeded their tolerance. This bit is set when voltage error occurs and is cleared either by RESET or by writing a zero to the bit.

Bit: 0 **Name:** CPU Power failure
Function: (see explanation above)

Processor Status Word 17 77 776



The Processor Status Word contains information on the current status of the CPU. This information includes current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; an indicator for detecting the execution of an instruction to be trapped during program debugging; and an indicator to determine whether a commercial instruction was in progress.

Processor Traps
These are a series of errors and programming conditions which will cause the central processor to trap to a set of fixed locations. These include Power Failure, Stack Errors, Timeout Errors (Non-Existent Memory References), Memory Errors, Memory Management Violations, Floating Point Processor Exception Traps, use of Reserved Instructions, use of the T bit in the Processor Status Word, and use of the IOT, EMT, BPT, and TRAP instructions.

Power Failure
Whenever ac power drops below 90 volts for 120V power (180 volts for 240V) or outside a limit of 47 to 63 Hz, as measured by dc power, the powerfail sequence is initiated. The central processor automatically traps through location 24 and the user's powerfail program has approximately 5 msec to save all volatile information (data in registers; I/O status, etc.), and to condition peripherals for power failure.

If battery backup is present, and batteries are not depleted when power is restored, the processor again traps to location 24 and executes the user's power-up Kernel routine to restore the machine to its state prior to power failure. If batteries are not present, a boot to default device is executed.

Time-Out Error
This error occurs when a MSYN pulse is placed on the UNIBUS and there is no SSYN pulse within 20 μsec. This error usually occurs in attempts to address non-existent memory or peripherals.

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

Reserved Instruction
There is a set of illegal and reserved opcodes which cause the processor to trap through location 10. An example would be an attempt to execute a floating-point instruction when no floating-point processor is present.

Trap Handling
Appendix A includes a list of the reserved Trap Vector locations and System Error Definitions which cause processor traps. When a trap occurs, the processor follows the same procedure for traps as it does for interrupts (saving the PC and PS on the new Processor Stack, etc.).

In cases where multiple traps and interrupts occur concurrently, the processor will service the conditions according to the priority sequence illustrated.

Trap Priorities
1. DCLO (Power up)
2. Reserved Instruction Trap
3. Memory Management Fault
4. Bus Error Traps
5. Memory Parity Errors
6. Trace Trap
7. Stack Overflow Trap
8. Power Fail Trap
9. Bus Request (BR) level 7
10. Line Clock (Highest B6 Device)
11. BR 6
12. BR 5
13. BR 4
14. HALT
15. WAIT LOOP

Stack Limit Violations
When instructions cause the Kernel R6 to exceed (go lower than) 400_8, a Stack Violation occurs. Operations that cause a Stack Viola-

tion are completed, then a bus error trap is effected (Trap to 4). The error trap, which itself uses the stack, executes without causing an additional violation.

PDP-11/24 CPU and I/O Device Registers and Addresses

Address	Register
17 777 776	Processor Status Word (PSW)
17 777 766	CPU Error (Optional with UNI-BUS map)
17 777 707 — 17 777 700	CPU General Register (not accessible by address)
17 777 656 — 17 777 640	User Instruction PAR, Reg. 0-7
17 777 616 — 17 777 600	User Instruction PDR, Reg. 0-7
17 777 576	MM Status Register 2 (SR2)
17 777 574	MM Status Register 1 (SR1)
17 777 572	MM Status Register 0 (SR0)
17 777 570	Display Register
17 777 566 — 17 777 560	Console Terminal SLU
17 776 500 — 17 776 506	SLU 2
17 772 516	MM Status Register 3 (SR3)
17 772 356 — 17 772 340	Kernel Instruction PAR, Reg. 0-7
17 772 316 — 17 772 300	Kernel Instruction PDR, Reg. 0-7
17 770 372 — 17 770 200	UNIBUS MAP Registers (optional with UNIBUS map)

SPECIFICATIONS

Packaging

A basic PDP-11/24 consists of either a 5.25 in or a 10.5 in box with a 9-slot backplane, power supply, CPU, 128 Kbyte or 256 Kbyte memory.

There are prewired areas within the backplane for expansion with optional equipment.

Component Parts
The basic PDP-11/24 system contains:

- **Standard Equipment**

PDP-11/24 CPU
Memory Management
Bootstrap Loader
Line Frequency Clock
Second Serial Line Interface
Terminal Interface
128 Kbyte or 256 Kbyte Parity MOS Memory
BA11-L or BA11-A Box with Power Supply

- **Prewired Expansion Space for Optional Equipment**
  Floating Point Unit
  SPC Slots for Peripherals, up to 6 Hex and 1 Quad (depending on the memory configuration).
  768 Kbyte Parity MOS Memory (up to 1,024 Kbytes maximum)
  4 system units of open space in BA11-A Box

OTHER SPECIFICATIONS

AC Power
5¼" Box:

104-127 Vrms, 47-63 Hz, 1 phase power, 5 amps rms maximum @ 120 Vac

208-258 Vrms, 47-63 Hz, 1 phase power, 2.5 amps rms maximum @ 240 Vac

10½" Box:

90-128 Vrms, 47-63 Hz, 1 phase power, 16 amps rms maximum @ 120 Vac

180-256 Vrms, 47-63 Hz, 1 phase power, 9 amps rms maximum @ 240 Vac

The Mounting Box

Size
5¼" Box:

Cabinet is 13.5 cm high × 48 cm wide × 69 cm deep (5.25" × 19" × 25")

10½" Box:

Cabinet is 26.3 cm high × 42.4 cm wide × 66 cm deep (10.35" × 16.62" × 26")

Weight

5¼" Box:

20 Kg (45 lbs.)

10½" Box:

32 Kg (70 lbs.)

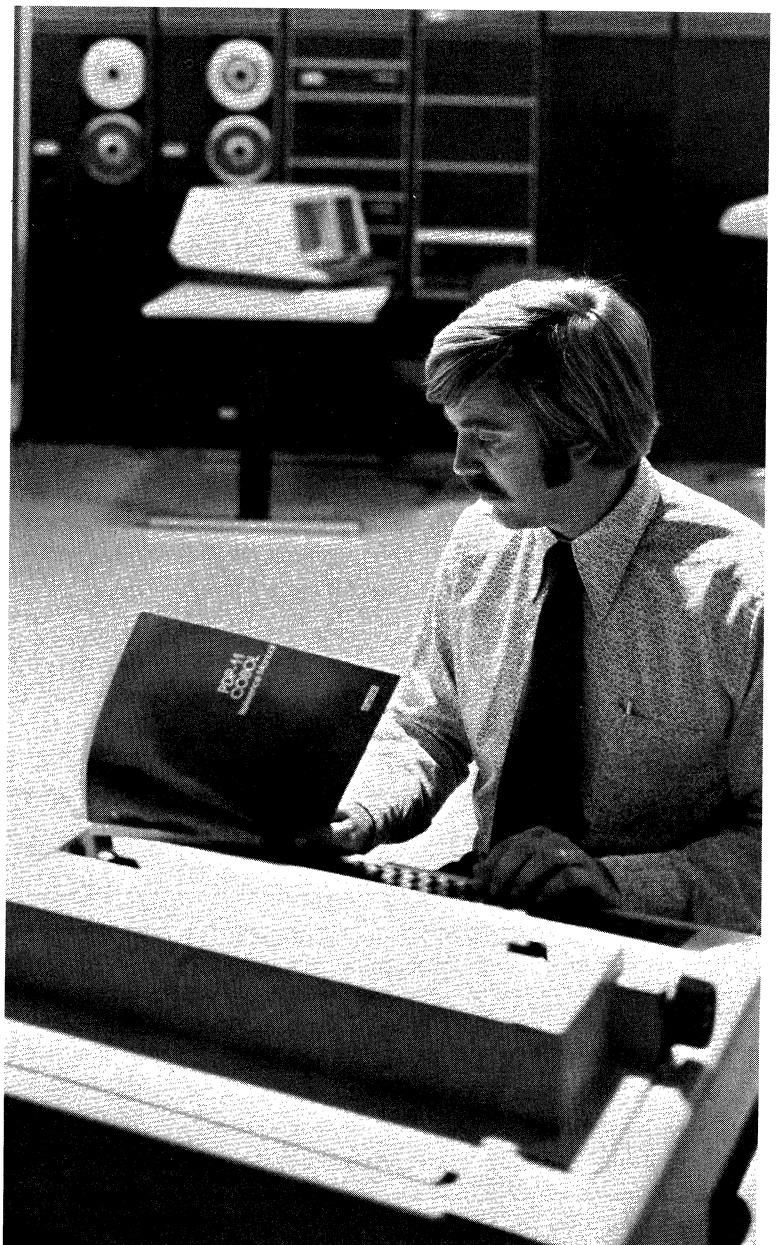
Operating Environment

The 5¼" and 10½" CPU boxes have the same operating and nonoperating environment specifications.

Temperature:	5°C to 50°C (41°F to 122°F)
Humidity:	10% to 95% with max. wet bulb of 32°C (89.6°F) and minimum dew point of 2°C (36°F)
Altitude:	To 2.4 Km (8000 ft.) noncondensing.

Nonoperating Environment

Temperature:	−40°C to 80°C (−40° to 176°F)
Humidity:	To 95% (noncondensing)
Altitude:	To 9.1 Km (30,000 ft.)



CHAPTER 9
PDP-11/44

THE MID-RANGE MINICOMPUTER STANDARD

The PDP-11/44, a fourth generation mid-range PDP-11, offers high levels of functionality and performance for a machine in its price range. Many large, high-performance features such as a high-speed central processor, support of one megabyte memory and large 8,192 byte high-speed cache memory are standard on the PDP-11/44. Available options include the Floating Point Processor, Commercial Instruction Set Processor and the Battery Backup Unit. The PDP-11/44 provides more system up-time since it is designed to meet a rigorous Reliability, Availability and Maintainability Program (RAMP).

FEATURES

Integral to the PDP-11/44 central processor unit are these hardware features and expansion capabilities:

- Cache memory organization to provide very fast program execution speed and high system throughput
- Extended Instruction Set (EIS) for faster integer arithmetic execution
- Memory management for relocation and protection in multiuser, multitask environments
- Intelligent ASCII console with which the user can operate the computer without requiring access to the front panel
- TU58 cartridge tape interface port to make it easier to load software patches or Field Service diagnostic programs
- Ability to access up to 1 million bytes of main memory (1 byte = 8 bits) provides ample memory space for application programs
- Real-time clock which provides KW11-L compatible line-frequency clock
- 256 Kbyte ECC MOS main memory modules provide high-density, low-cost memories with error correcting codes to insure better memory reliability
- Integral DL11-W serial line unit capability
- Optional remote diagnosis
- Optional KE44A Commercial Instruction Set for faster COBOL execution
- Optional FP11-F Floating Point Processor with advanced features, operating with 32-bit and 64-bit numbers for faster FORTRAN or BASIC execution

- Optional battery backup unit for data integrity during most power outages

SYSTEM ARCHITECTURE

The PDP-11/44 is a medium scale general-purpose computer which uses an enhanced, upwardly compatible version of the basic PDP-11 architecture. A block diagram is shown in Figure 9-1.

Memory Management is standard with the basic computer, allowing expanded memory addressing, relocation, and protection. Also standard is a UNIBUS Map which translates 18-bit UNIBUS addresses to 22-bit physical memory addresses. The cache contains 8,192 bytes of fast, static MOS memory that buffers the processor data from main memory.

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

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

CENTRAL PROCESSOR

The PDP-11/44 processor acts as the arbitration unit for UNIBUS control by regulating bus requests and transferring control of the bus to the requesting device with the highest priority.

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

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

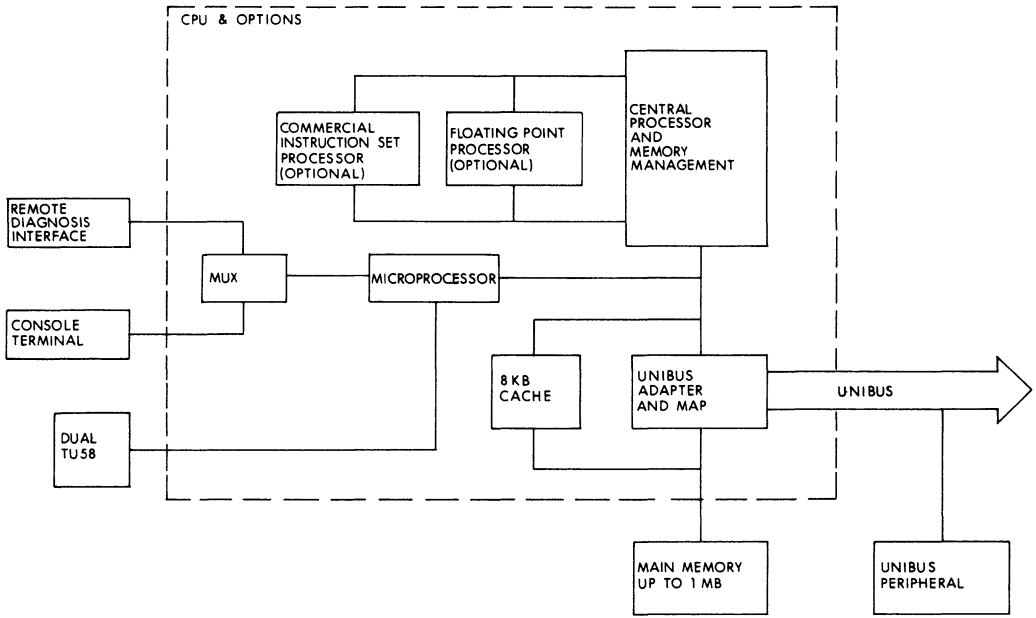


Figure 9-1 PDP-11/44 Block Diagram

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

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

General Registers

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

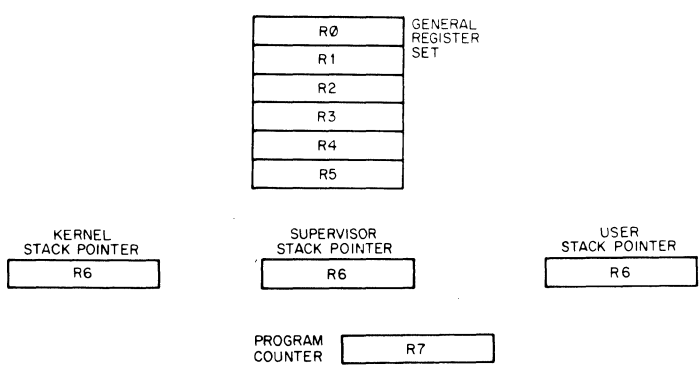


Figure 9-2 The General Registers

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

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

The remaining six registers are R0-R5.

Registers can be used to increase the speed of real-time data handling or facilitate multiprogramming. The six general registers could each be used as an accumulator or index register for a real-time task or device.

Processor Status Word

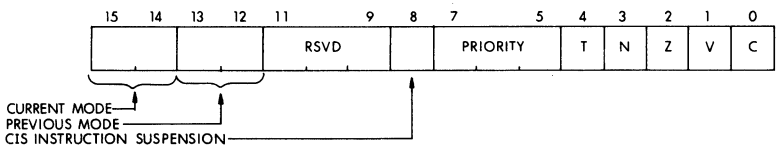


Figure 9-3 Processor Status Word

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

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

Modes

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

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

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

A program operating in Kernel mode can map users' programs anywhere in memory and thus explicitly protect key areas (including the device's registers and the Processor Status Word) from the User operating environment.

Processor Priority

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

Condition Codes

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

Trace Trap

The trace trap bit (T) can be set or cleared under program control. When set, a processor trap will occur through location 14 on completion of instruction execution and a new Processor Status Word and program control will be loaded. This bit is especially useful for debug-

ging programs as it provides an efficient method of installing breakpoints.

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

Stack Limit

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

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

MEMORY SYSTEM

MOS Memory with ECC

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

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

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

MOS memory is volatile. It depends on electricity to store information. Since a power loss or shutdown would cause data loss, battery backup

units (BBU) are designed to temporarily preserve contents in memory. These units are available as options on the PDP-11/44.

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

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

Memory Management

The Memory Management hardware is standard with the PDP-11/44 computer. It is a hardware relocation and protection facility that can convert the 16-bit program virtual addresses to 22-bit addresses. The unit may be enabled or disabled under program control. There is a small speed advantage when in the 16-bit mode.

UNIBUS Map

The UNIBUS map is the hardware relocation facility for converting the 18-bit UNIBUS addresses to 22-bit addresses. The relocation mapping may be enabled or disabled under program control.

CACHE MEMORY

PDP-11/44 Cache Specification and Design Description

An overall block diagram of the PDP-11/44 is shown in Figure 9-1. Functionally, main memory and the cache can be treated as a single unit of memory.

Introduction

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

Physical Description

The PDP-11/44 cache memory interfaces to the processor through the processor backplane. Two user-accessible switches (S1 and S2) enable the cache to be shut off by causing a forced-miss condition in either upper or lower cache address space. Software bits for enabling or disabling cache are also provided in the MMU registers, discussed later in this chapter.

General System Architecture

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

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

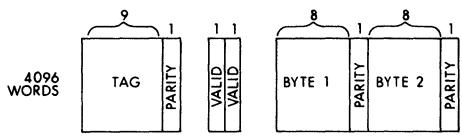
Figure 9-4 Cache Response Matrix

The response of the cache to a CPU read bypass hit is jumper selectable. In its normal configuration, jumper W1 (M7097 module) is in and jumper W2 is removed to allow a forced miss only to occur for a CPU read hit bypass. If the PDP-11/44 and the KK11-B cache are to be used in a multiported memory system, jumper W1 is removed and jumper W2 is inserted, to allow a CPU read hit with bypass to cause an

invalidation to occur to that location. This allows the software to clean potentially stale cache data that might arise in a multiported memory system.

Cache Memory Organization

The cache memory array consists of 30 4,096 × 1 RAM chips, arranged as follows:



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

I/O PAGE REGISTERS

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

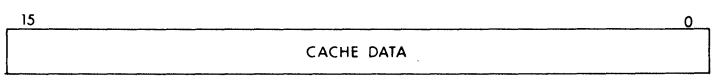


Figure 9-5 17 777 754 Cache Data Register (CDR)

Bit: 15:0 (Read-only)
Name: Cache Data Register Bits
Function: These bits are loaded from the 16-bit data array section of the cache RAM on every read access to main memory space, except the top 256K bytes, which are reserved for the UNIBUS address space. This register can be used with the Hit on Destination Only bit to aid the cache diagnostics in identifying failures in the data section of the cache array.

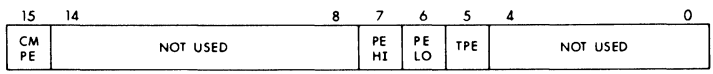


Figure 9-6 17 777 744 Cache Memory Error Register (CMER)

Bit: 15 **Name:** Cache Memory Parity Error (CMPE)
Function: Set if a cache parity error is detected while the cache parity abort, bit 7, is set, or if a memory parity error occurs. If set, cache will force a miss. Cleared by any write to the CME Register or by console INIT. This bit must be cleared before the Disable Cache Parity Interrupt (DCPI) is cleared. If the cache detects a parity error in itself, the LED mounted on the right side of the board will be on.

Bit: 7 **Name:** Parity Error High Byte (PEHI)

Bit: 6 **Name:** Parity Error Low Byte (PELO)

Bit: 5 **Name:** Tag Parity Error (TPE)
Function: These bits are set individually when a parity error occurs in the high data byte, low data byte or tag field, respectively, if the cycle is aborted (Cache Parity Abort bit is set). If the cycle is not aborted, all three bits, 5, 6 and 7, are set upon any cache parity error occurrence as an aid to system software compatibility. Cleared by any write to the CMPE register or by console INIT.

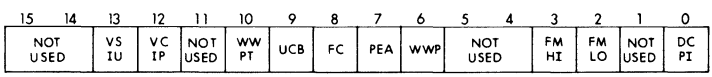


Figure 9-7 17 777 746 Cache Control Register (CCR)

Bit: 13 (Read-only)
Name: Valid Store in Use (VSIU)
Function: This bit indicates which set of valid store bits is currently being used to determine the validity of the contents of the tag store memory. It is complemented each time that the cache is flushed.

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

Bit: 12 (Read-only)
Name: Valid Clear in Progress (VCIP)
Function: This is set to indicate that the cache is currently in the process of clearing a valid store set. The clear cycle occurs on power-up and when the flush cache bit is set.

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

Bit: 10 (Read/write)
Name: Write Wrong Parity Tag (WWPT)
Function: This bit when set causes tag parity bits to be written with wrong parity on CPU read misses and write hits. A parity error will thus occur on the next access to that location.

Bit: 9 (Read/write)
Name: Unconditional Cache Bypass (UCB)
Function: When this bit is set all references to memory by the CPU will be forced to go to main memory. Read or write hits will result in invalidation of those locations in the cache and misses will not change the contents.

Bit: 8 (Write-only)
Name: Flush Cache (FC)
Function: This bit will always read as 0. Writing a 1 into it will cause the entire contents of the cache to be declared invalid. Writing a 0 into this bit will have no effect.

Bit: 7 (Read/write)
Name: Parity Error Abort (PEA)
Function: This bit controls the response of the cache to a parity error. When set, a cache parity error will cause a forced miss and an abort to occur (asserts UNIBUS signal PB L). When cleared, this bit inhibits the abort and enables an interrupt to parity error vector 114. All cache parity errors result in forced misses.

Bit: 6 (Read/write)
Name: Write Wrong Parity Data (WWPD)
Function: This bit when set causes high and low parity bytes to be written with wrong parity on all update cycles (CPU read misses and write hits). This will cause a cache parity error to occur on the next access to that location.

Bit: 3 (Read/write)
Name: Force Miss High (FMHI)
Function: This bit when set causes forced misses to occur on CPU reads of addresses where address bit 12 is a 1. This bit can also be set by moving the toggle switch S1 to the right side of the board. The bit cannot be cleared via the toggle switch.

Bit: 2 (Read/write)
Name: Force Miss Low (FMLO)
Function: This bit when set causes forced misses to occur on CPU

reads of addresses where address bit 12 is a 0. This bit can also be set by moving the toggle switch S2 to the right side of the board. The bit cannot be cleared via the toggle switch.

NOTE: Setting bits 3 and 2 will cause all CPU reads to be misses.

Bit: 0 (Read/write)
Name: Disable Cache Parity Interrupt (DCPI)
Function: This bit when set overrides the cleared condition of the Parity Error Abort bit, disabling the interrupt to location 114. The Cache Memory Parity Error bit must be cleared before Disable Cache Parity Interrupt (DCPI) is cleared.

Bit 7	Bit 0	Result of Cache Parity Error
0	0	Interrupt to 114 and force miss
0	1	Force miss only
1	X	Abort and force miss

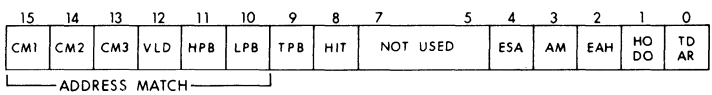


Figure 9-8 17 777 750 Cache Maintenance Register (CMR)

Bit: 15:10 (Write-only)
Name: Address Match Bits <21:16>
Function: This register is used to set bits 21:16 of the address match register, which provides a scope sync pulse to a user-accessible test point when the memory address lines (21:0) match the address match register (21:0). This feature is useful for troubleshooting the cache and PDP-11/44 system.

Bit: 15 **Name:** Compare 1 H

Bit: 14 **Name:** Compare 2 H

Bit: 13 **Name:** Compare 3 H

Bit: 12 **Name:** Valid H

Bit: 11 **Name:** High Parity bit H

Bit: 10 **Name:** Low Parity bit H

Bit: 9 **Name:** Tag Parity bit H

Bit: 8 **Name:** Hit L
Function:

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

Bit: 4 **Name:** Enable Stop Action
Function: This bit can be set to allow the cache to stop the CPU clock upon detection of a cache parity error or address match condition.

Bit: 3 (Read/write)
Name: Address Matched (AM)
Function: This bit is set when the 22-bit address match register is equal to the 22-bit cache address. The bit being set is indicated by the left LED mounted on top of the board.

Bit: 2 **Name:** Enable Halt Action
Function: This bit can be set to allow the cache to halt the CPU upon detection of a cache parity error or address match condition.

Bit: 1 (Read/write)
Name: Hit on Destination Only (HODO)
Function: When set, this bit causes the cache to be enabled only during the destination memory access of an instruction. Read hits and updates will only happen during the final destination access. This feature is a very powerful tool for cache diagnostics. When cleared, this bit has no effect on the cache. This bit should be used with caution, as it can cause stale data in the cache.

Bit: 0 (Read/write)
Name: Tag Data from Address Match Register (TDAR)
Function: When set, this bit enables the tag field of the cache to be written with data from bits 8:0 of the address match register. Once this bit is set, it will cause all cache writes to clear the valid bit in these locations. This feature allows the cache diagnostics to identify failures in the tag field of the cache array.

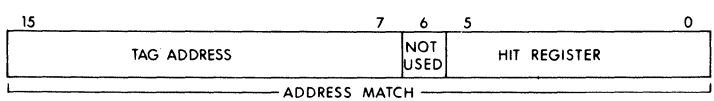


Figure 9-9 17 777 752 Cache Hit Register (CHR)

Bit: 15:0 (Write-only)
Name: Address Match Bits
Function: This register is used to set bits 15:0 of the address match register, which provides a scope sync pulse to a user-accessible test point when the memory address lines (21:0) match the address match register (21:0). It is used in conjunction with bits <15:10> of the Cache Maintenance Register. This feature is useful for troubleshooting the cache and PDP-11/44 system.

Bit: 15:7 (Read-only)

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

Bit: 5:0 (Read-only)
Name: Hit Register
Function: This six-bit field shows the number of cache hits (read and write hits) on the last six CPU accesses to non-I/O page memory. The bits flow from LSB to MSB of the field with a 1 indicating a hit and a 0 indicating a miss.

OTHER PDP-11/44 PROCESSOR EQUIPMENT

Floating Point Processor

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

Backplane

Figure 9-10, below, is an example of the PDP-11/44 CA Backplane. In this diagram, the standard and optional hardware features, described in the "Features" section of this chapter, are seen in their corresponding 14-Hex slots on the backplane.

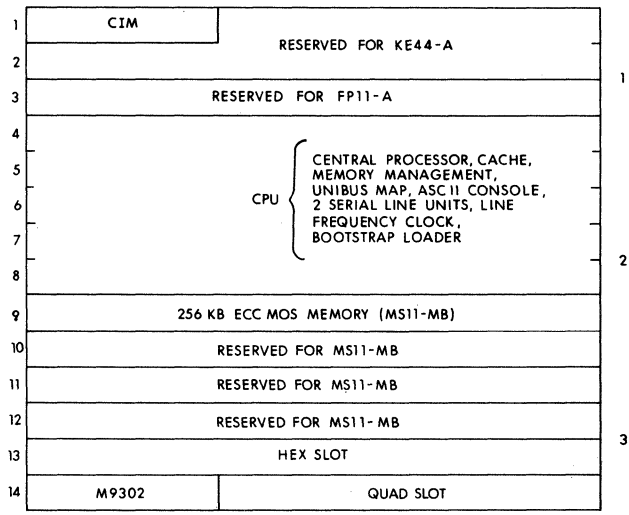


Figure 9-10 PDP-11/44 Backplane Configuration

ASCII CONSOLE

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

Physically, the I/O port used for the serial console function is shared with the standard system terminal (also called the "system console"), and is mode (or state) switchable by typing ASCII characters on the system terminal (the LA120 or equivalent which serves as the system console/programmer console).

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

NOTE

The console state can be entered only when the key switch is not in the local disable position.

Console State

The Console state is entered by typing a reserved input character, Control P (ASCII ↑P <020> or <220>). This is also called the Console Break character. The console state is also entered when the CPU halts. It can be entered only when the front panel key switch is in the local position. The reserved character is not passed to a running program, and console state is entered after printing the current output character, if any. While in the Console state, all input characters are interpreted by the console logic as commands to the CPU control interface. The console performs all character echoing while in the Console state.

A program running in the processor is inhibited by the console logic from sending or receiving any characters. This is accomplished by inhibiting the "ready" and "done" bits from being set. (See NOTE). The Console state is exited to the Program I/O state by typing a specific console command such as CONTINUE, START or BOOT. If there is no console command in execution, a front panel CONTINUE will cause exit from the Console state. Turning the front panel keyswitch to the local disable position will cause exit, and the beginning of a power-down sequence will also cause exit from the console state.

NOTE

When in console state, if a program just sends output to the printer without testing status bits, the characters will be printed if the logic happens to be ready.

Program I/O State

The Program I/O state is entered from the Console state by typing the CONTINUE command. A running program will then resume any input/output that might have been interrupted by the Console Break character. Any ASCII character may be output by the program, and any ASCII character, except the Console Break character, may be input to the program. Character echoing is the responsibility of the CPU software in Program I/O state.

The Program I/O state is exited to the Console state by typing the Console Break character, or by CPU execution of a HALT.

CONSOLE COMMAND SYNTAX AND SEMANTICS

< > Angle brackets are used to denote category names. For example, the category name <ADDRESS> may be used to represent any valid address, instead of actually listing

	all the strings of characters that can represent an address.
[ ]	Brackets surrounding part of an expression indicate the part of the expression which does not have to be typed, since it is optional.
<SP>	Represents one space.
<COUNT>	Represents a numeric count in octal.
<ADDRESS>	Represents an address argument in octal or a mnemonic in some cases. (See Address Mnemonics Section).
<DATA>	Represents a numeric argument in octal.
<QUALIFIER>	A command modifier (switch).
<INPUT-PROMPT>	Represents the console's input prompt string ">>>".
<CR>	Carriage return.
<LF>	Line feed.
<DEVICE-NAME>	Represents an alphabetic or optionally alphanumeric argument (in octal). Used with the Boot command.

Console defaults:

Address defaults:	A physical 22-bit address is always assumed in octal
Data defaults:	All transfers are 16-bit word transfers in octal

Control Characters and Special Characters

This section lists the control characters and special characters recognized by the console, and describes their functions. All control characters, with the exception of ↑P, are optional.

CONTROL C (↑C) Causes the suspension of all repetitive console operations such as:

1. Successive operations as a result of a /N qualifier.
2. Repeated command executions as a result of a REPEAT command.

CONTROL O (↑O) Suppresses/enables console terminal output (toggle). Console terminal output is always enabled at the next console input prompt.

CONTROL P (↑P) Enters Console state (if key switch is not in local disable position). Characters typed are now fielded by the console. If the console was already in Console mode another console <INPUT-PROMPT> is typed.

CONTROL U (↑U) When this is typed before a line terminator, it causes the deletion of all characters typed since the last line terminator. The console echoes: ↑U<CR><LF>

CONTROL S (↑S) Will stop execution of current command and character transmission until either a ↑Q or a ↑C is received. A system power failure will cause the ↑S action to be cleared and an exit from Console state.

CONTROL Q (↑Q) Will cause execution of current command and character transmission to continue. If no command or character transmission is in process, there is no response to this command. Transmission of characters, if any, continues.

NOTE

Typing the ↑S command on many intelligent terminals, including the LA120, will cause output to stop until a ↑Q command is typed or an ac line power interuption occurs on the terminal. No action at the front panel will change this condition, including turning CPU power off and on at the keyswitch. This is because the terminal responds to renewed system output by sending ↑S to stop it. Only typing the ↑Q command will inform the terminal that you desire output to continue.

CARRIAGE RETURN <CR> Terminates a console command line.

QUALIFIERS

The following is a list of allowable qualifiers and their descriptions:

/G Specifies general register addressing. This is a shorthand method to get to the general registers.

The user need only type an E or D (examine or deposit) followed by the /G qualifier, and then, instead of a full 22-bit address, simply enter the register number (e.g. 0, 1, 2, 3...).
Example: E/G<SP>7<CR> will examine R7 as compared to: E<SP>17777707<CR>

/N The /N qualifier is provided to permit examine and deposit commands to operate on multiple sequential addresses. The syntax of the /N qualifier is:
/N[:<COUNT>]. The [:<COUNT>] argument specifies the number of executions of the command to be performed. The default value for no [:<COUNT>] specified is two.

/M The /M qualifier allows the operator to examine various data and control paths in the PDP-11/44, and, in one special case, allows the operator to change (deposit to) the CPU's MPC (Micro Program Counter).

Example: E/M<SP>0<CR> will examine the data that are on the data bus internal to the floating point option. A list of machine-dependent addresses follows:

Address	=	Data Examined	Read or Write
0	=	Floating Point Data	R
1	=	CIS MPC	R
2	=	CIS Data	R
3	=	CPU Data	R
4	=	CPU MPC	R/W
5	=	Cache Data	R
6	=	CPU Error Register	R
7	=	MFM Data	R
10	=	UNIBUS Data	R
11	=	MFM Signal Register*	R

* The MFM Signal register has been added as an enhancement to subsequent M7096 modules (CS revision D or later).

/CB Cache bypass. Allows a user to force memory transfers even though cache is turned on and normally results in a cache hit.

/TB Take bus. A maintenance feature which allows the console to perform bus transfers even though the bus may be hung.

/E Extensive test. Used only with Test command. (See Test command description for more information.)

/A Extensive test followed by execution of continue command. Used only with Test command. (See Test command description for more information.)

CONSOLE COMMANDS

ADDER A<CR>

This command prints the 16-bit result of the current address pointer and the last data examined plus two. This command can be used to calculate the effective address for an instruction using mode 6, register 7 or mode 7, register 7.

Note: This command has been added as an enhancement to subsequent M7096 modules (CS revision D or later). CS revision D M7096 modules contain console microprocessor software which implements this command.

BOOT B[<SP><DEVICE-NAME>]<CR>

<DEVICE-NAME> is of the following format: DDn where DD is a two-letter device mnemonic (such as DT for DECtape), and n is the octal unit number. The unit number will default to zero if no number is typed.

If no <DEVICE-NAME> is given with the BOOT command, the console will perform the boot sequence for the default system device. This is the equivalent of using the front panel BOOT switch.

The BOOT command is executed only if the CPU is halted. Otherwise, an error message is generated. Console state is exited before boot execution is continued.

CONTINUE C<CR>

The CPU begins instruction execution at the address currently contained in the CPU program counter (PC) or continues execution if already running. CPU initialization is not performed. Addi-

tionally, the console enters Program I/O state (see Console State and Program I/O State sections) at the same time as issuing the CONTINUE to the CPU. This command may be used to return the console to Program I/O state even if the CPU was already running.

DEPOSIT

D[<QUALIFIER-LIST>]<SP><ADDRESS><SP><DATA><CR>

/M, /N, /G, /TB

Deposits <DATA> into the <ADDRESS> specified. The address space used will depend upon the qualifiers specified with the command (i.e., general registers if /G, or machine-dependent register if /M, or the default physical address, if no qualifiers are specified).

<ADDRESS> is a one- to eight-digit octal number (see note). Nonspecified upper bits are set to zero. Alternately, the address may be specified by one of the address mnemonics described below.

<DATA> is a one- to six-digit, 16-bit, octal number, and as with the address, nonspecified upper bits are set to zero.

The response to the DEPOSIT command is <CR><LF><INPUT-PROMPT> after execution of the command is completed. Deposits are legal only when the CPU is halted. Otherwise, an error message is generated.

NOTE

When the /M (machine-dependent register) qualifier is used, the value of <ADDRESS> can only be 4 (this is the only machine-dependent register which is writeable).

When the /G (general register) qualifier is used, the value of <ADDRESS> may not exceed 17 octal.

Address Mnemonics:

SW	Deposits to the Switch Register.
+	Deposits to the location immediately following the last location referenced.
–	Deposits to the location immediately preceding the last location referenced.
*	Deposits to the location last referenced.
@	Deposits to a physical address represented by the last data examined or deposited. Memory management is not used. Physical address bits 16-21 are set to zero.

EXAMINE

E[<QUALIFIER-LIST>]<SP><ADDRESS><CR>

/M, /N, /G, /CB, /TB

Examine the contents of the specified <ADDRESS>.

<ADDRESS> is a one- to eight-digit octal number (see note) with nonspecified upper address bits set to 0. Alternately, the address may be specified by one of the address mnemonics described below.

The response to the EXAMINE command is <CR><LF>ADDRESS><SP><DATA><CR><LF><INPUT-PROMPT>. The EXAMINE command is legal whether or not the CPU is running. The CPU is temporarily halted to perform the transfer if it is running.

NOTE

When the /M (machine-dependent register) qualifier is specified, the value of <ADDRESS> may not exceed 11 octal.

When the /G (general register) qualifier is specified, the value of <ADDRESS> may not exceed 17 octal.

Address Mnemonics:

SW	Examines the Switch Register.
+	Examines the location immediately following the last location referenced.
–	Examines the location immediately preceding the last location referenced.
*	Examines the location last referenced.
@	Examines the physical address represented by the last data examined or deposited. Memory management is not used. Physical address bits 16-21 are set to zero.

FILL

F[<SP><COUNT>]<CR>

Until a power failure has occurred, the console will send <COUNT> (in system radix) null characters after each <CR><LF> before any further transmission. A power failure will clear <COUNT>. Also, neither entering/exiting Console state nor execution of **any** other console command (including Test) affects <COUNT>. F<CR> sets fill to zero.

HALT

H<CR>

The CPU will stop instruction execution after completing the instruction in progress.

Upon halting the CPU, the console will display the physical address and contents of the PC.

INITIALIZE

I<CR>

A UNIBUS and processor intialize is executed for 150 ms. The response is <CR><LF><INPUT-PROMPT> after command execution is completed.

The INITIALIZE command is executed only if the CPU is halted. Otherwise, an error message is generated.

MICROSTEP

M[<SP><COUNT>]<CR>

The CPU is allowed to execute the number of microinstructions indicated by <COUNT>. If no <COUNT> is specified, one instruction is performed, and the console enters SPACE-BAR-

STEP mode. (See below.) The console enters Program I/O state immediately before issuing the step, and re-enters Console state as soon as the step completes. The macroinstruction may be completed by typing N<CR>.

The MICROSTEP command is executed only if the CPU is halted. Otherwise, an error message is generated.

NEXT

N[<SP><COUNT>]<CR>

The CPU is allowed to execute the number of MACRO instructions indicated by <COUNT>. If no <COUNT> is specified, one instruction is executed, and the console enters SPACE-BAR-STEP mode.

The console enters Program I/O state immediately before issuing the step, and re-enters Console state as soon as the step is completed.

SPACE-BAR-STEP FEATURE

Each time a NEXT or MICROSTEP command is given, the step(s) is/are executed and SPACE-BAR-STEP mode is entered. Each depression of the SPACE-BAR will cause a single step of the microcycle or instruction.

To exit SPACE-BAR-STEP mode, type any character except SPACE.

REPEAT

R<SP><CONSOLE COMMAND>

This causes the console to repeatedly execute the <CONSOLE COMMAND> specified, until execution is terminated by a Control-C (↑C). Any valid console command may be specified for <CONSOLE COMMAND> except the REPEAT, BINARY LOAD, FILL, TEST, and ADDER. The BOOT, HALT, CONTINUE and START commands are executed only once, since they result in an exit from the Console state.

START

S[<SP><ADDRESS>]<CR>

The START command performs the equivalent of the following sequence of console commands:

1. A system INITIALIZE is performed.
2. <ADDRESS> is deposited into the CPU Pro-

gram Counter (PC). If no address is specified, no address is loaded.

3. A CONTINUE is issued to begin CPU execution.

TEST T[/E or /A]<CR>

The console subsystem will execute a self-test, to check its own integrity. TEST may be executed while the CPU is running. Internal microprocessor program store data and RAM addressing/data are checked.

The /E qualifier results in extensive console testing, modifying main memory. TEST/EXTENSIVE may be executed only when the CPU is halted.

The /A qualifier is used optionally by diagnostic CZDLD <revision> to run the T/E command followed by a Continue command, automatically. The T/A command may be started with or without the CPU halted, but always continues the CPU after execution of the command is done.

BINARY LOAD X<SP><ADDRESS><SP><COUNT><CR><CHECKSUM>

This command instructs the console to prepare to load or unload <COUNT> binary data bytes starting from location <ADDRESS>. Only an even byte <COUNT> may be used.

A count with bit 15 set indicates that the data are to be sent to the requester (Binary Unload). The remaining bits in the count field are considered an unsigned, positive number indicating the number of bytes to load or unload.

All checksums used by this command are calculated by performing a 2's complement addition of each character into a register initially set to zero, with exceptions noted below. If no errors occurred, the low eight bits of the register should be zero after the checksum has been received and added into it.

Once a <CR> has been received, the console will stop echoing input bytes. A byte of binary data must follow the command after the requester has received the <LF> character from the console. This byte of data is a 2's complement byte checksum of the ASCII characters

which made up the command string (including the <CR>), and will not be loaded into memory. <COUNT> will not be decremented.

If the checksum is correct, the console will respond with the input prompt, but remain in binary mode (echo suppressed) and either send data to the requester or be prepared to receive data.

If the checksum calculation detects an error, the console will respond within one second with an error message, re-enable echo of received characters, issue its input prompt and await another command. This will prevent inadvertent operator entry into a mode where the console is accepting the next several thousand input characters as data with no escape sequence possible from the keyboard.

Binary Loading

A binary string of data of length $<COUNT>_1$ should be sent once the requester receives the input prompt, indicating that the console has accepted the command. The console will deposit all but the last byte (the checksum, which is not included in <count>) into the specified address space. As the console is receiving the data, it is also adding the bytes together to form another checksum. This sum starts at zero with M7096 modules CS rev. D and later. Earlier revisions start with the command string checksum value, instead of zero.

Once the <COUNT> is exhausted, the final byte transmitted to the console will be the block checksum of all the data. The console will compute the checksum as above, and respond within one second with an error message if an error is detected. In any case, the console will re-enable echo, issue an input prompt, and await the next command.

Binary Unload

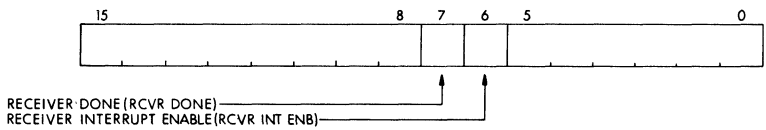
As in the load command, the console processes the command and checks the checksum. If the checksum is correct, the console responds with a normal input prompt, followed by a string of bytes which is the binary data requested. As each byte is sent, it is added to the checksum. This sum starts at zero with M7096 modules CS rev. D and later. Earlier revisions start with the command string checksum value, instead of zero. At the end of the transmission, the 2's complement of the checksum is sent. The console then re-enables echo, issues an input prompt, and awaits the next command.

If the original checksum fails, the console will respond with an error message. It will then issue an input prompt and await the next command. If the data checksum indicates an error, the device driving the console must take any action.

TERMINAL SERIAL LINE UNIT REGISTERS

All unused or write-only bits are zero when examined.

Receiver Control Status Register (TERM RCSR) 17 777 560



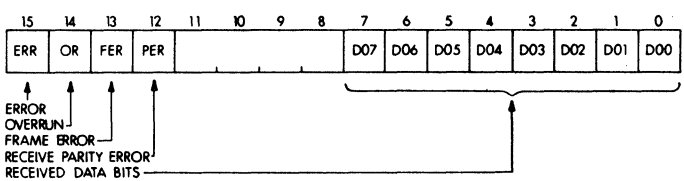
Bit: 15:8
Function: Unused

Bit: 7 (Read-only)
Name: RECEIVER DONE
Function: Set during the Program I/O state only when an entire character has been received and is ready for transfer to the CPU. Cleared by INIT or addressing (read-only) RBUF. Starts an interrupt sequence when set, if RECEIVER INTERRUPT ENABLE is also set.

Bit: 6 (Read/write)
Name: RECEIVER INTERRUPT ENABLE
Function: Cleared by INIT. When set, an interrupt sequence will start on BR4 each time RECEIVER DONE is set.

Bit: 5:0
Function: Unused

Receiver Data Buffer (TERM RBUF) 17 777 562



Bit: 15 (Read-only)
Name: ERROR
Function: Logical OR of OVERRUN ERROR, FRAMING ERROR and PARITY ERROR. ERROR is not tied to the interrupt logic, but RECEIVER DONE is.

Bit: 14 (Read-only)
Name: OVERRUN ERROR

Function: Set if previously received character is not read (RECEIVER DONE not cleared) before another character is received.

Bit: 13 (Read-only)
Name: FRAMING ERROR
Function: Set if the character received has no valid stop bit(s). Also used to detect a "break" character.

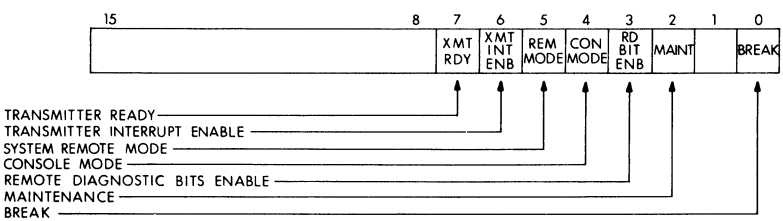
Bit: 12 (Read-only)
Name: PARITY ERROR
Function: Set if received parity does not agree with the expected parity. Always cleared if no parity is selected.

NOTE: Error bits remain set until the next character is received, at which time the error bits are updated. INIT does not clear the console terminal error bits. However, a power-up sequence does clear them. Error bits may be disabled via a jumper removal on the M7096 module.

Bit: 11:8
Function: Unused

Bit: 7:0 (Read-only)
Name: RECEIVED DATA
Function: These bits contain the character just received. If less than eight bits are selected, the buffer will be right-justified with the unused bits read as 0. Not cleared by INIT.

Transmitter Control Status Register (TERM XCSR) 17 777 564



Bit: 15:8
Function: Unused

Bit: 7 (Read-only)
Name: TRANSMITTER READY
Function: Set during the Program I/O state only by INIT or when XBUF can accept another character. Cleared when a character is written into the XBUF. Starts an interrupt sequence if TRANSMITTER INTERRUPT ENABLE is also set.

Bit: 6 (Read/write)
Name: TRANSMITTER INTERRUPT ENABLE
Function: Cleared by INIT. When set, an interrupt sequence will start on BR4 each time TRANSMITTER READY is set.

Bit: 5 (Read-only)
Name: SYSTEM REMOTE MODE
Function: Set when CPU is operating in the remote diagnostic mode.

Bit: 4 (Read-only)
Name: CONSOLE MODE
Function: Set to indicate that the CPU is operating in the Console state or mode.

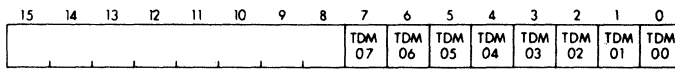
Bit: 3 (Read-only)
Name: REMOTE DIAGNOSTIC BITS ENABLE
Function: Set by turning on switch #2 of E79 on the M7096 module. When set, the status of bits 4 and 5 are entered into this register. When cleared (switch off), all three bits will be zero.

Bit: 2 (Read/write)
Name: MAINTENANCE
Function: Cleared by INIT. When set, it disables the serial line input to the RECEIVER and sends the serial output of the TRANSMITTER into the serial input of the RECEIVER. Forces receiver to run at transmitter speed.

Bit: 1
Function: Unused

Bit: 0 (Read/write)
Name: BREAK
Function: Cleared by INIT. When set, a continuous space is transmitted, equivalent to sending a null character with no stop bits (framing error). May be disabled with a jumper removal on the M7096 module.

Transmitter Data Buffer (TERM XBUF) 17 777 566

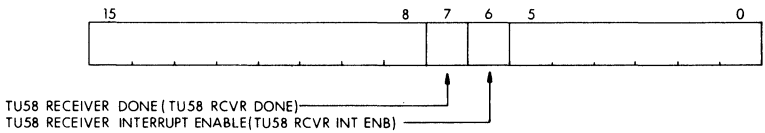


Bit: 15:8
Function: Unused

Bit: 7:0 (Write-only)
Name: TRANSMITTER DATA BUFFER
Function: If less than eight bits are jumper selected, the character must be right-justified.

TU58 SERIAL LINE UNIT REGISTERS

TU58 Receiver Control/Status Register (TU58 RCSR)



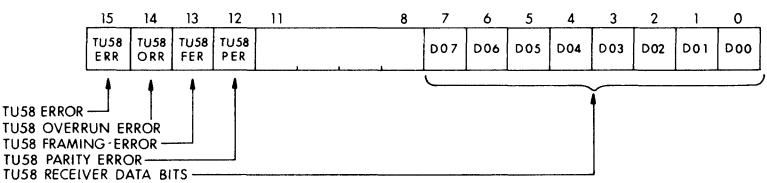
Bit: 15:8
Function: Unused

Bit: 7 (Read-only)
Name: TU58 RECEIVER DONE
Function: Set when an entire character has been received and is ready for transfer to the CPU. Cleared by INIT or addressing (read-only) RBUF. Starts an interrupt sequence when set if TU58 RECEIVER INTERRUPT ENABLE is also set.

Bit: 6 (Read/write)
Name: TU58 RECEIVER INTERRUPT ENABLE
Function: Cleared by INIT. When set, an interrupt sequence will start on BR4 each time TU58 RECEIVER DONE is set.

Bit: 5:0
Function: Unused

TU58 Receiver Data Buffer (TU58 RBUF)



Bit: 15 (Read-only)
Name: TU58 ERROR
Function: Logical OR of TU58 OVERRUN ERROR, TU58 FRAMING ERROR and TU58 PARITY ERROR. TU58 ERROR is not tied to the interrupt logic, but TU58 RECEIVER DONE is cleared by INIT. Bits 12 through 15 may be disabled and cleared via a jumper removal on the M7096 module.

Bit: 14 (Read-only)
Name: TU58 OVERRUN ERROR

Function: Set if previously received character is not read (TU58 RECEIVER DONE not cleared) before another character is received. Cleared by INIT or reading before receiving another character.

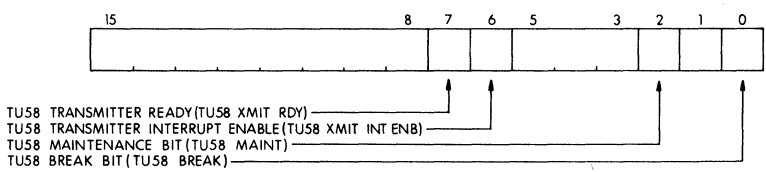
Bit: 13 (Read-only)
Name: TU58 FRAMING ERROR
Function: Set if character received has no valid stop bit(s). Cleared by INIT or when a valid character is received. This bit indicates an error in transmission or the reception of a "break" character.

Bit: 12 (Read-only)
Name: TU58 PARITY ERROR
Function: Set if received parity does not agree with expected parity. Cleared by INIT or when the parity of the next character is valid. Always cleared if no parity is selected.

Bit: 11:8
Function: Unused

Bit: 7:0 (Read-only)
Name: TU58 RECEIVED DATA
Function: These bits contain the character just received. If less than eight bits are selected, the buffer will be right-justified with the unused bits read as zero. Not cleared by INIT.

TU58 Transmitter Control/Status Register (TU58 XCSR)



Bit: 15:8
Function: Unused

Bit: 7 (Read-only)
Name: TU58 TRANSMITTER READY
Function: Set by INIT or when the TU58 XBUF can accept another character. Starts an interrupt sequence when set if TU58 TRANSMITTER INTERRUPT ENABLE is also set. Cleared when a character is written into the XBUF.

Bit: 6 (Read/Write)
Name: TU58 TRANSMITTER INTERRUPT ENABLE
Function: Cleared by INIT. When set, an interrupt sequence will start on BR4 each time TU58 TRANSMITTER READY is set. Cleared by the program or by the Initialize sequence.

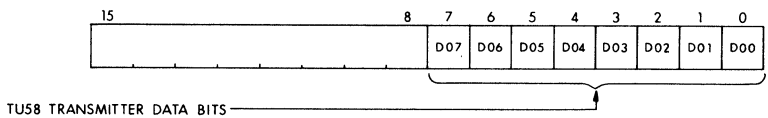
Bit: 5:3
Function: Unused.

Bit: 2 (Read/write)
Name: TU58 MAINTENANCE
Function: Cleared by INIT. When set, it disables the serial line input to the receiver and sends the serial output of the transmitter into the serial input of the receiver. Forces receiver to run at transmitter speed.

Bit: 1
Function: Unused

Bit: 0 (Read/write)
Name: TU58 BREAK
Function: Cleared by INIT. When set, a continous space is transmitted equivalent to sending a null character with no stop bits (framing error). May be disabled with a jumper removal on the M7096 module.

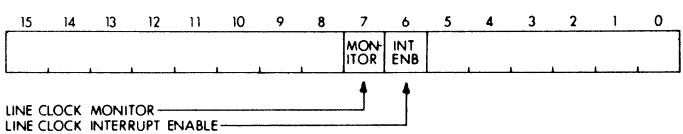
TU58 Transmitter Data Buffer (TU58 XBUF)



Bit: 15:8
Function: Unused

Bit: 7:0 (Write-only)
Name: TU58 TRANSMITTER DATA
Function: If less than eight bits are selected, the character must be right-justified.

Line Clock Status Register (LKS) 17 777 546



Bit: 15:8
Function: Unused

Bit: 7 (Read/write, clear only)
Name: LINE CLOCK MONITOR
Function: Set by INIT or by the line frequency clock signal, LTC. Cleared only by the program.

Bit: 6 (Read/write)
Name: LINE CLOCK INTERRUPT ENABLE
Function: Cleared by INIT. When set, starts an interrupt sequence each time LINE CLOCK MONITOR is set.

Bit: 5:0
Function: Unused

ADDRESS AND VECTOR ASSIGNMENTS

Integral to the PDP-11/44 CPU are the above two serial line units and a real-time clock. The serial line units and clock follow the same address and vector assignments as the KL11, DL11-A, B, C, D and W. SLU #1 is for the system console and has fixed addresses and vectors. SLU #2, normally used for the TU58, has switch-selectable contiguous addresses and vectors. The real-time clock has a fixed address and vector.

	Address	**Vector**	**Priority**
Console	17 777 560		BR4
(SLU #1)	17 777 562	60	(fixed)
	17 777 564		
	17 777 566	64	BR4 (fixed)
TU58	17 7YX XX0		BR4
(SLU #2)	17 7YX XX2	XX0	(fixed)
	17 7YX XX4		
	17 7YX XX6	XX4	BR4 (fixed)
	Where Y=6 or 7 and X=0-7 (Vector)		
Line Clock	17 777 546	100	BR6 (fixed)

NOTE

Recommended address and vector assignments for SLU #2 when used for a TU58 are:

Address: 17 776 500
Vector: 300
(These are the base values used to set switches.)

SERIAL LINE UNIT TIMING CONSIDERATIONS

Receiver

The RECEIVER DONE bit sets when the UART has assembled a full character, which occurs approximately at the middle of the first stop bit. Since the UART is double buffered, data remain valid until the next character is received and assembled. This allows one full character time for servicing the RECEIVER DONE bit or interrupt caused by it.

NOTE

The UART (Universal Asynchronous Receiver/ Transmitter) is an asynchronous subsystem. The transmitter accepts parallel characters and converts them to serial asynchronous output. The receiver accepts asynchronous serial characters and converts them to parallel output.

Transmitter

The UART's transmitter section is also double buffered. After initialization, the TRANSMITTER READY bit is set. When the buffer is loaded with the first character, the bit clears but sets again within a fraction of a character transmission time period. A second character can then be loaded, clearing the bit again. This time it remains clear until the first character and its stop bit(s) have been transmitted (about one character time).

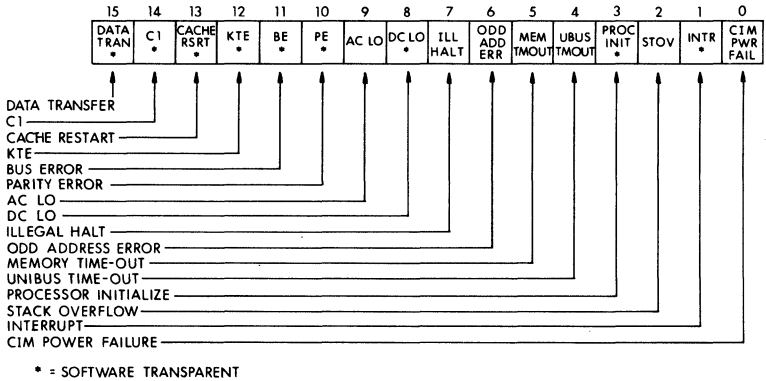
Break Generation

Setting the break bit causes the transmission of a continuous space. Since the TRANSMITTER READY bit continues to function normally, the duration of break can be timed by the "pseudo-transmission" of a number of characters. However, since the transmitter is double buffered, a null character (all zeros) should precede transmission of break to insure that the previous character completes transmission. Likewise, the last "pseudo-transmitted" character under break should be null.

REGISTERS

The following CPU registers are accessed by program or console control.

CPU Error Register 17 777 766



This register identifies the source of the abort or trap that used the vector at location 4. Bits 7:4, Bit 2 and Bit 0 are cleared when the CPU error register is written. When set, Bit 9 indicates to software that a software powerdown is in progress. The remaining bits are software transparent and are accessible only when the console has control. They serve to provide diagnostic visibility into the processor.

Bit: 15 **Name:** DATA TRANSFER
Function: Monitors the DATA TRAN line of the processor. When clear, this bit indicates the processor is initiating a data transfer on the UNIBUS.

Bit: 14 **Name:** C1
Function: Set when the control signal Bus C1 is asserted, indicating a DATO or DATOB transfer is being performed.

Bit: 13 **Name:** CACHE RESTART
Function: Set when the cache has generated the signal necessary to restart the processor clock.

Bit: 12 **Name:** KTE
Function: Set when a memory management error (nonresident, page length or read-only abort) has occurred.

Bit: 11 **Name:** BUS ERROR
Function: Set when processor has attempted to access nonexistent memory, odd address during word reference, or if there was no response on the UNIBUS within approximately 20 μs.

Bit: 10 **Name:** PARITY ERROR
Function: Set when processor has received a memory parity error.

Bit: 9 **Name:** AC LO
Function: Set when UNIBUS AC LO is asserted. To software, when

this bit is set, a powerdown is in progress. This signal is not latched and therefore Bit 9 is not affected by a processor INIT.

Bit: 8 **Name:** DC LO
Function: Set when UNIBUS DC LO is asserted. This signal is not latched and therefore Bit 8 is not affected by a processor INIT.

Bit: 7 **Name:** ILLEGAL HALT
Function: Set when a HALT instruction is attempted when the processor is in User or Supervisor mode.

Bit: 6 **Name:** ODD ADDRESS ERROR
Function: Set when the program attempts a word reference on an odd address.

Bit: 5 **Name:** MEMORY TIME-OUT
Function: Set when program attempts to read a word from a nonexistent memory location. This does not include UNIBUS addresses.

Bit: 4 **Name:** UNIBUS TIME-OUT
Function: Set when there is no response on the UNIBUS within approximately 20 μs.

Bit: 3 **Name:** PROCESSOR INITIALIZE
Function: Set when processor initialize signal is asserted.

Bit: 2 **Name:** STACK OVERFLOW
Function: Set when the Kernel hardware stack is less than 400 octal.

Bit: 1 **Name:** INTERRUPT
Function: Set when the PAX interrupt line is asserted.

Bit: 0 **Name:** CIM Power Failure
Function: Set when dc power to the machine has exceeded voltage tolerance limits for a period of 1.5 μs or greater.

Processor Status Word 17 77 776 (PSW)

The Processor Status Word contains information on the current status of the CPU. This information includes current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; an indicator for detecting the execution of an instruction to be trapped during program debugging;

and an indicator to determine whether a commercial instruction was in progress.

Processor Traps
These are a series of errors and programming conditions which will cause the central processor to trap to a set of fixed locations. These include Power Failure, Odd Addressing Errors, Stack Errors, Timeout Errors, Nonexistent Memory References, 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 90 volts for 120V power (180 volts for 240V) or outside a limit of 47 to 63 Hz, as measured by dc power, the powerfail sequence is initiated. The central processor automatically traps to location 24 and the user's powerfail Kernel program has 5 ms to save all volatile information (data in registers).

If battery backup is present, and the batteries are not depleted when power is restored, the processor traps to location 24 and executes the user's power-up routine to restore the machine to its state prior to power failure. If batteries are not present, a boot to default device is executed.

Odd Addressing Errors
This error occurs whenever a program attempts to execute a word instruction on an odd address between word boundaries. The instruction is aborted and the CPU traps through location 4.

Time-Out Error
This error occurs when a MSYN pulse is placed on the UNIBUS and there is no SSYN pulse within 20 μs. This error usually occurs in attempts to address nonexistent memory or peripherals.

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

Nonexistent Memory Errors
This error occurs when a program attempts to reference a nonexistent memory location. The cycle is aborted and the processor traps through vector 4.

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

Trap Handling

Appendix A includes a list of the reserved Trap Vector locations and System Error Definitions which cause processor traps. When a trap occurs, the processor follows the same procedure for traps as it does for interrupts (saving the PC and PS on the new Processor Stack, etc.).

In cases where traps and interrupts occur concurrently, the processor will service the conditions according to the priority sequence illustrated.

Trap Priorities

1. HALT (Instruction, Switch, or Command)
2. Memory Management Fault
3. Memory Parity Errors
4. Bus Error Traps
5. Floating Point Traps
6. TRAP Instruction
7. Trace Trap
8. Stack Overflow Trap
9. Power Fail Trap
10. Console Bus Request (Console Operation)
11. Program Interrupt Request (PIR) level 7
12. Bus Request (BR) level 7
13. PIR 6
14. BR 6
15. PIR 5
16. BR 5
17. PIR 4
18. BR 4
19. PIR 3
20. PIR 2
21. PIR 1
22. WAIT LOOP

Stack Limit Violations

When instructions cause a stack address to go lower than 400 octal, a Stack Violation occurs. The operation that caused the Stack Violation is completed, then a bus error trap is effected (Trap to 4). The error trap, which itself uses the stack, executes without causing an additional violation.

Program Interrupt Requests

A request is booked by setting one of Bits 15 through 9 (for PIR 7 - PIR 1) in the Program Interrupt Register at location 17 777 772. The hardware sets Bits 7:5 and 3:1 to the encoded value of the highest PIR bit set. This Program Interrupt Active (PIA) should be used to set the Processor Level and also index through a table of interrupt vectors for the seven software priority levels. Figure 9-11 below shows the layout of the PIR Register.

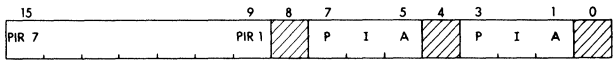


Figure 9-11 Program Interrupt Request Register

When the PIR is granted, the Processor will trap to location 240 and pick up the PC in 240 and the PSW in 242. It is the interrupt service routine's responsibility to queue requests within a priority level and to clear the PIR bit before the interrupt is dismissed.

The actual interrupt dispatch program should look like this:

```
MOVB PIR,PS          ;places Bits 7:5 in PSW
                     ;Priority Level Bits
MOV R5,-(SP)         ;save R5 on the stack
MOV PIR,R5
BIC #177761,R5       ;Gets Bits 3:1
JMP @DISPAT(R5)      ;use to index through table
                     ;which requires 15 core
                     ;locations
```

PDP-11/44 CPU and I/O Device Registers and Addresses

Address	Register
17 777 776	Processor Status Word (PSW)
17 777 772	Program Interrupt Request (PIRQ)
17 777 766	CPU Error
17 777 707 — 17 777 700	CPU General Registers
17 777 676 — 17 777 660	User Data PAR, Reg. 0-7
17 777 656 — 17 777 640	User Instruction PAR, Reg. 0-7
17 777 636 — 17 777 620	User Data PDR, Reg. 0-7
17 777 616 — 17 777 600	User Instruction PDR, Reg. 0-7

Address	Register
17 777 576	MM Status Register 2 (SR2)
17 777 574	MM Status Register 1 (SR1)
17 777 572	MM Status Register 0 (SR0)
17 777 570	Switch Register
17 777 566 — 17 777 560	Console Terminal SLU
17 777 776 — 17 760 000 (switch-selectable)	TU58 DECtape SLU (Normally 17 776 500)
17 777 516	MM Status Register 3 (SR3)
17 772 376 — 17 772 360	Kernel Data PAR, Reg. 0-7
17 772 356 — 17 772 340	Kernel Instruction PAR, Reg. 0-7
17 772 336 — 17 772 320	Kernel Data PDR, Reg. 0-7
17 772 316 — 17 772 300	Kernel Instruction PDR, Reg. 0-7
17 772 276 — 17 772 260	Supervisor Data PAR, Reg. 0-7
17 772 256 — 17 772 240	Supervisor Instruction PAR, Reg. 0-7
17 772 236 — 17 772 220	Supervisor Data PDR, Reg. 0-7
17 772 216 — 17 772 200	Supervisor Instruction PDR, Reg. 0-7
17 770 372 — 17 770 200	UNIBUS MAP Registers

SPECIFICATIONS

Packaging

A basic PDP-11/44 consists of a 10.5″ box with a 14-slot backplane, power supply, CPU, 256 Kbyte memory, and two cabinets.

There are prewired areas within the backplane for expansion with optional equipment.

Component Parts

The basic PDP-11/44 system includes:

- **Standard Equipment**

PDP-11/44 CPU
Memory Management
Bootstrap Loader
Line Frequency Clock

Serial Bus Interface for TU58
Terminal Interface
8 Kbyte Cache Memory
256 Kbyte ECC MOS Memory
BA11-A Box with Power Supply

- **Prewired Expansion Space for Optional Equipment**

Floating Point Processor
Commercial Instruction Set
2 SPC Slots for Peripherals, 1 Hex, 1 Quad
768 Kbyte ECC MOS Memory (up to 1,024 Kbytes maximum)
3 SU Open Space in CPU Box

INPUT POWER SPECIFICATIONS

AC Power

90-128 Vrms, 47-63 Hz, 1 phase power, 19 amps rms maximum @ 120 Vac
180-256 Vrms, 47-63 Hz, 1 phase power, 9.5 amps rms maximum @ 240 Vac

THE MOUNTING BOX

Size

Each cabinet is 26.4 cm high × 42.2 cm wide × 66.0 cm deep (10.4″ × 16.6″ × 26″)

Weight

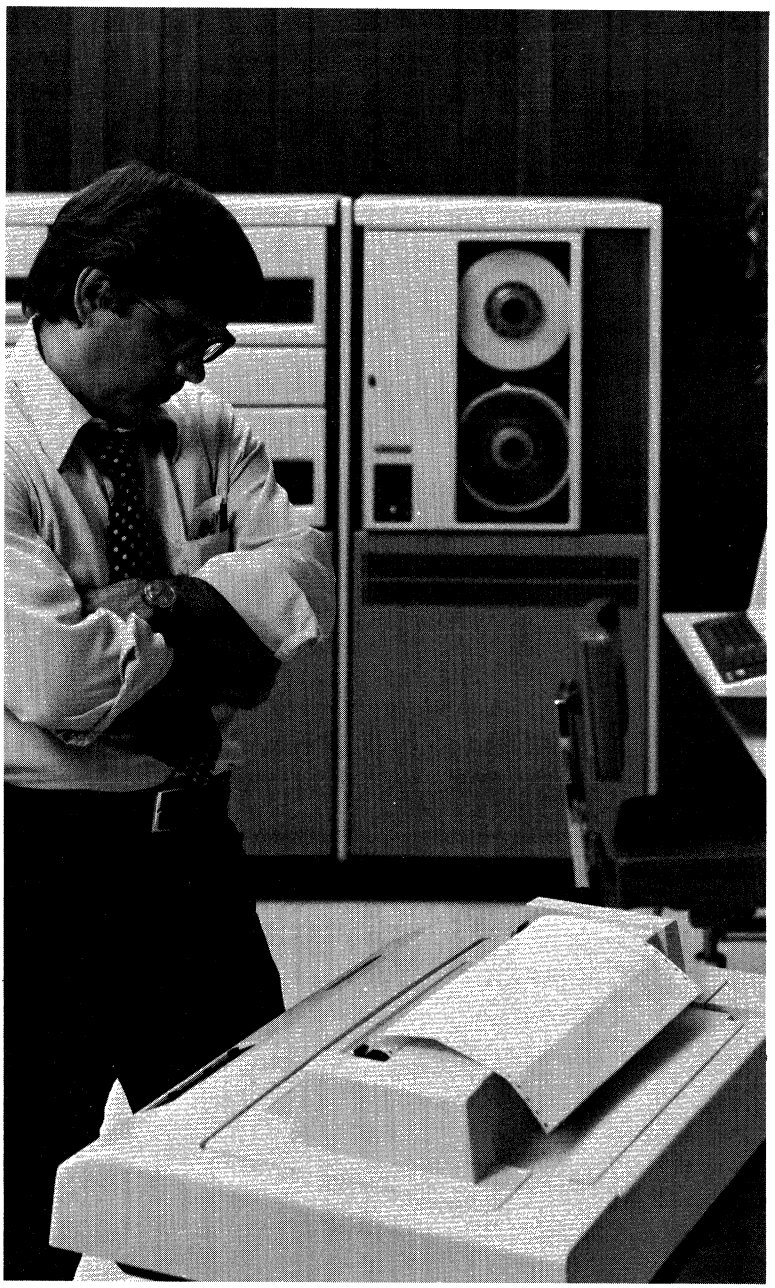
CPU Box:	40.5 kg (90 lbs.)

Operating Environment

Temperature:	5°C to 50°C (41°F to 122°F)
Humidity:	10% to 95% with max. wet bulb of 32°C (89.6°F) and minimum dew point of 2°C (36°F)
Altitude:	To 2.4 km (8000 ft.) noncondensing

Nonoperating Environment

Temperature:	−40°C to 80°C (−40°F to 176°F)
Humidity:	To 95% noncondensing
Altitude:	To 9.1 km (30,000 ft.)



CHAPTER 10
PDP-11/70

The PDP-11/70 is the most powerful computer in the PDP-11 family. It is designed to operate in large, sophisticated, high-performance systems, and can be used as a powerful computational tool for high-speed, real-time applications and for large multiuser, multitasking, timeshared applications requiring large amounts of addressable memory space. This systems-level PDP-11/70 uniquely applies the power of 32-bit internal architecture to demanding, multifunction computing requirements.

FEATURES

Integral to the PDP-11/70 central processor unit are these hardware features and expansion capabilities:

- Cache memory organization to provide very fast program execution speed and high system throughput
- Memory management for relocation and protection in multiuser, multitask environments
- Ability to access up to 3.9 million bytes of main memory (1 byte = 8 bits)
- Optional high-speed, mass storage controllers as an integral part of the CPU, to provide dedicated paths to high performance storage devices
- Optional Floating Point processor with advanced features, operating with 32-bit and 64-bit numbers

SYSTEM ARCHITECTURE

The PDP-11/70 is a medium scale general purpose computer using an enhanced, upwardly-compatible version of the basic PDP-11 architecture. A block diagram of the computer is shown in Figure 10-1.

The central processor performs all arithmetic and logical operations required in the system. Memory Management is standard with the basic computer, allowing expanded memory addressing, relocation, and protection. Also standard is a UNIBUS Map which translates UNIBUS addresses to physical memory addresses. The cache contains 2,048 bytes of fast, bipolar memory that buffers the data from main (core or MOS) memory.

Also within the CPU assembly are prewired areas for a floating point processor, and up to four high-speed I/O controllers.

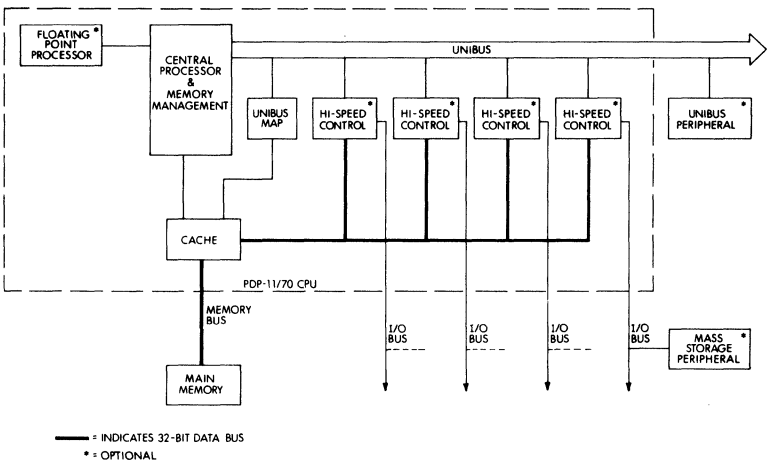


Figure 10-1 PDP-11/70 Block Diagram

The PDP-11/70 system has an expanded internal implementation of the PDP-11 architecture for greatly improved system throughput. The memory is on its own high data rate bus. The internal high-speed I/O controllers for mass storage devices have direct connections through the cache to memory for transferring data (using the cache only for timing purposes). The processor has a direct connection to the cache memory system for very high-speed memory access.

The UNIBUS remains the primary control path in the PDP-11/70 system. It is conceptually identical with other PDP-11 systems; the memory in the system still appears to be on the UNIBUS to all UNIBUS devices. Control and status information to and from the high speed I/O controllers is transferred over the UNIBUS. This expanded internal implementation of the PDP-11 architecture has no effect on PDP-11/70 programming.

RELIABILITY, AVAILABILITY, MAINTAINABILITY (RAMP) FEATURES

As the largest system computer of the PDP-11 family, the PDP-11/70 has extensive RAMP features and hardware. Reliability means minimizing failures. Availability and maintainability mean planning for ease of maintenance and spending minimum time isolating faults and making repairs.

In summary, the PDP-11/70 contains these RAMP features:

- Use of a self-diagnostic bootstrap module to test the viability of the instruction set and memory
- Extensive use of ECC memory and parity checking on addresses and internal data transfers
- Extensive use of error detection/correction by both hardware and software (cache, main memory, RP04s)
- Error logging to provide maximum diagnostic information to the user and Field Service
- Availability of user-mode diagnostics
- Availability of a wide range of subsystems
- Availability of function-level stand-alone diagnostics

CENTRAL PROCESSOR

The PDP-11/70 CPU performs all arithmetic and logical operations required in the system. It also acts as the arbitration unit for UNIBUS control by regulating bus requests and transferring control of the bus to the requesting device with the highest priority.

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

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

The central processor contains 16 general registers which can be used as accumulators, index registers, or as stack pointers. Stacks are extremely useful for nesting programs, creating re-entrant coding, and as temporary storage when a Last-In/First-Out structure is desirable. One of the general registers is used as the PDP-11/70's program counter. Three others are used as Processor Stack Pointers, one for each operational mode.

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

General Registers

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

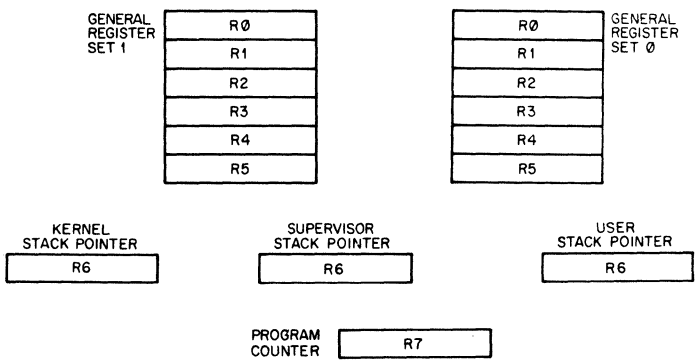


Figure 10-2 The General Registers

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

The R6 register is normally used as the Processor Stack Pointer (SP) indicating the last entry in the appropriate stack, a common temporary storage area with Last-In/First-Out characteristics. (For information on the programming use of stacks, please refer to Chapter 5.) The three stacks are called the Kernel Stack, the Supervisor Stack and the User Stack. When the central processor is operating in Kernel mode, it uses the Kernel Stack; in Supervisor mode, the Supervisor Stack; and in User mode, the User Stack. When an interrupt or trap occurs, the PDP-11/70 automatically saves its current status on the Processor Stack selected by the service routine. This stack-based architecture facilitates re-entrant programming.

The remaining 12 registers are divided into two sets of unrestricted registers, R0-R5. The current register set in operation is determined by the Processor Status Word.

The two sets of registers can be used to increase the speed of real-time data handling or facilitate multiprogramming. The six registers in General Register Set 0 could each be used as an accumulator and/or index register for a real-time task or device, or as general registers for a Kernel or Supervisor mode program. General Register Set 1 could be used by the remaining programs or User mode programs. The Supervisor can therefore protect its general registers and stacks from User programs, or other parts of the Supervisor.

Processor Status Word

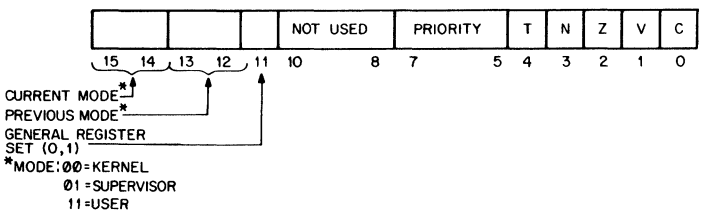


Figure 10-3 Processor Status Word

The Processor Status Word, at location 17 777 776, contains information on the current status of the PDP-11/70. This information includes the register set currently in use; current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

Modes — Mode information includes the present mode, either User, Supervisor or Kernel (bits 15, 14); the mode the machine was in prior to the last interrupt or trap (bits 13, 12); and which register set (General Register Set 0 or 1) is currently being used (bit 11).

The three modes permit a fully protected environment for a multiprogramming system by providing the user with three distinct sets of Processor Stacks and Memory Management Registers for memory mapping. In all modes except Kernel, a program is inhibited from executing a HALT instruction and the processor will trap through location 4 if an attempt is made to execute this instruction. Furthermore, in

other than Kernel mode, the processor will ignore the RESET and SPL (Set Priority Level) instructions. In Kernel mode, the processor will execute all instructions.

A program operating in Kernel mode can map users' programs anywhere in main memory and thus explicitly protect key areas (including the device registers and the Processor Status Word) from the User operating environment.

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

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

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

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

Stack Limit Register

All PDP-11s have a Stack Overflow Boundary at location 400_8. The Kernel Stack Boundary, in the PDP-11/70, is a variable boundary set through the Stack Limit Register found in location 17 777 774.

Once the Kernel stack exceeds its boundary, the processor will complete the current instruction and then trap to location 4 (Yellow, or Warning Stack Violation). If, for some reason, the program persists beyond the 16-word grace limit, the processor will abort the offending instruction, set the stack pointer (R6) to 4 and trap to location 4 (Red, or Fatal Stack Violation). A description of these traps is contained in Appendix A.

Error Correcting Code and Parity

ECC and Parity are used extensively in the PDP-11/70 to ensure the integrity of information. Parity for both data and addresses is generated on transfers to memory and is checked on all transfers from memory. Registers are provided within the CPU to provide information on the location of ECC errors, types of errors, and other relevant information so that software can respond to the situation, take corrective action, and log the occurrence of errors.

MEMORY SYSTEM

Address Space

The PDP-11/70 uses 22 bits for addressing physical memory. This represents a total of 2^{22} (over 4 million) byte locations.

Of the over 4 million byte locations possible with the 22-bit address, the top 256K are used to reference the UNIBUS rather than physical memory. Maximum main memory is therefore $2^{22} - 2^{18}$, or a total of 3,932,160 bytes.

Three separate address spaces are used with the PDP-11/70. Main memory uses 22-bit addresses, the UNIBUS uses an 18-bit address, and the computer program uses a 16-bit virtual address. The information is summarized below:

		Bytes
16 bits	program virtual space	2^{16} = 64K
18 bits	UNIBUS space	2^{18} = 256K
22 bits	physical memory space	4 million

Memory Management

The Memory Management hardware is standard on the PDP-11/70 computer. This hardware relocation and protection facility can convert the 16-bit program virtual addresses to 22-bit addresses. The unit may be enabled and disabled under program control. There is no increase in access time when the Memory Management unit is enabled.

UNIBUS Map

The UNIBUS Map responds as memory on the UNIBUS. It is the hardware relocation facility for converting the 18-bit UNIBUS addresses to

22-bit addresses. The relocation mapping may be enabled or disabled under program control.

Cache

The cache memory is a very high-speed memory that buffers data between the processor and main memory. The cache is completely transparent to all programs; programs are treated as if there were one continuous bank of memory.

Whenever a request is made to fetch data from memory, the cache circuitry checks to see if that data are already in the cache. If they are, then they are fetched from there and no main memory read is required. If the data are not already in cache memory, four bytes are fetched from main memory and stored in the cache, with the requested word or byte being passed directly to the CPU.

When a CPU request is made to write data into memory, it is written both to the cache and to main memory to insure that both stores are always updated immediately.

The key to the effectiveness of PDP-11/70's cache memory is its size. Because it holds 2,048 bytes at any given point in time, the PDP-11/70 cache already contains the next needed data a very high percentage of the time.

Detailed descriptions of cache memory and the other parts of memory appear later in this chapter.

Error Correcting Code (ECC)

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. The process of checking is accomplished by combining the bits in a number of unique ways, so that parity, or check bits, are generated for each unique combination and stored along with the data bits. The memory word length is extended to store these unique bits. When memory is read, the data word is again checked, and check bits are regenerated and compared with the check 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 syndrome bits are created and decoded to determine which data bit is in error. The bit in error is then corrected and sent on to the processor.

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

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

Battery Backup

Since MOS memory is volatile, meaning it depends on electricity to store information, a power loss or power shutdown would cause data loss. To prevent this loss from occurring, a Battery Backup Unit (BBU) has been designed to temporarily preserve the contents in memory. The Battery Backup unit is standard on the PDP-11/70.

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

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

Memory Options

To accommodate the need for MOS memory expansion or core memory replacement with MOS on the PDP-11/70, DIGITAL offers these three sets of option configurations:

- MOS memory expansion to MOS PDP-11/70s
- MOS memory expansion to core PDP-11/70s
- MOS memory to replace core in Core PDP-11/70s

MOS Memory Expansion to MOS — When expanding an MK11-B memory box with MK11-C memory arrays (newer memories), an ample number of slots must exist in the memory box, and all 16 slots may be utilized to realize a maximum capacity of 3.8 Mb. When expanding an MK11-C memory box with an MK11-C memory array, only 14 of the 16 slots may be filled with MK11-C memory, for a total of 3.5 Mb of interleaved memory. Mixing the MK11-B and MK11-C memory arrays in the same box results in the memory acquiring MK11-C characteristics. Memory array modules MK11-CE and MK11-CF are offered in pairs to maintain interleaving configurations. (See listing below for a brief description and capacity definition.) The Battery Backup Unit (BBU) is standard on all three PDP-11/70 memory configurations.

The H960 cabinet will hold two MK11-C MOS memory boxes. If more than two MOS memory boxes are required, an additional cabinet will be needed to accommodate space for the additonal memory box. The

cabinet option is MK11-CC/CD (see below). Available memory box options are the MK11-CA/CB and the MK11-CG/CH, listed below.

Option	Capacity	Brief Description
MK11-CA/CB	512 Kb	MOS/ECC w/Box
MK11-CC/CD	512 Kb	MK11-CA/CB w/Cabinet (H960)
MK11-CE	512 Kb	MOS Memory Expansion
MK11-CF	1 Mb	MOS Memory Expansion
MK11-CG/CH	1 Mb	MOS/ECC w/Box

MOS Memory Expansion to CORE — To provide MOS memory expansion to an existing CORE PDP-11/70 system, a MOS box option is required. This MOS memory box may be added to an existing cabinet that contains a maximum of one CORE memory box. MOS and CORE memory modules cannot be mixed within the same box. The option which contains the necessary hardware to expand a CORE CPU with MOS memory is the MK11-FA/FB. This option has 512 Kb capacity.

MOS Replacing CORE — The MK11-UA MOS replacing CORE kit contains the hardware that must be installed in the H960 memory cabinet of a CORE PDP-11/70, so that a MOS memory box can be mounted in that cabinet. Since no memory is included with this option, an MK11-C memory box option must be purchased in conjunction with the MK11-UA, in order to provide a total replacement option to the CORE memory box. (See below.) This kit services the H960 cabinet only.

Option	Capacity	Brief Description
MK11-CA/CB and MK11-UA	512 Kb	MOS/ECC w/Box and core to MOS Reconfiguration Kit
MK11-CG/CH and MK11-UA	1 Mb	MOS/ECC w/Box and core to MOS Reconfiguration Kit

OTHER CPU EQUIPMENT

Floating Point Processor

The PDP-11/70 Floating Point Processor fits integrally into the central processor. It provides a supplemental instruction set for performing single- and double-precision floating point arithmetic operations and

floating-integer conversion in parallel with the CPU. The floating point processor provides speed and accuracy in arithmetic computations. It provides 7 decimal digit accuracy in single-word calculations and 17 decimal digit accuracy in double-word calculations.

Floating point calculations take place in the FPP's six 64-bit accumulators. The 46 floating point instructions include hardware conversion from single- or double-precision floating point to single- or double-precision integers. There is a detailed description in Chapter 11.

High-Speed Mass Storage

The PDP-11/70 busing structure is optimized for high-speed device transfers. As many as four such devices can be plugged directly into the processor with a dedicated 32-bit bus feeding through to the main memory.

SYSTEM INTERACTION

High-speed Nonprocessor Request (NPR) devices use separate dedicated buses to the individual high-speed I/O controllers. From the controllers there is a single four-byte wide bus that interfaces to the cache. The order of priorities in the system is:

1. UNIBUS (via UNIBUS Map)
2. High-speed I/O controllers (A through D)
3. CPU

Control information and lower speed data transfers are carried out through the UNIBUS.

A device will request the UNIBUS for one of two purposes:

1. To make an NPR transfer of data (direct data transfers such as DMA)
2. To interrupt program execution and force the processor to branch to a service routine

There are two sources of interrupts, hardware and software.

Hardware Interrupt Requests

A hardware interrupt occurs when a device wishes to indicate to the program, or the central processor, that a condition has occurred (such as transfer completed, end of tape, etc.). The interrupt can occur on any one of the four Bus Request levels and the processor responds to the interrupt through a service routine.

Program Interrupt Requests

Hardware interrupt servicing is often a two-level process. The first level is directly associated with the device's hardware interrupt and

consists of retrieving the data. The second is a software task that manipulates the raw information. The second process can be run at a lower priority than the first, because the PDP-11/70 provides the user with the means of scheduling his software servicing through seven levels of Program Interrupt Requests. The Program Interrupt Request Register is located at address 17 777 772. An interrupt is generated by the programmer setting a bit on the high-order byte of this register.

SPECIFICATIONS

PACKAGING

A basic PDP-11/70 consists of two H960 cabinets (see Figure 10-4), or a double-width corporate cabinet (see Figure 10-5).

H960 Cabinet

1. A CPU cabinet which contains the processor, CPU-related equipment and interface equipment.
2. A Memory Cabinet which contains the first 512K bytes of ECC MOS memory, with expansion capability to 3,932,100 bytes within the memory box. Another H960 memory cabinet located next to it can house an additional 2,048K bytes of core or MOS memory.

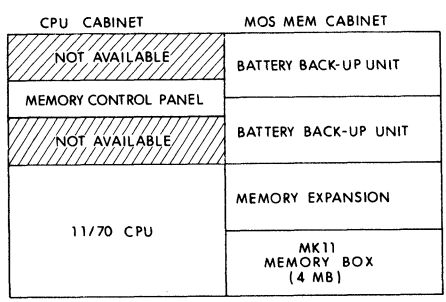


Figure 10-4 PDP-11/70 Equipment in H960 Cabinets

Corporate Cabinet*

1. A CPU cabinet which contains the processor, CPU-related equipment, interface equipment, and the first 512K bytes of ECC MOS memory (with expansion capability to 3,932,100 bytes within the memory box).
2. Another memory corporate cabinet located next to it can house memory for interleaving between memory boxes.

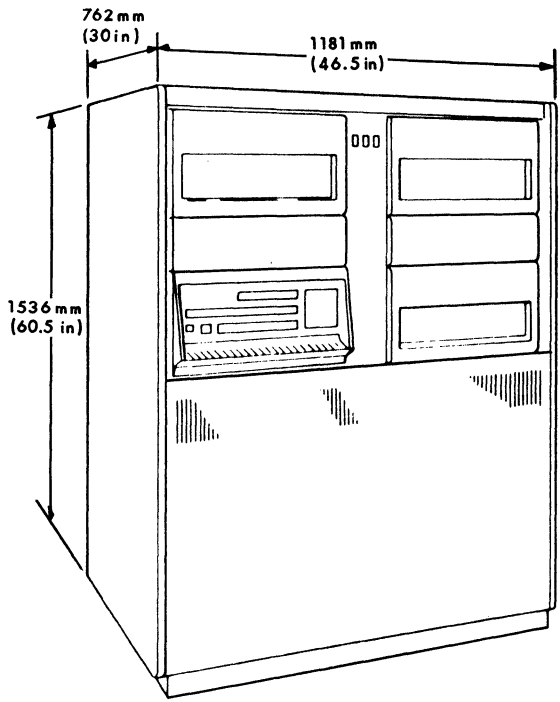


Figure 10-5 PDP-11/70 Equipment in Corporate Cabinet

An LA120 DECwriter III console terminal is included with the PDP-11/70 system. There are prewired areas within the mounting assemblies for expansion with optional equipment.

* NOTE: By using the 256K byte memory arrays, the entire PDP-11/70 main memory is contained in a single BA11-K box.

COMPONENT PARTS

The basic PDP-11/70 system has:

Standard Equipment

- PDP-11/70 CPU
- Memory Management
- Bootstrap loader
- DECwriter (LA120)
- Terminal interface (DL11-W) with integral line clock

- 2K byte cache memory
- 512K byte MOS ECC
- CPU cabinet
- Memory cabinet

Prewired Expansion Space for Optional Equipment

- Floating Point Processor
- Four high-speed I/O controllers
- Four SPC slots for peripherals
- 128 Kbyte parity core or MOS (within 1st memory expansion frame)

OTHER SPECIFICATIONS

AC Power

120/208 Vac ±10%, 47 to 63 Hz, 3 phase power
240/400 Vac ±10%, 47 to 63 Hz, 3 phase power

	120 Vac	240 Vac
Basic CPU cabinet (maximum current on each of 2 phases)	15A	7.5A
Memory, each BA11-K Box (maximum current on 1 phase)	12A	6A

Size

Each H960 cabinet is 1829 mm high × 533 mm wide × 762 mm deep (72 in. × 21 in. × 30 in.)
Each double-width corporate cabinet is 1536 mm × 1181 mm wide 762 mm deep (60.5 in. × 46.5 in. × 30 in.)

Weight (H960 cabinet)

CPU cabinet:	227 kg (500 lbs.)
Memory cabinet:	114 kg (including 1st 512 Kbytes) (500 lbs.)
Memory expansion frame:	67.5 kg (each additional 512 Kbytes) (150 lbs.)

Operating Environment

Temperature:	15C° to 32C° (59F° to 90F°)
Humidity:	20% to 80% with max. wet bulb 28°C (82°F) and minimum dew point 2°C (36°F)
Altitude:	to 2.4 km (8000 ft.)

Nonoperating Environment

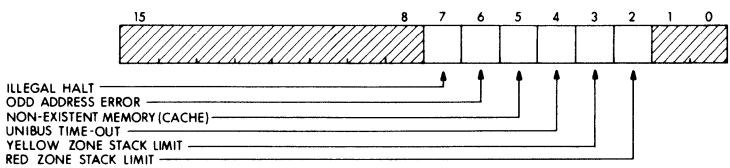
Temperature:	−40°C to 66°C (−40°F to 151°F)
Humidity:	0 to 95%
Altitude:	to 9.1 km (30,000 ft.)

PROCESSOR CONTROL

REGISTERS

The following five CPU registers are not accessible from the UNIBUS. They are accessed by program or console control.

CPU Error Register 17 777 766



This register identifies the source of the abort or trap that used the vector at location 4.

Bit: 7 **Name:** Illegal HALT
Function: Set when trying to execute a HALT instruction when the CPU is in User or Supervisor mode (not Kernel).

Bit: 6 **Name:** Odd Address Error
Function: Set when a program attempts to do a word reference to an odd address.

Bit: 5 **Name:** Nonexistent Memory
Function: Set when the CPU attempts to read a word from a location higher than indicated by the System Size register. This does not include UNIBUS addresses.

Bit: 4 **Name:** UNIBUS Timeout
Function: Set when there is no response on the UNIBUS within approximately 10 μsec.

Bit: 3 **Name:** Yellow Zone Stack Limit
Function: Set when a yellow zone trap occurs.

Bit: 2 **Name:** Red Zone Stack Limit
Function: Set when a red zone trap occurs.

Lower Size Register 17 177 760
This read-only register specifies the memory size of the system. It is defined to indicate the last addressable block of 32 words in memory (bit 0 is equivalent to bit 6 of the Physical Address).

Upper Size Register 17 777 762
This register is an extension of the system size, which is reserved for future use. It is read-only and its contents are always read as zero.

System I/D Register 17 777 764
This read-only register contains information uniquely identifying each system.

Microprogram Break Register 17 77 770
This register is used for maintenance purposes only. It is used with maintenance equipment to provide synchronization and testing facilities.

Processor Status Word 17 777 776
The Processor Status Word contains information on the current status of the CPU. This information includes the register set currently in use; current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

PROCESSOR TRAPS
There are a series of errors and programming conditions which will cause the central processor to trap to a set of fixed locations. These include Power Failure, Odd Addressing Errors, Stack Errors, Timeout Errors, Nonexistent Memory References, Memory Parity Errors, Memory Management Violations, Floating Point Processor Exception Traps, use of Reserved Instructions, use of the T bit in the Processor Status Word, and use of the IOT, EMT, and TRAP instructions.

Power Failure
Whenever ac power drops below 95 volts for 110V power (190 volts for 240V) or outside a limit of 47 to 63 Hz, as measured by dc power, the power fail sequence is initiated. The central processor automatically traps to location 24 and the power fail program has 2 msec. to save all volatile information (data in registers), and to condition peripherals for power failure.

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

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

Time-Out Error
This error occurs when a Master Synchronization pulse is placed on the UNIBUS and there is no slave pulse within 10 μsec. This error usually occurs in attempts to address nonexistent memory or peripherals.

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

Nonexistent Memory Errors
This error occurs when a program attempts to reference a memory address that is larger than that indicated by the system size register. The cycle is aborted and the processor traps through location 4.

Reserved Instruction
There is a set of illegal and reserved instructions which causes the processor to trap through location 10. The set is fully described in the Programming Techniques Chapter (Chapter 5).

Trap Handling
Chapter 5 includes a list of the reserved Trap Vector locations, and System Error Definitions which cause processor traps. When a trap occurs, the processor follows the same procedure for traps as it does for interrupts (saving the PC and PS on the new Processor Stack, etc.).

In cases where traps and interrupts occur concurrently, the processor will service the conditions according to the priority sequence illustrated.

Trap Priorities
- Parity error
- Memory Management violation
- Stack Limit Yellow
- Power Failure (power down)
- Floating Point exception trap
- Program Interrupt Request (PIR) level 7
- Bus Request (BR) level 7
- PIR 6
- BR 6
- PIR 5

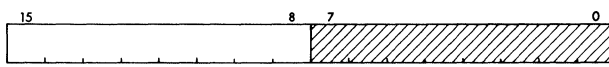
- BR 5
- PIR 4
- BR 4
- PIR 3
- PIR 2
- PIR 1
- Trace trap

STACK LIMIT

The Stack Limit allows program control of the lower limit for permissible stack addresses. This limit may be varied in increments of 400_8 bytes (200_8 words), up to a maximum address of 177 400 (almost the top of a 64 Kb memory).

The normal boundary for stack addresses is 400. The Stack Limit option allows this lower limit to be raised, providing more address space for interrupt vectors or other data that should not be destroyed by the program.

There is a Stack Limit Register, with the following format:



The Stack Limit Register can be addressed as a word at location 17 777 774, or as a byte at location 17 777 775. The register is accessible to the processor and console, but not to any bus device.

The eight bits, 15 through 8, contain the stack limit information. These bits are cleared by System Reset, Console Start, or the RESET instruction. The lower eight bits are not used. Bit 8 corresponds to a value of 400_8 or 256_{10}.

Stack Limit Violations

When instructions cause a stack address to exceed (go lower than) a limit set by the programmable Stack Limit Register, a Stack Violation occurs. There is a Yellow Zone (grace area) of 32 bytes below the Stack Limit which provides a warning to the program so that corrective steps can be taken. Operations that cause a Yellow Zone Violation are completed, then a bus error trap is effected. The error trap, which itself uses the stack, executes without causing an additional violation, unless the stack has entered the Red Zone.

A Red Zone Violation is a Fatal Stack Error. (Odd Stack or Nonexistent Stack are the other Fatal Stack Errors.) When detected, the operation

causing the error is aborted, the stack is repositioned to address 4, and a bus error occurs. The old PC and PS are pushed into locations 0 and 2, and the new PC and PS are taken from locations 4 and 6.

Stack Limit Addresses

The contents of the Stack Limit Register (SL) are compared to the stack address to determine if a violation has occurred. The least significant bit of the register (bit 8) has a value of 400_8. The determination of the violation zones is as follows:

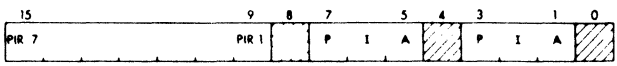
Yellow Zone = (SL) + $(340—377)_8$ execute, then trap
Red Zone ≤ (SL) + $(337)_8$ abort, then trap to location 4

If the Stack Limit Register contents were zero:

Yellow Zone = 340 through 377
Red Zone = 000 through 337

PROGRAM INTERRUPT REQUESTS

A request is booked by setting one of the bits 15 through 9 (for PIR 7—PIR 1) in the Program Interrupt Register at location 17 777 772. The hardware sets bits 7-5 and 3-1 to the encoded value of the highest PIR bit set. This Program Interrupt Active (PIA) should be used to set the Processor Level and also index through a table of interrupt vectors for the seven software priority levels. The figure below shows the layout of the PIR Register.



Program Interrupt Request Register

When the PIR is granted, the Processor will trap to location 240 and pick up the PC in 240 and the PSW in 242. It is the interrupt service routine's responsibility to queue requests within a priority level and to clear the PIR bit before the interrupt is dismissed.

The actual interrupt dispatch program should look like:

```
MOVB PIR, PS          ;places Bits 5-7 in PSW Priority
                      ;Level Bits

MOV R5,-(SP)          ;save R5 on the stack
```

```
MOV PIR,R5
BIC #177761,R5          ;Gets Bits 1-3
JMP @DISPAT(R5)         ;use to index through table
                        ; which requires 15 core locations.
```

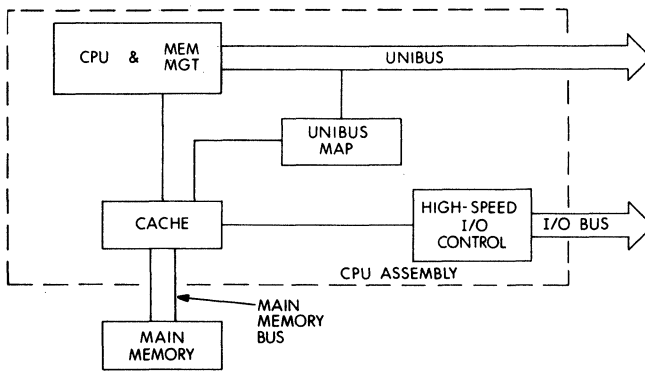


Figure 10-6 Block Diagram of PDP-11/70

CACHE MEMORY

An overall block diagram of the PDP-11/70 is shown in Figure 10-6. From a functional standpoint, main memory and the cache can be treated as a single unit of memory.

The PDP-11/70 Cache

The architecture of the cache chosen for the PDP-11/70 is described in this section. It represents a carefully developed approach, backed by extensive program simulations to determine hit statistics. The size of the cache memory is 1,024 words (2,048 bytes), organized as a two-way set associative cache with two-word blocks. There are two groups in the cache; each group contains 256 blocks of data, and each block contains two PDP-11 words (see Figures 10-7 and 10-8). Each block also has a tag field, which contains information to construct the address in main memory where the original copy of this data block resides. The data from main memory can be stored within the cache in one index position determined by its physical address. Refer to Figure 10-9 for the organization of the 22-bit physical address. The 8-bit index field (bits 2 to 9) determines which element of the array will contain the data (it can be in either Group 0 or Group 1).

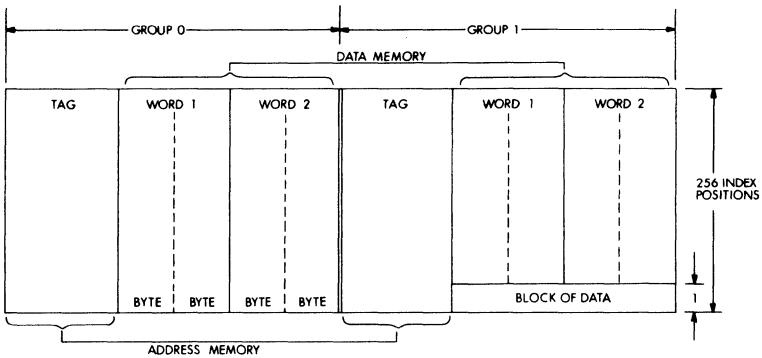


Figure 10-7 Cache Memory (2,048 Bytes)

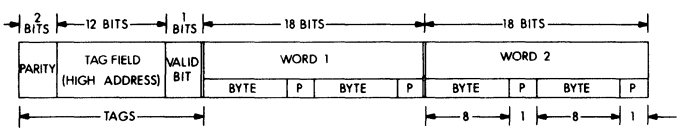


Figure 10-8 Block of Data plus Tags

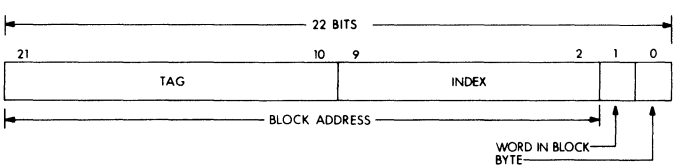


Figure 10-9 Physical Address

The elements of the cache must store not only the data, but also the address identification. Since the index position itself implies part of the address, only the high address field (called tag field) must be stored. The combination of the tag plus index gives the address of the

two-word block in main memory. The lowest two bits in the physical address select the particular word in the block, and the byte (if needed).

There are two places in the cache where any block of data can go; a particular index position in either Group 0 to Group 1. Random selection determines into which group the information is placed, overwriting the previous data. Another bit is needed within the cache to determine if the block has been loaded with data. When power is first applied, the cache data are invalid, and the valid bit for each data block is cleared. When a particular block location is updated, the associated valid bit is set to indicate good data.

Figure 10-8 shows the organization for a single block of data within a set. Note that data have byte parity, and that the nondata part, called "tags," contains a 12-bit high order address field, plus a valid bit and two parity bits.

General Operation

The system always looks for data in the fast cache memory first. If it is there (a hit), execution proceeds at the fastest rate. If the information is not there (a miss), and the operation was a read, a two-word block of data is transferred from main memory to the cache. If there is a miss while trying to write, cache is not updated. Main memory and the cache are both updated on write hits.

The operation of hits or misses is summarized in Table 10-1.

Table 10-1 Operation on Hit or Miss

	What Happens In	
	CACHE	MAIN MEMORY
READ		
hit	no change	no change
miss	updated	no change
WRITE		
hit	updated	updated
miss	no change	updated

When power is first applied (power-up), all of the valid bits are cleared. If power is suddenly lost, cache data may become invalid, but main memory, with nonvolatile core or battery-backed-up MOS, will have a correct copy of all the data.

With a typical program, writes occur only 10% of the time. Reads occur 90% of the time. Read hits will average 80% to 95% of all cycles with a typical program.

PARITY

System Reliability

Parity is used extensively in the main memory of the PDP-11/70 to ensure the integrity of data storage and transfer, and to enhance the reliability of system operation. All of memory (cache and main memory) has byte parity. Parity is generated and checked on all transfers between core or MOS and cache, again between cache and the CPU, between high-speed mass storage devices and their controllers, and again between the controllers and main memory. A software routine can be used to log the occurrence of parity errors, to handle recovery from errors, and to provide information on system reliability and performance.

Parity in the System

Main memory stores one parity bit for each 8-bit byte in core, or an equivalent function in check bits for ECC MOS memory. Refer to Figure 10-10. The cache also stores byte parity for data, and it stores two parity bits for the address and control information (tag storage) associated with each 2-word block of data.

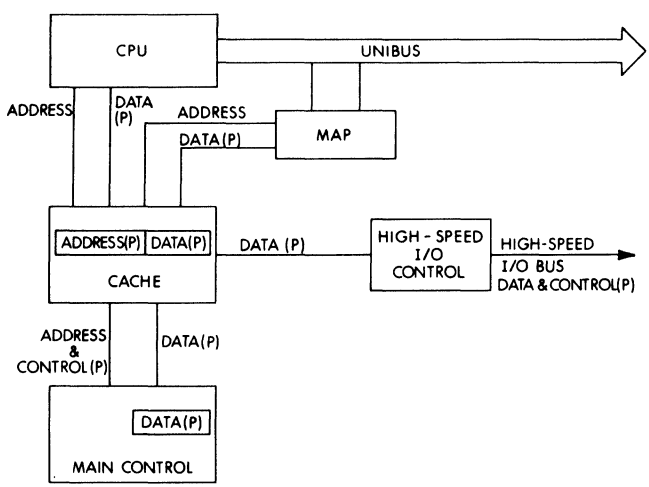


Figure 10-10 Parity (P) in the PDP-11/70 System

The bus between main memory and the cache contains parity on the data and address and control lines. The high-speed I/O controllers check and generate parity for data transfers to main memory, and they have the capability of handling address errors that are flagged by the control in the cache memory.

System Handling of Parity Errors

Extensive capabilities have been designed into the PDP-11/70 to allow recovery from parity errors, and to allow operation in a degraded mode if a section of the memory system is not operating properly. This type of operation is possible under program control by using the built-in control registers.

If part or all of the cache memory is malfunctioning, it is possible to bypass half or all of the cache. Misses can be forced within the cache, such that all read data are brought from main memory. Operation will be slower, but the system will yield correct results. If part of main memory is not working, the Memory Management unit can be used to map around it. If data found in the cache do not have correct parity, the memory system automatically tries the copy in main memory, to allow program execution to proceed.

Details of how to perform this programming are explained in the next section on the CPU and memory control registers.

Aborts and Traps

Two actions can take place after detection of a parity error. The cycle can be aborted. Then the computer transfers control through the vector at location 114 to an error handling routine. The other action is that the instruction is completed, but then the computer traps (also through location 114). In the first case it was not possible to complete the cycle, whereas in the second case, it was. This second type of parity error usually (but not always) causes the trap before the next instruction is fetched. Refer to Table 10-2.

Table 10-2 Response to Parity Errors

Parity Error Detected	Condition for Abort	Condition for Trap
CPU cycle, data error, read from main memory	Error in requested word	Error in the other word
UNIBUS cycle,* data error, read from main memory		Error in either word

CPU cycle, address error, reference to main memory	All reads and writes	
UNIBUS cycle address error reference to main memory		All reads and writes
CPU or UNIBUS cycle, data or address error, reference to cache		All reads
High-speed I/O cycle, data or address error, ref to main memory	(no CPU aborts or traps occur; high-speed I/O controllers handle their parity errors)	

* When a parity error is detected on data going to the UNIBUS, the parity error signal is asserted.

System Response to Parity Errors

Data are read from main memory to the cache in two-word blocks. If the read cycle was caused by the CPU, and a parity error is detected in the requested word, an abort occurs. If it was in the other word, a trap occurs. On UNIBUS cycles, a trap is caused if there is a read error in either word.

When an address parity error is detected on any read or write to main memory, an abort is caused for both CPU and UNIBUS cycles.

When any fast data memory or address memory parity error is detected on any read from the cache, a trap occurs. On a fast data memory parity error, the CPU will try to get the data from main memory, and also overwrite the same cache location with the new (correct) word just fetched. On an address memory parity error, the CPU will go to main memory for the data, and will correct (overwrite) the tag storage in the cache.

Data transfers for the high-speed mass storage devices take place with main memory. No data are stored in the cache. Parity errors are handled by the device controllers; no CPU aborts or traps occur, and no cache status registers are affected.

Table 10-2 summarizes the system response.

CACHE REGISTERS

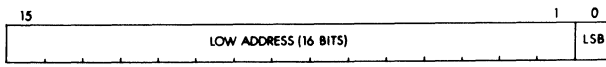
The registers described in this section provide information about pari-

ty errors, memory status and CPU status. These hardware registers have program addresses in the top 4K words of physical address space (Peripheral Page).

Register	**Address**
Low Error Address	17 777 740
High Error Address	17 777 742
Memory System Error	17 777 744
Control	17 777 746
Maintenance	17 777 750
Hit/Miss	17 777 752

Some bit positions of the registers are not used (not implemented with hardware) and are indicated by cross-hatching. These bits are always read as zeros by the program. Most of the bits can be read or written under program control. The above six registers are located on the cache control board of the 11/70.

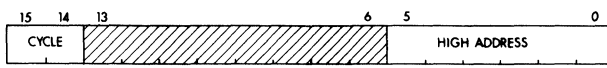
Low Error Address Register 17 777 740



This register contains the lowest 16 bits of the 22-bit address of the first error. The least significant bit is bit 0. The high order bits are contained in the High Error Address Register.

All the bits are read-only. The bits are undetermined after a Power-Up. They are not affected by a Console Start or RESET instruction.

High Error Address Register 17 777 742



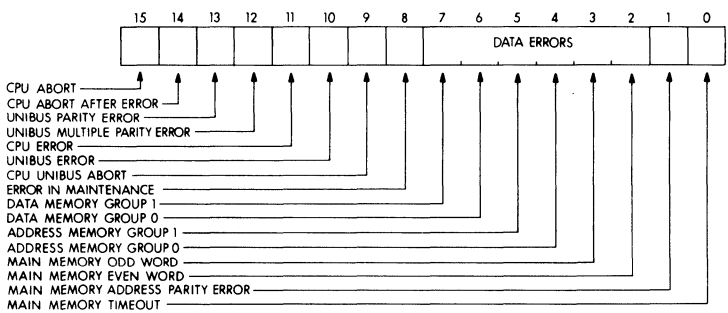
Bit: 15:14 **Name:** Cycle Type
Function: These bits are used to encode the type of memory cycle which was being requested when the parity error occurred.

Bit 15	Bit 14	Cycle Type
0	0	Data In (read)
0	1	Data In Pause
1	0	Data Out
1	1	Data Out Byte

Bit: 5:0 **Name:** Address
Function: These bits contain the highest six bits of the 22-bit address of the first error. Register Bit 5 corresponds to the physical address Bit 22.

All the bits are read-only. The bits are undetermined after a power-up. They are not affected by a Console Start or RESET instruction.

Memory System Error Register 17 77 744



Bit: 15 **Name:** CPU Abort
Function: Set if an error occurs which caused the cache to abort a processor cycle.

Bit: 14 **Name:** CPU Abort After Error
Function: Set if an abort occurs with the Error Address Register locked by a previous error.

Bit: 13 **Name:** UNIBUS Parity Error
Function: Set if an error occurs which resulted in the UNIBUS Map asserting the parity error signal on the UNIBUS.

Bit: 12 **Name:** UNIBUS Multiple Parity Error
Function: Set if an error occurs which caused the parity error to be asserted on the UNIBUS with the Error Address Register locked by a previous error.

Bit: 11 **Name:** CPU Error
Function: Set if any memory error occurs during a cache CPU cycle.

Bit: 10 **Name:** UNIBUS Error
Function: Set if any memory errors occur during a cache cycle from the UNIBUS.

Bit: 9 **Name:** CPU UNIBUS Abort
Function: Set if the processor traps to vector 114 because of UNIBUS parity error on a DATI or DATIP memory cycle.

Bit: 8 **Name:** Error in Maintenance
Function: Set if an error occurs when any bit in the Maintenance Register is set. The Maintenance Register will then be cleared.

Bit: 7:6 **Name:** Data Memory
Function: These bits are set if a parity error is detected in the fast data memory in the cache. Bit 7 is set if there is an error in Group 1; bit 6 for Group 0.

Bit: 5:4 **Name:** Address Memory
Function: These bits are set if a parity error is detected in the address memory in the cache. Bit 5 is set if there is an error in Group 1; bit 4 for Group 0.

Bit: 3:2 **Name:** Main Memory
Function: These bits are set if a parity error is detected on data from main memory. Bit 3 is set if there is an error in either byte of the odd word: bit 2 for the even word. (Main memory always transfers two words at a time.) An abort occurs if the error is in the word needed by a CPU reference. A trap occurs if the error is in the other word, or if it is a UNIBUS reference.

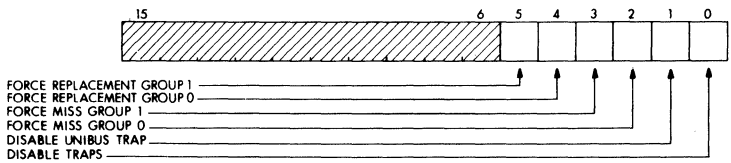
Bit: 1 **Name:** Main Memory Address Parity Error
Function: Set if there is a parity error detected on the address and control lines on the main memory bus.

Bit: 0 **Name:** Main Memory Timeout
Function: Set if there is no response from main memory. For CPU cycles, this error causes an abort. When a UNIBUS device requests a nonexistent location, this bit will set, cause a timeout on the UNIBUS, and then cause the CPU to trap to vector 114.

The bits are cleared on power-up or by Console Start. They are unaffected by a RESET instruction.

When writing to the Memory System Error Register, a bit is unchanged if a 0 is written to that bit, and it is cleared if a 1 is written to that bit. Thus, the register is cleared by writing the same data back to the register. This guarantees that if additional error bits were set between the read and the write, they will not be inadvertently cleared.

Control Register 17 777 746



Bit: 5:4 **Name:** Force Replacement
Function: Setting these bits forces data replacement within a Group in the cache by main memory data on a read miss. Bit 5 selects Group 1 for replacement; bit 4 selects Group 0.

Bit: 3:2 **Name:** Force Miss
Function: Setting these bits forces misses on reads to the cache. Bit 3 forces misses on Group 1; bit 2 forces misses on Group 0. Setting both bits forces all cycles to main memory.

Bit: 1 **Name:** Disable UNIBUS Trap
Function: Set to disable traps to vector 114 when the parity error signal is placed on the UNIBUS.

Bit: 0 **Name:** Disable Traps
Function: Set to disable traps from non-fatal errors.

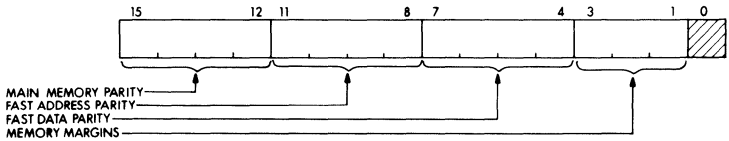
Bits 5 through 0 are read/write. The bits are cleared on power-up or by Console Start.

The PDP-11/70 can run in a degraded mode if problems are detected in the cache. If Group 0 of the cache is malfunctioning, it is possible to force all operations through Group 1. If bits 2 and 5 of the Control Register are set, and bits 3 and 4 are clear, the CPU will not be able to read data from Group 0, and all main memory data replacements will occur within Group 1. In this manner, half the cache will be operating. But system throughput will not decrease by 50%, since the statistics of read hit probability will still provide reasonably fast operation.

If Group 1 is malfunctioning, bits 3 and 4 should be set, and bits 2 and 5 cleared, so that only Group 0 is operating. If all of the cache is malfunctioning, bits 2 and 3 should be set. The cache will be by-passed, and all references will be to main memory.

Bits 1 and 0 can be set to disable trapping; more memory cycles will be performed, but overall system operation will produce correct results.

Maintenance Register 17 777 750



Bit: 15:12 **Name:** Main Memory Parity
Function: Setting these bits causes the four parity bits to be 1s. There is one bit per byte; there are four bytes in the data block.

Bit Set	Byte
15	odd word, high byte
14	odd word, low byte
13	even word, high byte
12	even word, low byte

Bit: 11:8 **Name:** Fast Address Parity
Function: Setting these bits causes the four parity bits for fast address memory to be wrong. Bits 11 and 10 affect Group 1; bits 9 and 8 affect Group 0.

Bit: 7:4 **Name:** Fast Data Parity
Function: Setting these bits causes the four parity bits to be 1s.

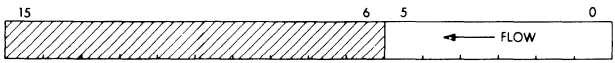
Bit Set	Byte
7	Group 1, high byte
6	Group 1, low byte
5	Group 0, high byte
4	Group 0, low byte

Bit: 3:1 **Name:** Memory Margins
Function: These bits are encoded to do maintenance checks on main memory.

Bit 3	Bit 2	Bit 1	CORE	MOS
0	0	0	Normal operation	Normal operation
0	0	1	Check wrong address parity	Check wrong address parity
0	1	0	Early strobe margin	Early strobe margin
0	1	1	Late strobe margin	Late strobe margin
1	0	0	Low current margin	Low current margin
1	0	1	High current margin	High current margin
1	1	0	(reserved)	(reserved)
1	1	1	(reserved)	(reserved)

All of main memory is margined simultaneously.

Hit/Miss Register 17 777 752



This register indicates whether the six most recent references by the CPU were hits or misses. A 1 indicates a read hit; a 0 indicates a read miss or a write. The lower numbered bits are for the more recent cycles.

All the bits are read-only. The bits are undetermined after a power-up. They are not affected by a RESET instruction.

HIGH-SPEED CONTROLLERS

Mounting Space

The PDP-11/70 CPU assembly provides dedicated, prewired space for up to four high-speed I/O controllers. Refer to Figure 10-11. DC power for the controllers is derived from the cabinet power supply.

Interfacing

Each group of mass storage peripherals communicates with its high-speed controller through a separate high-speed I/O bus. This I/O bus consists of a set of 56 signals for data, control, status, and parity. High transfer rate is achieved by using synchronous block transfer of data simultaneously with asynchronous control information. The controller contains an eight-word data buffer.

Data are transferred in a Direct Memory Access (DMA) mode. An internal 32-bit wide data bus transfers four bytes in parallel between memory and the high-speed controllers. The Priority Arbitration logic within the cache memory controls the timing of data transfers; but the cache itself is not used for storage. Data transfers are between main memory and the mass storage peripheral. The cache is not affected, except that on a write hit from the I/O bus to memory, the valid bit is cleared for that particular two-word block within the cache. In this way, the affected areas of the cache are flagged as having incorrect data, but main memory always contains the correct, updated information.

The UNIBUS plays a subordinate role with respect to the high-speed controllers. The UNIBUS is used:

1. To supply control and status information
2. To generate an interrupt request (by the controller)

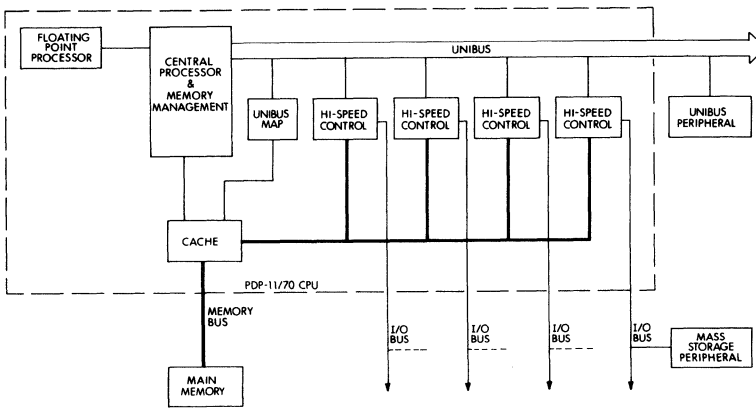


Figure 10-11 PDP-11/70 Block Diagram

The UNIBUS is not used for data transfer.

The registers within the controller (which can be read and written directly) are addressed from the UNIBUS. In a typical DMA transfer, the registers would first be loaded with the following data:

1. Number of words to be transferred
2. Starting address in memory for data transfers
3. Control information specifying the device and type of operation

Increased Data Transfer Rate

The architecture of the PDP-11/70 allows overlapping of some operations, providing faster program execution speed. CPU and UNIBUS read hits with the cache memory are overlapped with mass storage device reads from main memory. It is possible to overlap the read cycles of several mass storage devices.

Parity

Parity is generated and checked in the system for data and address and control information, to ensure the integrity of the information transferred. The RHCS3 register in the controller is used to indicate the occurrence of parity errors during memory transfers.

REGISTERS

The controller contains six local registers, plus part of one more which is shared with the mass-storage device. Other registers needed by the particular mass storage system and device are contained in the device itself. Appendix B contains information about the mass storage device

registers. For a detailed description of the mass storage device registers, please refer to the PERIPHERALS Handbook.

Controller Registers

RHCS1	Control and Status 1 (partial)
RHWC	Word Count
RHBA	Bus Address (Main Memory Bus)
RHBAE	Bus Address Extension (Main Memory Bus)
RHCS2	Control and Status 2
RHCS3	Control and Status 3
RHDB	Data Buffer (Maintenance)

CONSOLE OPERATION

The PDP-11/70 console allows direct control of the computer system. It contains a power switch for the CPU, which is also usually used as the Master Switch for the system. The console is used for starting, stopping, resetting, and debugging. Lights and switches provide the facilities for monitoring operation, system control, and maintenance. Debugging and detailed tracing of operations can be accomplished by having the computer execute single instructions or single cycles. Contents of all locations can be examined, and data can be entered manually from the console switches.

GENERAL

The PDP-11/70 Operator's Console provides the following facilities:

a) Power Switch (with a key lock)

b) ADDRESS Register Display (22 bits)

c) DATA Register Display (16 bits), plus Parity Bit Low Byte and Parity Bit High Byte

d) Switch Register (22 switches)

e) Error Lights
   - ADRS ERR (Address Error)
   - PAR ERR (Parity Error)

f) Processor State Lights (7 indicators)
   - RUN
   - PAUSE
   - MASTER
   - USER
   - SUPERVISOR

KERNEL
DATA

g) Mapping Lights
   16 BIT
   18 BIT
   22 BIT

h) ADDRESS Display Select Switch (8 positions)
   USER I
   USER D
   SUPER I (Virtual)
   SUPER D
   KERNEL I
   KERNEL D
   PROG PHY (Program Physical)
   CONS PHY (Console Physical)

i) DATA Display Select Switch (4 positions)
   DATA PATHS
   BUS REGISTER
   μADRS FPP/CPU
   DISPLAY REGISTER

j) Lamp Test Switch

k) Control Switches
   LOAD ADRS
   EXAM (Examine)
   DEP (Deposit)
   CONT (Continue)
   ENABLE/HALT
   S INST/S BUS CYCLE (Single Instruction/Single Bus Cycle)
   START

STARTING AND STOPPING

Starting

Once power is on, execution can be started by placing the ENABLE/HALT switch in the ENABLE position, putting the starting address in the Switch Register, and depressing the LOAD ADRS switch. Verify in the Address Display Lights that the address was entered correctly, then depress the START switch. The computer system will be cleared and will then start running. Once execution has begun, depressing the START switch again has no effect.

If the system needs to be initialized but execution is not wanted, the START switch should be depressed while the HALT/ENABLE switch is in the HALT position.

Stopping
Set the ENABLE/HALT switch to the HALT position. The computer will stop execution, but the contents of all memory locations will be retained. The switch can then be set to the ENABLE position with no effect on the system.

NOTE

NPRs are still serviced after HALT from the console if S BUS CYCLE is disabled.

Continuing
After the computer has been stopped, execution can be resumed from the point at which it was halted by using the CONT (Continue) Switch. The function of the CONT Switch depends on the position of the ENABLE/HALT Switch:

ENABLE (up)	CPU resumes normal execution.
HALT (down)	The mode is used for debugging purposes and forces execution of a single instruction or a single bus cycle.

REFERENCING MEMORY

Unmapped References
When performing unmapped memory references from the console, the Address Select Switch must be set to CONS PHY. This means that the 22-bit address entered in the Switch Register should be the physical address desired. To examine a memory location, depress the LOAD ADRS switch and then the EXAM switch. The address referenced will appear in the Address Display Lights. The DATA Select switch should be selecting DATA PATHS, and the contents of that location are displayed in the Data Display Lights. To deposit information into a memory location, depress the LOAD ADRS switch, then enter the desired data in the Switch Register and raise the DEP switch. The DATA Select switch should be in the DATA PATHS position, and the deposited information will appear in the DATA Display Lights.

Mapped References
Sometimes, when software is running with Memory Management enabled, the physical addresses generated are not known. This makes examining and depositing memory locations more difficult. For this reason, the six positions, KERNEL I through USER D, of the ADDRESS Select switch are provided. When doing a memory reference, the low-order 16 bits of the Switch Register are considered to be a Virtual Address and are relocated by Memory Management using the set of PAR/PDRs indicated by the ADDRESS Select switch.

To examine a memory location, depress the LOAD ADRS switch and the EXAM switch. The DATA Select switch should be selecting DATA PATHS, and the contents of that location are displayed in the DATA Display Lights. To deposit information into a memory location, depress the LOAD ADRS switch, then enter the desired data in the Switch Register and raise the DEP switch. The Data Select Switch should be in the DATA PATHS position, and the deposited information will appear in the DATA Display Lights.

The PROG PHY (Program Physical) position of the ADDRESS Select switch is used as a debugging tool. After an examine or deposit has been performed on a virtual address, changing the ADDRESS Select switch to select PROG PHY will display the Physical Address generated by Memory Management in the Address Display Lights. Using the PROG PHY position in any other way will produce meaningless results.

NOTE

An EXAM or DEP operation which causes an addressing error (ADRS ERR or PAR ERR) will be aborted and must be corrected by performing a new LOAD ADRS operation with a valid address.

STEP OPERATIONS

Performing more than one EXAM operation in a row or more than one DEP operation in a row results in a STEP operation. Depressing the EXAM switch after previous examination of a location displays the contents of the next location in memory. Raising the DEP switch after a previous deposit into a memory location causes the current contents of the Switch Register to be deposited into the next location in memory.

In each case, the Address Display is updated by two to hold the value of the now current address. This allows consecutive EXAM operations and consecutive DEP operations without the use of the LOAD ADRS switch. An EXAM-STEP or DEP-STEP operation will not cross a 32 Kword memory block boundary.

NOTE

The EXAM and DEP switches are coupled to enable an EXAM—DEP—EXAM sequence to be carried out on a location without having to do extra LOAD ADRS operations. The following example deposits values into consecutive memory locations.

Operation (Activate Switch)	Location shown in ADDRESS Display
LOAD ADRS	X
EXAM	X
DEP	X
EXAM	X
EXAM (result is EXAM-STEP)	X+2
DEP	X+2
EXAM	X+2

GENERAL REGISTERS

The General Registers can be examined and deposited using the EXAM and DEP switches provided the previous LOAD ADRS operation loaded the Address Display with a "register address."

Address	Register
17 777 700	Register 0 (Set 0)
.	.
.	.
.	.
.	.
17 777 705	Register 5 (Set 0)
17 777 706	Register 6, Kernel Mode
17 777 707	Program Counter
17 777 710	Register 0 (Set 1)
.	.
.	.
.	.
.	.
17 777 715	Register 5 (Set 1)
17 777 716	Register 6, Supervisor Mode
17 777 717	Register 6, User Mode

Examining and depositing into General Register Addresses is independent of the ADDRESS Select switch. It is not possible to be mapped to a General Register.

EXAM-STEP and DEP-STEP operations can be performed on the General Registers, similar to those for memory locations, except that:

a) ADDRESS Display is incremented by one (instead of two)

b) The STEP after address 17 777 717 is 17 777 700, such that the addresses are looped

c) It is not possible to STEP up to the first General Register (17 777 700) from 17 777 676

SINGLE INSTRUCTION/SINGLE BUS CYCLE

Once the machine is halted, a useful debugging tool is being able to execute code a small segment at a time. The S INST/S BUS CYCLE (Single Instruction/Single Bus Cycle) switch provides that capability. The ENABLE/HALT switch must be in the HALT position. To start execution of a segment, depress the CONT switch. How much is executed is a function of the S INST/S BUS CYCLE switch.

Position	
S INST	Depressing the CONT Switch will result in the execution of one instruction. This means that the machine state can be determined after each instruction. Examining and depositing into memory locations is a method of accomplishing this. The contents of the DATA Display Lights are not necessarily meaningful.
S BUS CYCLE	For this mode to have any meaning, the DATA Select switch should be selecting the BUS REG (Bus Register). Depressing the CONT Switch will cause execution until the end of the next bus cycle. The Address Display Lights will then contain the address of the location at which the bus cycle was performing. (Virtual or Physical, depending on the position of the ADDRESS Select switch). The DATA Display Lights, on a read operation, will contain the data that were read (this could be an instruction or data). During a write operation, the lights will contain the data just written (except during a stack operation or Floating Point Instruction). Examine and deposit operations cannot be used in this mode. Depressing the LOAD ADRS, EXAM, or DEP switch will not cause

anything to happen. If an examine or deposit operation is desired, the S INST/S BUS CYCLE switch should be changed to select S INST and the CONT switch should be depressed once. (This will cause execution until the end of the current instruction.) The system will then be ready to perform an examine or deposit.

FUNCTIONS OF SWITCHES & INDICATORS

Power Switch

OFF	Power to the processor is OFF.
POWER	Power to the processor is ON, and all console switches function normally.
LOCK	Power to the processor is ON, but the seven control switches LOAD ADRS through START are disabled. All other switches are functional.

Control Switches

When a LOAD ADRS switch is depressed, the contents of the Switch Register are loaded into the ADDRESS Display. The address displayed in the Address Display Lights is a function of the position of the ADDRESS Select switch.

EXAM (Examine)

Depressing the EXAM switch causes the contents of the current location specified in the Address Display to be displayed in the DATA Display Register when the DATA Select switch is in the DATA PATHS position. The address in the Address Display will be mapped or unmapped depending on the position of the ADDRESS Select switch. The location displayed in the Address Display Lights is also a function of that switch.

DEP (Deposit)

Raising the DEP switch causes the current contents of the Switch Register to be deposited into the address specified by the current contents of the Address Display.

The address in the Address Display will be mapped or unmapped depending on the position of the ADDRESS Select switch. The location displayed in the Address Display Lights is also a function of that switch.

CONT (Continue)
Depressing the CONT switch causes the CPU to resume execution. The CONT switch has no effect when the CPU is in RUN state.

ENABLE/HALT
The ENABLE/HALT switch is a two-position switch used to stop machine execution and to enable the system to run.

S INST/S BUS CYCLE (Single Instruction/Single Bus Cycle)
The S INST/S BUS CYCLE switch affects only the operation of the CONT switch. It controls whether the machine stops after instructions or bus cycles. This switch has no effect on any switches when the ENABLE/HALT switch is set to ENABLE.

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

ENABLE	Starts execution
HALT	Clears the computer system

Switch Register
The switches are used to manually load data or an address into the processor, as determined by the control switches and the ADDRESS Select switch.

Note that bits 0 to 15 of the current setting of the Switch Register may be read under program control from a read-only register at address 17 777 570.

Lamp Test
The Lamp Test switch (which is not labeled) is located between the Switch Register and the LOAD ADRS switch. It is used for maintenance purposes. When the Lamp Test switch is raised, all console indicator lights should go on. An indicator which does not light is defective and should be replaced.

Address Select Switch

VIRTUAL (six-position for User, Supervisor, and Kernel)	Uses a 16-bit virtual address where bits 16 to 21 are always OFF
CONS PHY (Console Physical)	Uses a 22-bit physical address to perform console operations (e.g., LOAD ADRS, EXAM, and DEP)

PROG PHY (Program Physical)	Displays the 22-bit physical address of the current bus cycle that was generated by the Memory Management Unit

Address Display
The ADDRESS Display lights are used to show the address of data being examined or just deposited. The address is interpreted as a Virtual or Physical Address as determined by the ADDRESS Select switch.

Data Select Switch

DATA PATHS	The normal display mode, shows examined or deposited data
BUS REG	The internal CPU register used for bus cycles
μADRS FPP/CPU	The ROM address, FPP control microprogram (bits 15 to 8) and the CPU control microprogram (bits 7 to 0)
DISPLAY REGISTER	The contents of the Display Register; this has an address of 17 777 570

Data Display
The Data Display lights are used to show the 16-bit word data just examined or deposited, or other data within the CPU. The PARITY HIGH & LOW lights indicate the parity bit for the respective bytes on read operations; on write operations the bits are off. The interpretation of the data is determined by the DATA Select switch.

Status Indicator Lights
ERROR INDICATORS

PAR ERR	Lights to indicate a parity error during a reference to memory.
ADRS ERR	Lights to indicate any of the following addressing errors: a) Reference to nonexistent memory b) Access control violation c) Reference to unassigned memory pages

PROCESSOR STATE	
RUN	The CPU is executing program instructions. If the instruction being executed is a WAIT instruction, the RUN light will be on. The CPU will proceed from the WAIT on receipt of an external interrupt, or on console intervention.
PAUSE	The CPU is inactive because the current instruction execution has been completed as far as possible without more data from the UNIBUS or memory, or the CPU is waiting to regain control of the the UNIBUS (UNIBUS mastership).
MASTER	The CPU is in control of the UNIBUS (UNIBUS Master only when it needs the UNIBUS). The CPU relinquishes control of the UNIBUS during DMA and NPR data transfers.
MODE	
USER	The CPU is executing program instructions in User mode.
SUPER (Supervisor)	The CPU is executing program instructions in Supervisor mode.
KERNEL	The CPU is executing program instructions in Kernel mode.
DATA	If on, the last memory reference was to D address space in the current CPU mode. If off, the last memory reference was to I address space in the current mode.
ADDRESS	
16 BIT	Lights when the CPU is using 16-bit mapping.
18 BIT	Lights when the CPU is using 18-bit mapping.
22 BIT	Lights when the CPU is using 22-bit mapping.

M9301-YC, YH/M9312 BOOTSTRAP LOADER

Features

- Contains bootstrap routines for a wide range of storage media
- Allows bootstrapping of any drive unit on a particular controller
- Runs diagnostic programs to test the basic CPU, cache, and main memory
- Allows booting to selected physical memory segments in 32K increments
- Switch-selectable default loading device

Description

The M9312 is a dedicated diagnostic bootstrap loader for use with the PDP-11/70. It contains a ROM organized as 512 16-bit words which are separated into hardware verification programs and bootstrap routines. They are double-height extended modules which occupy rows E and F of slot one in the PDP-11/70 CPU.

Diagnostics (M9312)

The M9312 provides basic diagnostic tests for the CPU, memory, and cache when used with PDP-11/60 and PDP-11/70 computers. All diagnostic tests reside in ROM (read-only memory) locations 765 000 through 765 776 (console emulator routine is eliminated). These diagnostics test the basic CPU including the branches, the registers, all addressing modes, and many of the instructions in the PDP-11 repertoire. Memory from virtual address 1000 to the highest available address up to 28K will also be checked. After main memory has been verified, with the cache off, the cache memory will be tested to verify that hits occur properly. Main memory will be scanned again to ensure that the cache is working properly throughout the 28K of memory to be used in the boot operation. If one of the cache memory tests fails, the operator can attempt to boot the system anyway by pressing CONTINUE. This will cause the program to force misses in both groups of the cache before going to the bootstrap section of the program. The following is a list of M9312 diagnostic tests.

TEST 1	This test verifies the unconditional branch
TEST 2	Test CLR, MODE 0, and BMI, BVS, BHI, BLT, BLOS
TEST 3	Test DEC, MODE 0, and BPL, BEQ, BGE, BLE
TEST 4	Test ROR, MODE 0, and BVC, BHIS, BNE
TEST 5	Test register data path

TEST 6	Test ROL, BCC, BLT
TEST 7	Test ADD, INC, COM and BCS, BLE
TEST 10	Test ROR, DEC, BIS, ADD, and BLO
TEST 11	Test COM, BIC, and BGT, BLE
TEST 12	Test SWAB, CMP, BIT, and BNE, BGT
TEST 13	Test MOVB, SOB, CLR, TST and BPL, BNG
TEST 14	Test JSR, RTS, RTI, and JMP
TEST 15	Test main memory from virtual 1000 to last address; cache memory diagnostic tests
TEST 16	Test cache data memory
TEST 17	Test memory with the data cache on

DIAGNOSTICS (M9301-YC, -YH)

The diagnostic portion of the program will test the basic CPU, including the branches, the registers, all addressing modes, and most of the instructions in the PDP-11 repertoire. It will then set the stack pointer to Kernel D-space PAR 7. It will also turn on, if requested, Memory Management and the UNIBUS map, and will check memory from virtual address 1000 to 157 776. After main memory has been verified, with the cache off, the cache memory will be tested to verify that hits occur properly. Main memory will be scanned again to ensure that the cache is working properly throughout the 28K of memory to be used in the boot operation.

If one of the cache memory tests fails, the operator can attempt to boot the system anyway by pressing CONTINUE. This will cause the program to force misses in both groups of the cache before going to the bootstrap section of the program. A listing of the M930-YC, -YH diagnostic tests follows.

TEST 1	This test verifies the unconditional branch
TEST 2	Test CLR, MODE 0, and BMI, BVS, BHI, BLOS
TEST 3	Test DEC, MODE 0, and BPL, BEQ, BGE, BGT, BLE
TEST 4	Test ROR, MODE 0, and BVC, BHIS, BHI, BNE
TEST 5	Test BHI, BLT, and BLOS
TEST 6	Test BLE, and BGT

TEST 7	Test register data path and modes 2, 3, 6
TEST 10	Test ROL, BCC, BLT, and MODE 6
TEST 11	Test ADD, INC, COM, and BCS, BLE
TEST 12	Test ROR, BIS, ADD, and BLO, BGE
TEST 13	Test DEC and BLOS, BLT
TEST 14	Test COM, BIC, and BGT, BGE, BLE
TEST 15	Test ADC, CMP, BIT, and BNE, BGT, BEQ
TEST 16	Test MOVB, SOB, CLR, TST and BPL, BNE
TEST 17	Test ASR, ASL
TEST 20	Test ASH and SWAB
TEST 21	Test 16 Kernel PARs
TEST 22	Test and load KIPDRs
TEST 23	Test JSR, RTS, RTI, and JMP
TEST 24	Load and turn on Memory Management and the UNIBUS map
TEST 25	Test main memory from virtual 1000 to 28K
TEST 26	Test cache data memory
TEST 27	Test virtual 28K with cache on

ERROR RECOVERY

If the processor halts in one of the two cache tests, the error is recoverable. By pressing CONTINUE, the program will either attempt to finish the test (if at either 17 773 644 or 17 773 736) or force misses in both groups of the cache and attempt to boot the system monitor with the cache fully disabled (if at 17 773 654, 17 773 746, or 17 773 764). The run time for this program is approximately three seconds.

CHAPTER 11
PDP-11 FLOATING POINT

INTRODUCTION

The PDP-11 processor family has two sets of floating point instructions:

1. The FIS (Floating Instruction Set) option, consisting of four instructions (FADD, FSUB, FMUL, FDIV) that operate on single-precision floating point formats, is available on the LSI-11 and LSI-11/2. Please refer to the Microcomputer Handbook for a description of FIS.
2. The FP11 instruction set supports both single- and double-precision floating point arithmetic. It is available as a microcode option, KEF11-AA, for the PDP-11/23 and PDP-11/24. It is also available as a hardware option on the PDP-11/34 (FP11-A), PDP-11/44 (FP11-F) and PDP-11/70 (FP11-C). In this discussion, the term floating point processor (FPP) will be used to refer to the hardware or microcode implementation of the FP11 instruction set.

A floating point processor is much faster and more effective for high-speed numerical data handling than software floating point routines. Users who program in FORTRAN, BASIC and APL find that the FPP gives them the speed and capability that they require for data and number manipulation.

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)
- High-speed operation
- Single- and double-precision (32- or 64-bit) floating point modes
- Flexible addressing modes
- Six 64-bit floating point accumulators
- Error recovery aids

ARCHITECTURE

The floating point processors contain scratch registers, a floating exception address pointer (FEA), a program counter, a set of status and error registers, and six general-purpose accumulators, AC0-AC5. (Please refer to Figure 11-1.)

The accumulators are 32 or 64 bits long, depending on the instruction and FPP status. In a 32-bit instruction, only the leftmost 32 bits are used.

The six floating point accumulators are used in numeric calculations and in interaccumulator data transfers. The first four accumulators (AC0-AC3) are also used for all data transfers between the FPP and the general registers, or memory.

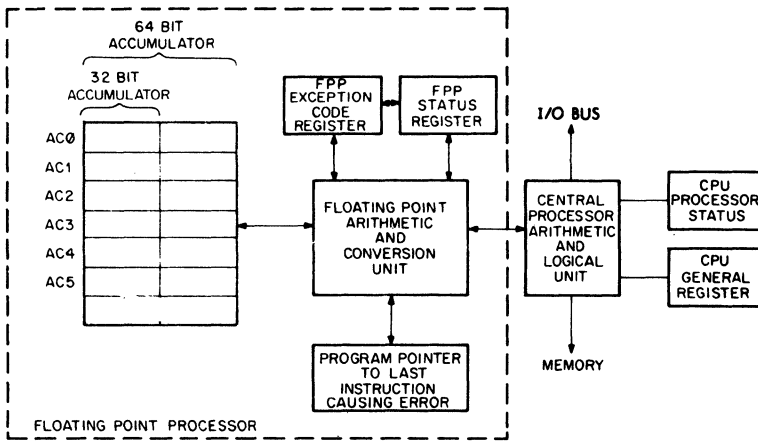


Figure 11-1 Conceptual Structure of the Floating Point Processor

OPERATION

A floating point processor functions as an integral part of the central processor. It operates using similar address modes and 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.

When an FPP instruction is fetched from memory, the FPP will execute that instruction in parallel with the CPU as the CPU continues *its* instruction sequence. The CPU is delayed a very short period of time during the FPP instruction fetch operation, and then is free to proceed independently of the FPP. The interaction between the two processors is automatic, permitting a program to take full advantage of the parallel operation of the two processors, by the intermixing of FPP and CPU instructions. This is all accomplished by the hardware of the processors. When an FPP instruction is encountered in a program, the CPU first initiates floating point handshaking and calculates the address of the operand. It then checks the status of the FPP. If the FPP is

busy, the CPU waits until it receives a DONE signal before continuing execution of the program. For example:

```
        LDD(R3)+,AC3        ;Pick up constant operand
                            ;and place it in AC3
ADDLP:  LDD(R3)+,AC0        ;Load AC0 with next value
                            ;in table
        MULD 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

Mathematically, a floating point number may be defined as having the form (2**K)*f, where K is an integer and f is a fraction. For a nonvanishing number, K and f are uniquely determined by imposing the condition $\frac{1}{2} \leq f < 1$. The fractional part, f, of the number is then said to be normalized. For the number 0, f must be assigned the value 0, and the value of K is indeterminate.

The FP11 floating point data formats are derived from this mathematical representation for floating point numbers. Two types of floating point data are provided. In single-precision, or floating mode, the data are 32 bits long. In double-precision, or double mode, the data are 64 bits long. Sign magnitude notation is used.

Nonvanishing Floating Point Numbers

The fractional part, f, is assumed to be normalized, so that its most significant bit must be 1. This 1 is the **hidden** bit; it is not stored explicitly in the data word, but the microcode restores it before carrying out arithmetic operations. The floating and double modes respectively reserve 23 and 55 bits for f. These bits, with the hidden bit, imply effective fractions of 24 bits and 56 bits.

Eight bits are reserved for storage of the exponent K in excess 128

(200_8) notation (i.e., K + 200_8), giving a biased exponent. Thus, exponents from −128 to +127 are represented by 0 to 377_8, or 0 to 255_{10}. For reasons listed below, a biased exponent of 0 (true exponent of -200_8), is reserved for floating point 0. Thus, exponents are restricted to the range −127 to +127 inclusive (-177_8 to $+177_8$) or, in excess 200_8 notation, 1 to 377_8.

The remaining bit of the floating point word is the sign bit. The number is negative if the sign bit is a 1.

Floating Point Zero

Because of the hidden bit, the fractional part is not available to distinguish between 0 and nonvanishing numbers whose fractional part is exactly ½. Therefore, the FP11 reserves a biased exponent of 0 for this purpose, and any floating point number with a biased exponent of 0 either traps or is treated as if it were an exact 0 in arithmetic operations. An exact or clean 0 is represented by a word whose bits are all 0s. A dirty 0 is a floating point number with a biased exponent of 0 and a nonzero fractional part. An arithmetic operation for which the resulting true exponent exceeds 277_8 is regarded as producing a floating overflow; if the true exponent is less than -177_8, the operation is regarded as producing a floating underflow. A biased exponent of 0 can thus arise from arithmetic operations as a special case of overflow (true exponent = -200_8). Only eight bits are reserved for the biased exponent. The fractional part of results obtained from such overflow and underflow is correct.

The Undefined Variable

The undefined variable is defined as any bit pattern with a sign bit of 1 and a biased exponent of 0. The term **undefined variable** is used, for historical reasons, to indicate that these bit patterns are not assigned a corresponding floating point arithmetic value. Note that the undefined variable is frequently referred to as −0 elsewhere in this specification.

A design objective of the FP11 was to assure that the undefined variable would not be stored as the result of any floating point operation in a program run with the overflow and underflow interrupts disabled. This objective is achieved by storing an exact 0 on overflow and underflow, if the corresponding interrupt is disabled. This feature, together with an ability to detect reference to the undefined variable implemented by the FIUV bit mentioned later, is intended to provide the user with a debugging aid. If −0 occurs, it did not result from a previous floating point arithmetic instruction.

Floating Point Data

Floating point data are stored in words of memory as illustrated in Figures 11-2 and 11-3.

The FP11 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.

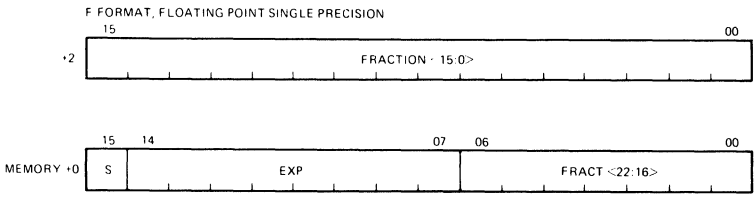


Figure 11-2 Single-Precision Format

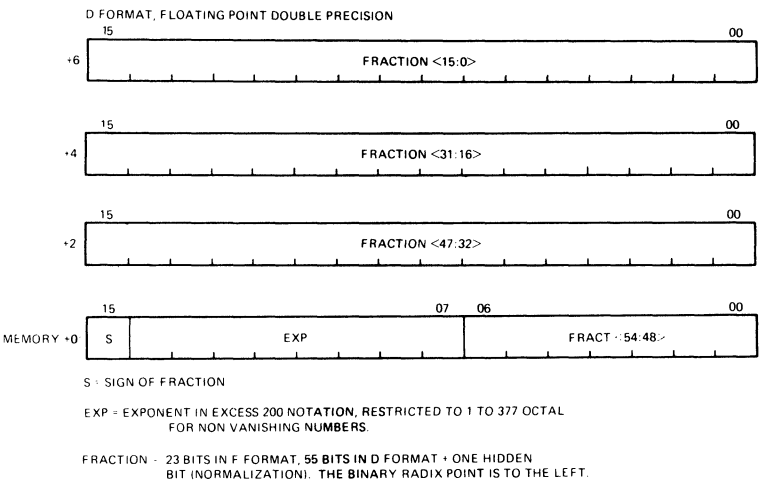


Figure 11-3 Double-Precision Format

FLOATING POINT STATUS REGISTER (FPS)

This register provides mode and interrupt control for the floating point unit and conditions resulting from the execution of the previous instruction.

For the purposes of discussion, a set bit = 1 and a reset bit = 0. Three bits of the FPS register control the modes of operation.

- Single/Double: floating point numbers can be either single- or double-precision.
- Short/Long: 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.

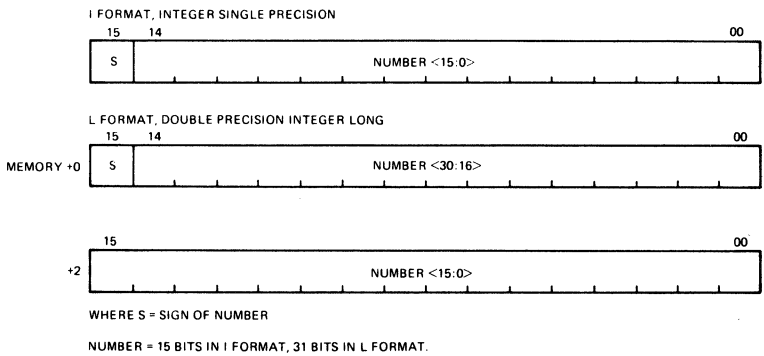


Figure 11-4 2's Complement Format

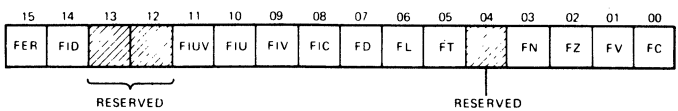


Figure 11-5 Floating Point Status Register

The FPS register contains an error flag and four condition codes (five 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 (FP11-C only)
- Attempt to divide by zero
- Illegal floating opcode

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 14 bits described above, five, the error flag and condition codes, are set by the FPP as part of the output of a floating point instruction. Any of the mode and interrupt control bits (except the FP11-C FMM bit) may be set by the user; the LDFS instruction is available for this purpose. These 14 bits are stored in the FPS register as follows.

FPS Register Bits

Bit: 15 **Name:** Floating Error (FER)
Function: The FER bit is set by a floating point instruction if:

- Division by zero occurs
- Illegal opcode occurs
- Any of the remaining errors occur and the corresponding interrupt is enabled

This action is independent of the FID bit status.

Also note 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 (the RESET instruction does not clear the FER bit). This means that the FER bit is up to date only if the most recent floating point instruction produced a floating point exception.

Bit: 14 **Name:** Interrupt Disable (FID)
Function: If the FID is set, all floating point interrupts are disabled.

The FID bit is primarily a maintenance feature. It should normally be clear. In particular, it must be clear if one wishes to assure that storage of −0 by the FPP is always accompanied by an interrupt.

Throughout 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: 13
Function: Reserved for future DIGITAL use.

Bit: 12
Function: Reserved for future DIGITAL use.

Bit: 11 **Name:** Interrupt on Undefined Variable (FIUV)
Function: 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, ABS, and TST, for which it 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 a simultaneous interrupt.

Bit: 10 **Name:** Interrupt on Underflow (FIU)
Function: 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 correct. The biased exponent will be too large by 400_8, except for the special case of 0, which is correct. An exception is discussed later 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: 9 **Name:** Interrupt on Overflow (FIV)
Function: When the FIV bit is set, floating overflow will cause an interrupt. The fractional part of the result of the operation causing the overflow will be correct. The biased exponent will be too small by 400_8.

If the FIV 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: 8 **Name:** Interrupt on Integer Conversion Error (FIC)
Function: When the FIC bit is set and 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 detailed 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 (bit 6).

Bit: 7 **Name:** Floating Double-Precision Mode (FD)
Function: The FD bit determines the precision that is used for floating point calculations. When set, double-precision is assumed; when reset, single-precision is used.

Bit: 6 **Name:** Floating Long Integer Mode (FL)
Function: The FL bit is active in conversion between integer and floating point format. When set, the integer format assumed 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: 5 **Name:** Floating Chop Mode (FT)
Function: When the FT bit is set, the result of any arithmetic operation is chopped (or truncated). When reset, the result is rounded.

Bit: 4 **Name:** Floating Maintenance Mode (FMM)
Function: FP11-C only. When set, the FPP is in maintenance mode. The FMM bit can be set only in Kernel mode.

Bit: 3 **Name:** Floating Negative (FN)
Function: FN is set if the result of the last floating point operation was negative, otherwise it is reset.

Bit: 2 **Name:** Floating Zero (FZ)
Function: FZ is set if the result of the last floating point operation was 0, otherwise it is reset.

Bit: 1 **Name:** Floating Overflow (FV)
Function: FV is set if the last floating point operation resulted in an exponent overflow, otherwise it is reset.

Bit: 0 **Name:** Floating Carry (FC)
Function: FC is set if the last operation resulted in a carry of the most significant bit. This can only occur in floating or double to integer conversion.

FLOATING EXCEPTION CODE AND ADDRESS REGISTERS

One interrupt vector is assigned to take care of all floating point exceptions (location 244). The six possible errors are coded in the four-bit floating exception code (FEC) register as follows:

2	Floating opcode error
4	Floating divide by zero
6	Floating or double to integer conversion error
8	Floating overflow
10	Floating underflow
12	Floating undefined variable
14	Maintenance trap (FP11-C only)

The address of the instruction producing the exception is stored in the FEA (Floating Exception Address) register.

The FEC and FEA registers are updated when one of the following occurs:

- Divide by zero
- Illegal opcode
- 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 OPTION INSTRUCTION ADDRESSING

Floating point option instructions use the same type of addressing as 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 mode 0. In mode 0 the operand is located in the designated floating point accumulator, rather than in a central processor general register. The modes of addressing are as follows:

0 = FP11 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 four for F format and eight for D format.

In mode 0, the user can make use of all six FP11 accumulators (AC0-AC5) as source or destination. Specifying FP11 accumulators AC6 or AC7 will result in an illegal opcode trap. In all other modes, which involve transfer of data to or from memory or the general registers, the user is restricted to the first four FP11 accumulators (AC0-AC3). When reading or writing a floating point number from or to memory, the low memory word contains the most significant word of the floating point number and the high memory word the least significant word.

ACCURACY

The descriptions of the individual instructions include the accuracy at which they operate. An instruction or operation is regarded as "exact" if the result is identical to an infinite precision calculation involving the same operands. The *a priori* accuracy of the operands is thus ignored. 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 0 operand implies that the instruction is exact. The same holds for DIV if the 0 operand is the dividend. But if the divisor is 0, division is undefined and an interrupt occurs.

For nonvanishing floating point operands, the fractional part is binary normalized. It contains 24 bits or 56 bits for floating mode or double mode, respectively. For ADD, SUB, MUL, and DIV, two guard bits are necessary and sufficient for the general case to guarantee return of a chopped or rounded result identical to the corresponding infinite precision operation chopped or rounded to the specified word length. 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 ½ LSB. These error bounds are realized by the KEF11-AA for all instructions, and for most instructions by the FP11-A, FP11-C, and FP11-F. The FP11-A, FP11-C, and FP11-F have an error bound greater than ½ LSB for ADD and SUB. 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 ¾ LSB (FP11-C), or 33/64 LSB (FP11-A and FP11-F) which is slightly larger than the ½ LSB error bound for all other rounded operations.

The error bound for the FP11-C differs from the FP11-A and FP11-F, since the FP11-C carries three guard bits while the FP11-A and FP11-F carry seven guard bits.

In this handbook, an arithmetic result is called exact if no nonvanishing 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 1, the rounded result is the chopped result incremented by one LSB
- If the rounding bit is 0, 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 0
  - — is increased by rounding if the rounding bit is 1

Occurrence of floating point overflow and underflow is an error condition: the result of the calculation cannot be stored correctly because the exponent is too large to fit into the eight bits reserved for it.

However, the internal hardware has produced the correct answer. For the case of underflow, replacement of the correct answer by 0 is a reasonable resolution of the problem for many applications. This is done by 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 provide you with an opportunity to implement your own correction of an overflow or underflow condition. If such a condition occurs and the corresponding interrupt is enabled, the microcode stores the fractional part and the low eight bits of the biased exponent. The interrupt will take place, and you can identify the cause by examining the FIV (floating overflow) bit or the FEC (floating exception) register. For the standard arithmetic operations ADD, SUB, MUL, and DIV, the biased exponent returned by the instruction bears the following relation to the correct exponent generated by the microcode:

- On overflow, it is too small by 400_8
- On underflow, if the biased exponent is 0, it is correct; if it is not 0, it is too large by 400_8

Thus, with the interrupt enabled, enough information is available to determine the correct answer. You may, for example, rescale your variables (via STEXP and LDEXP) to continue a calculation. The accuracy of the fractional part is unaffected by the occurrence of underflow or overflow.

FLOATING POINT INSTRUCTIONS

Each instruction that manipulates a floating point number can operate on either single- or double-precision numbers, depending on the state of FD mode bit. Similarly, there is a mode bit FL that determines whether 32-bit integers or 16-bit integers are used in conversion between integer and floating point representation. FSRC and FDST use floating point addressing modes; SRC and DST use CPU addressing modes.

In the descriptions of the floating point instructions, the operations of the KEF11-AA, FP11-A, FP11-F, and FP11-C are identical, except where explicitly stated otherwise.

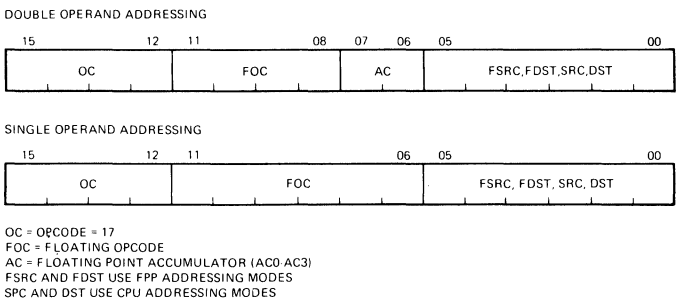


Figure 11-6 Single- and Double-Operand Addressing

Floating Point Instruction Format

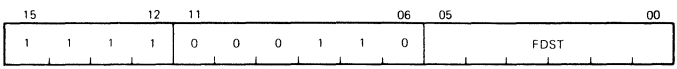
Mnemonic	**Description**
OC	Opcode = 17
FOC	Floating Opcode
AC	Contents of accumulator, as specified by AC field of instruction
fsrc	Address of floating point source operand
fdst	Address of floating point destination operand
f	Fraction
XL	Largest fraction that can be represented: 1 −2**(−24), FD=0, single precision 1 −2**(−56), FD=1; double precision
XLL	Smallest number that is not identically zero: 2**(−128)
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[(X)]	Absolute value of contents of memory location X
EXP[(X)]	Biased exponent of contents of memory location X
<	Less than

≤	Less than or equal to
>	Greater than
≥	Greater than or equal to
≠	Not equal to
LSB	Least significant bit

The accumulators are 32 or 64 bits long, depending on the instruction and FPP status. In a 32-bit instruction, only the leftmost 32 bits are used.

The six floating point accumulators are used in numeric calculations and in interaccumulator data transfers. The first four accumulators (AC0-AC3) are also used for all data transfers between the FPP and the general registers, or memory.

ABSF
ABSD
Make Absolute Floating/Double 1706 FDST



Format: ABSF FDST

Operation: If (fdst) < 0, (fdst) ← −(fdst).
If EXP[(fdst)] = 0, (fdst) ← exact 0.
For all other cases, (fdst) ← (fdst).

Condition Codes:
FC ← 0
FV ← 0
FZ ← 1 if (fdst) = 0, else FZ ← 0
FN ← 0

Description: Set the contents of fdst to its absolute value.

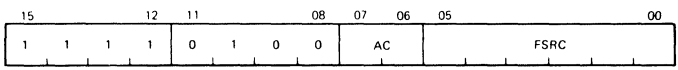
Interrupts: If FIUV is enabled, trap on −0 occurs after execution.
Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: If a −0 is present in memory and the FIUV bit is enabled, then an exact 0 is stored in memory. The condition codes reflect an exact 0 (FZ ← 1).

ADDF
ADDD

Add Floating/Double 172(AC)FSRC



Format: ADDF FSRC,AC

Operation: 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.

Condition Codes:
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 according to the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

1. Overflow with interrupt disabled
2. 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.

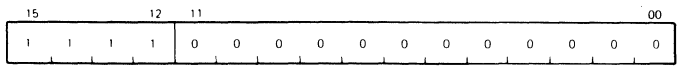
If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow and underflow are described above. If neither occurs, then for oppositely signed operands with an exponent difference of 0 or 1, the answer returned is exact if a loss of significance of one or more bits 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. 1 LSB in chopping mode with either single or double precision.
2. ½ LSB in rounding mode with either single (all FP11s and KEF11-AA) or double precision (for KEF11-AA); ¾ LSB (FP11-C) or 33/64 LSB (FP11-A and -F) in rounding mode with double precision.

Special Comment: The undefined variable −0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

CFCC
Copy Floating Condition Codes 170000

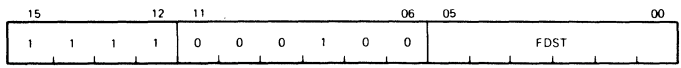


Format: CFCC

Operation:
C ← FC
V ← FV
Z ← FZ
N ← FN

Description: Copy the FPP condition codes into the CPU's condition codes.

CLRF
CLRD
Clear Floating/Double 1704 FDST



Format: CLRF FDST

Operation: (fdst) ← exact 0

Condition Codes:
FC ← 0
FV ← 0
FZ ← 1
FN ← 0

Description: Set (fdst) to 0. Set FZ condition code, clear other condition code bits.

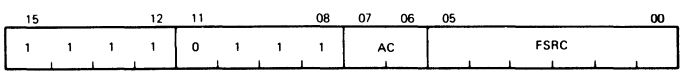
Interrupts: No interrupts will occur.

Overflow and underflow cannot occur.

Accuracy: The instructions are exact.

CMPF
CMPD

Compare Floating/Double 173(AC+4)FSRC



Format: CMPF FSRC,AC

Operation: (fsrc) ← AC

Condition Codes:
FC ← 0
FV ← 0
FZ ← 1 if (fsrc) = 0, else FZ ← 0
FN ← 1 if (fsrc) < 0, else FN ← 0

Description: Compare the contents of (fsrc) with the accumulator. Set the appropriate floating point condition codes. The accumulator and (fsrc) are left unchanged except as noted below.

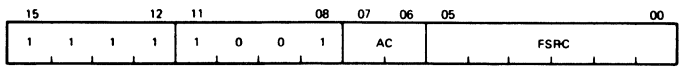
Interrupts: If FIUV is enabled, trap on −0 occurs before execution.

Accuracy: These instructions are exact.

Special Comment: An operand which has a biased exponent of 0 is treated as if it were an exact 0. In this case, where both operands are 0, the FPP will store an exact 0 in AC.

DIVF
DIVD

Divide Floating/Double 174(AC+4)FSRC



Format: DIVF FSRC,AC

Operation: If EXP[(fsrc)] = 0, AC ← AC and the 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.

If overflow occurs and FIV is not enabled, AC ← exact 0.

For all other cases, AC ← QUOT.

Condition Codes:
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 either operand has a biased exponent of 0, it is treated as an exact 0. For fsrc this would imply division by 0; 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 two guard bits for correct rounding. The quotient is rounded and chopped according to the values of the FD and FT bits in the FPS register. The result is stored in the AC except for:

1. Overflow with interrupt disabled
2. 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 (fsrc) = 0, interrupt traps on attempt to divide by 0.

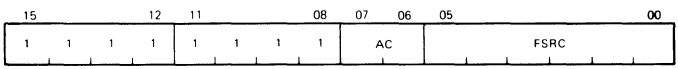
If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow and underflow are described above. If none of these occur, the error in the quotient will be bounded by 1 LSB in chopping mode and by ½ LSB in rounding mode.

Special Comment: The undefined variable −0 can occur only in conjunction with overflow and underflow. It will be stored in AC only if the corresponding interrupt is enabled.

LDCDF
LDCFD

Load and Convert from Double to Floating and from Floating to Double 177(AC+4)FSRC



Format: LDCDF FSRC,AC

Operation: 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 ← Cxy[(fsrc)], where Cxy specifies conversion from floating mode x to floating mode y.

x = D, y = F if FD = 0 (single) LDCDF
x = F, y = D if FD = 1 (double) LDCFD

Condition Codes:
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 into AC. The lower half of AC is cleared.

Interrupts: If FIUV is enabled, the trap on −0 occurs before execution. However, the condition codes will reflect a fetch of −0 regardless of the FIUV bit.

Overflow cannot occur for LDCFD.

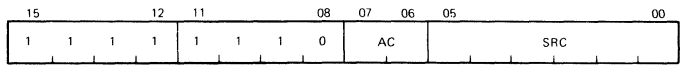
A trap occurs if FIV is enabled, and if rounding with LDCDF causes overflow. AC ← overflowed result. This result must be +0 or −0.

Underflow cannot occur.

Accuracy: LDCFD is an exact instruction. Except for overflow, described above, LDCDF incurs an error bounded by 1 LSB in chopping mode and by ½ LSB in rounding mode.

LDCIF LDCLF
LDCID LDCLD

Load and Convert Integer or Long Integer to Floating or Double Precision 177(AC)SRC



Format: LDCIF SRC,AC

Operation: AC ← Cjx[(src)], where Cjx specifies conversion from integer mode j to floating mode y.

j = I if FL = 0, j = L if FL = 1
x = F if FD = 0, x = D if FD = 1

Condition Codes:
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.

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

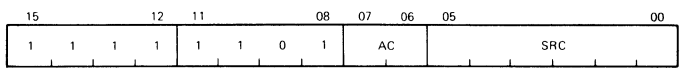
are left-justified and the remaining 16 bits loaded with 0s before conversion.

In the case of LDCLF, the fractional part of the floating point representation is chopped or rounded to 24 bits according to the state of FT (1 = chop, 0 = round).

Interrupts: None; (SRC) is not floating point, so trap on −0 cannot occur.

Accuracy: LDCIF, LDCID, and LDCLD are exact instructions. The error incurred by LDCLF is bounded by 1 LSB in chopping mode and by ½ LSB in rounding mode.

LDEXP
Load Exponent 176(AC+4)SRC



Format: LDEXP SRC,AR

Operation: If $-200_8 < (src) < 200_8$, EXP[AC] ← SRC $+200_8$ and the rest of AC is unchanged.

If (src) > 177_8 and FIV is enabled,
EXP[AC] ←[(src) + 200_8]<7:0> on the FP11-A, -F and KEF11-AA.

EXP[AC] ← (src)<6:0> on the FP11-C.

If (src) > 177_8 and FIV is disabled, AC ← exact 0.

If (src) < -177_8 and FIU is enabled,
EXP[AC] ← [(src) + 200_8] <7:0> on the FP11-A, -F and KEF11-AA.

EXP[AC] ← (src)<6:0> on the FP11-C.

If (src) < -177_8 and FIU is disabled, AC ← exact 0.

Condition Codes:
FC ← 0
FV ← 1 if (SRC) > 177_8, else FV ← 0
FZ ← 1 if (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_8 notation and insert it in the EXP field of AC. This is a meaningful operation only if ABS[(src)] ≤ 177_8.

If (src) > 177_8, the result is treated as overflow. If (src) < -177_8, the result is treated as underflow. Note that the KEF11-AA does not treat these abnormal conditions as the FP11-C do, but it does treat them as the FP11-A and FP11-F do.

Interrupts: No trap on −0 in AC occurs, even if FIUV is enabled.

If (src) > 177_8 and FIV is enabled, trap on overflow will occur.

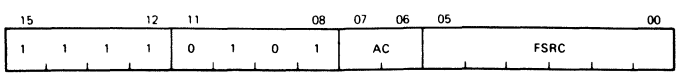
If (src) < -177_8 and FIU is enabled, trap on underflow will occur.

Accuracy: Errors due to overflow and underflow are described above. If EXP[AC] = 0 and (src) ≠ −200, AC changes from a floating point number treated as 0 by all floating arithmetic operations to a nonzero number. This is because the insertion of the "hidden" bit in the microcode implementation of arithmetic instructions is triggered by a nonvanishing value of EXP.

For all other cases, LDEXP implements exactly the transformation of a floating point number (2**K) * f into (2**(src)) * f where ½ ≤ ABS(f) < 1.

LDF
LDD

Load Floating/Double 172(AC+4)FSRC



Format: LDF FSRC,AC

Operation: AC ← (fsrc)

Condition Codes:
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 AC.

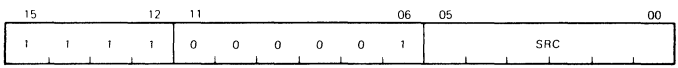
Interrupts: If FIUV is enabled, trap on −0 occurs before AC is loaded. However, the condition codes will reflect a fetch −0 regardless of the FIUV bit.

Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: These instructions permit use of −0 in a subsequent floating point instruction if FIUV is not enabled and (fsrc) = −0.

LDFPS
Load FPP Program Status 1701 SRC



Format: LDFPS SRC

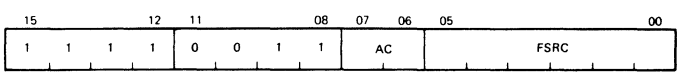
Operation: FPS ← (src)

Description: Load FPP status register from (src).

Special Comment: Bits 13, 12, and 4 should not be used for the user's own purposes, since these bits are not recoverable by the STFPS instruction. Bit 4 may be set in Kernel mode if the FPP implements maintenance mode.

MODF
MODD

Multiply and Separate Integer and Fraction Floating/Double 171(AC+4)FSRC



Format: MODF FSRC,AC

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 ≤ f < 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 ≤ 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 = 24 for floating mode and L = 56 for double mode.

1. If PROD overflows and FIV is enabled, ACv1 ← N, chopped to L bits, AC ← exact 0.

   Note that EXP[N] is too small by 400_8 and that −0 can get stored in ACv1.

   If FIV is not enabled, ACv1 ← exact 0, AC ← exact 0, and −0 will never be stored.

2. If 2**L ≤ ABS[PROD] and no overflow, ACv1 ← N, chopped to L bits, AC ← exact 0.

   The sign and EXP of N are correct, but low-order bit information 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 a 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 2**L because only 59 bits (64 bits for KEF11-AA) of PROD are generated.

If 2**p ≤ ABS[N] < 2**(p**1), with p > 2 (7 for KEF11-AA) the low-order p−2 (p−7 for KEF11-AA) bits of g may be in error.

4. If ABS[PROD] < 1 and no underflow, ACv1 ← exact 0 and 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 is enabled, ACv1 ← exact 0 and AC ← g.

   Errors are as in case 4, except that EXP[AC] will be too large by 400_8 (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).

Condition Codes:

FC ← 0
FV ← 1 if PROD overflows, else FV ← 0
FZ ← If AC = 0, else FZ ← 0
FN ← 1 if AC < 0, else FN ← 0

Interrupts:

If FIUV is enabled, trap on −0 in FSRC occurs 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 because the number of nonvanishing bits in the fractional part of PROD never exceeds L, and hence neither chopping nor rounding can introduce error.

2. To reduce the argument of a trigonometric function.

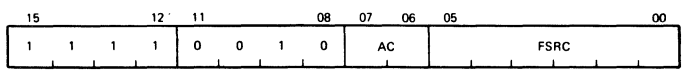
   ARG * 2/PI = N + g. The low two bits of N identify the quadrant, and g is the argument reduced to the first quadrant. The accuracy of N+g is limited to L bits because of the factor 2/PI. The accuracy of the reduced argument thus depends on the size of N.

3. To evaluate the exponential function e**x, obtain x * (log e base 2) = N + g, then e**x = (2**N) * (e**(g*ln 2)).

   The reduced argument is g * ln2 < 1 and the factor 2**N is an exact power of 2, which may be scaled in at the end via STEXP, ADD N to EXP and LDEXP. The accuracy of N + g is limited to L bits because of the factor (log e base 2). The accuracy of the reduced argument thus depends on the size of N.

MULF
MULD

Multiply Floating/Double — 171(AC)FSRC



Format: MULF FSRC,AC

Operation: 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.

Condition Codes:
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 0, (AC) ← exact 0. For all other cases, PROD is generated to 48 (32 for KEF11-AA) bits for floating mode and 59 (64 for KEF11-AA) bits for double mode. The product is rounded or chopped according to the value of the FT bit, and is stored in AC except for:

1. Overflow with interrupt disabled
2. 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.

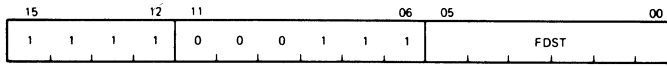
If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow and underflow are described above. If neither occurs, 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 the corresponding interrupt is enabled.

NEGF
NEGD

Negate Floating/Double 1707 FDST



Format: NEGF −(fdst)

Operation: (fdst) ← −(fdst) if EXP [(fdst)] ≠ 0, else (fdst) ← exact 0.

Condition Codes:
FC ← 0
FV ← 0
FZ ← 1 if (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).

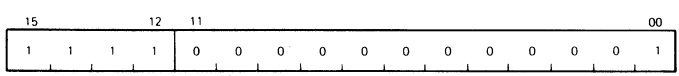
Interrupts: If FIUV is enabled, trap on −0 occurs after execution.

Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: If a −0 is present in memory and the FIUV bit is enabled, then the FPP stores an exact 0 in memory. The condition codes reflect an exact 0 (FZ ← 1).

SETF
Set Floating Mode 170001

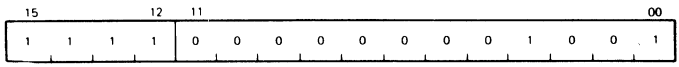


Format: SETF

Operation: FD ← 0

Description: Set the FPP to single-precision mode.

SETD
Set Floating Double Mode 170011

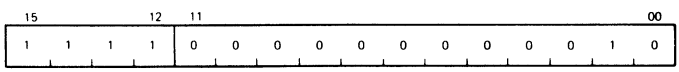


Format: SETD

Operation: FD ← 1

Description: Set the FPP to double-precision mode.

SETI
Set Integer Mode 177002

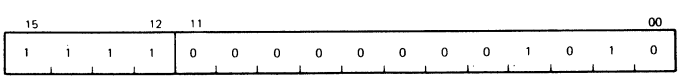


Format: SETI

Operation: FL ← 0

Description: Set the FPP for short integer data.

SETL
Set Long Integer Mode 177012



Format: SETL

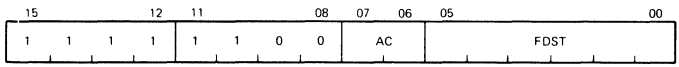
Operation: FL ← 1

Description: Set the FPP for long integer data.

STCFD
STCDF

Store and Convert from Floating to Double and from Double to Floating

176(AC)FDST



Format: STCFD AC,FDST

Operation: If AC = 0, (fdst) ← exact 0.

If FD = 1, FT = 0, FIV = 0 and rounding causes overflow, (fdst) ← exact 0.

In all other cases, (fdst) ← Cxy[AC], where Cxy specifies conversion from floating mode x to floating mode y.

x = F, y = D if FD = 0 (single) STCFD
x = D, y = F if FD = 1 (double) STCDF

Condition Codes:
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.

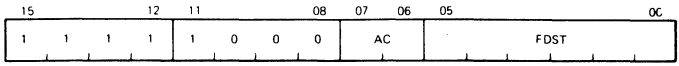
Overflow cannot occur for STCFD.

A trap occurs if FIV is enabled, and if rounding with STCDF causes overflow. (FDST) ← overflowed result. This 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 by ½ LSB in rounding mode.

STF
STD

Store Floating/Double 174(AC)FDST



Format: STF AC,FDST

Operation: (fdst) ← AC

Condition Codes:
FC ← FC
FV ← FV
FZ ← FZ
FN ← FN

Description: Store single- or double-precision number from AC.

Interrupts: These instructions do not interrupt if FIUV is enabled, because the −0, if present, is in AC, not in memory.

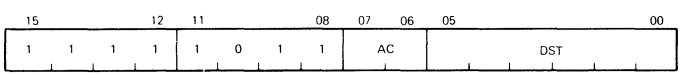
Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: These instructions permit storage of a −0 in memory from AC. There are two conditions in which −0 can be stored in AC of the FPP. One occurs when underflow or overflow is present and the corresponding interrupt is enabled. A second occurs when an LDF, LDD, LDCDF, or LDCFD instruction is executed and the FIUV bit is disabled.

STCFI STCDI
STCFL STCDL

Store and Convert from Floating or Double to Integer or Long Integer 175(AC+4)DST



Format: STCFI AC,DST

Operation: (dst) ← Cxj[AC] if −JL−1 < Cxj[AC] < JL+1, else

(dst) ← 0, where Cjx 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

Condition Codes:

C, FC ← 0 if −JL−1 < Cxj[AC] < JL+1, else C, FC ←1
V, FV ← 0
Z, FZ ← 1 if (dst) = 0, else Z, FZ ← 0
N, FN ← 1 if (dst) < 0, else N, 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 addressing 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 FPS register.

Interrupts:

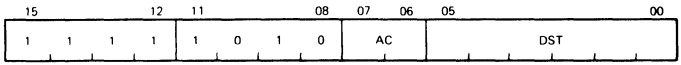
These instructions do not interrupt if FIUV is enabled, because the −0, if present, is in AC, not in memory.

If FIC is enabled, trap on conversion failure will occur.

Special Comment:

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.

STEXP
Store Exponent 175(AC)DST



Format: STEXP AC,DST

Operation: (dst) ← EXP[AC] -200_8

Condition Codes:
C, FC ← 0
V, FV ← 0
Z, FZ ← 1 if (dst) = 0, else Z, FZ ← 0
N, FN ← 1 if (dst) < 0, else N, FN ← 0

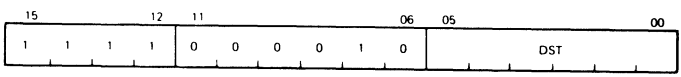
Description: Convert AC's exponent from excess 200_8 notation to 2's complement and store the result in dst.

Interrupts: This instruction will not trap on −0.

Overflow and underflow cannot occur.

Accuracy: This instruction is exact.

STFPS
Store FPP Program Status 1702 DST



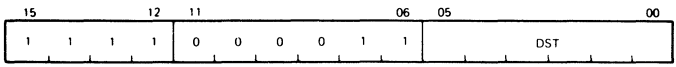
Format: STFPS DST

Operation: (dst) ← FPS

Description: Store FPP's status register in dst.

Special Comment: Bits 13, 12, and 4 (if Maintenance Mode is not implemented) are loaded with 0. All other bits are the corresponding bits in the FPSs.

STST
Store FPP Status 1703 DST



Format: STST DST

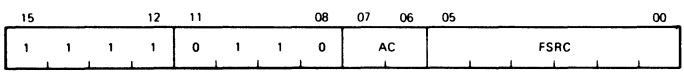
Operation: (dst) ← FEC
(dst + 2) ← FEA

Description: Store the FEC and FEA in dst and dst+2.

NOTE

1. If the 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 instruction caused a floating point exception.

SUBF
SUBD
Subtract Floating/Double 173(AC)FSRC



Format: SUBF FSRC,AC

Operation: 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 cases, AC ← DIFF.

Condition Codes: FC ← 0
FV ← 1 if overflow occurs, else FV ← 0

FZ ← 1 if AC = 0, else FZ ← 0
FN ← 1 if AC < 0, else FN ← 0

Description: Subtract the contents of fsrc from the contents of AC. The subtraction is carried out in single or double precision and is rounded or chopped according to the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

1. Overflow with interrupt disabled
2. 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.

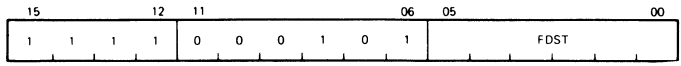
If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow and underflow are described above. If neither occurs, then for like signed operands with an exponent difference of 0 or 1, the answer returned is exact if a loss of significance of one or more bits 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. 1 LSB in chopping mode with either single or double precision
2. ½ LSB in rounding mode with either single (all FP-11s and KEF11-AA) or double precision (KEF11-AA only); ¾ LSB (FP11-C) and 33/64 LSB (FP11-A and FP11-F) in rounding mode with double precision.

Special Comment: The undefined variable −0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

TSTF
TSTD
Test Floating/Double 1705 FDST



Format:	TSTF FDST
Operation:	(fdst)
Condition Codes:	FC ← 0 FV ← 0 FZ ← 1 if (fdst) = 0, else FZ ← 0 FN ← 1 if (fdst) < 0, else FN ← 0
Description:	Set the FP11 condition codes according to the contents of fdst.
Interrupts:	If FIUV is set, trap on −0 occurs after execution. Overflow and underflow cannot occur.
Accuracy:	These instructions are exact.

CHAPTER 12

COMMERCIAL INSTRUCTION SET

Commercial Instruction Set

The PDP-11 Commercial Instruction set (CIS11) is applicable to both the PDP-11/44 and PDP-11/24 processors, and consists of the following extended instruction groups:

07602X	Commercial Load 2 Descriptors
07603X	Character String Move
07604X	Character String Search
07605X	Numeric String
07606X	Commercial Load 3 Descriptors
07607X	Packed String
07613X	Character String Move (in-line)
07614X	Character String Search (in-line)
07615X	Numeric String (in-line)
07617X	Packed String (in-line)

These include instructions which operate on character strings and on decimal numbers. Each generic type of instruction is provided in two forms. The essential difference between the two forms is the manner in which operands are delivered to the instruction. The first form is the "register" form, where operands are implicitly obtained from the general registers. The second form is the "in-line" form, where operands or word address pointers to operands follow the opcode word in the instruction stream. The mnemonic for the in-line form is the mnemonic for the register form suffixed with the letter "I." The condition codes are set identically for both forms. The in-line forms minimize register modification.

Instructions are also provided which efficiently load operands into the general registers.

UNPREDICTABLE Conditions

"UNPREDICTABLE" means that the outcome is indeterminate and nonrepeatable. Either the result of an instruction or the effect of an instruction can be UNPREDICTABLE. When the results of an instruction are UNPREDICTABLE, the condition codes and destination operands (but not their descriptors) will contain UNPREDICTABLE values; destinations may not even contain valid results. When the effect of an instruction is UNPREDICTABLE, the entire user or process state, and not only the portion typically used by the instruction, will be UNPREDICTABLE. In a machine with multiple modes and address spaces, an

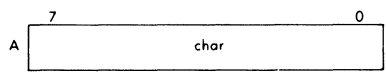
UNPREDICTABLE operation in a less privileged mode will not affect the state of a more privileged mode, nor will it result in accesses to memory from user mode which are outside the mapped limits of the user's program.

Note that architectural constraints exist on UNPREDICTABLE effects. In particular, an UNPREDICTABLE effect which manifests itself as a trap must meet all the requirements for the particular trap.

Character Data Types

There are three different character data types. The "character" is a single byte, and is an abbreviated string of length 1. The "character string" is a contiguous group of bytes in memory. The third is a "character set."

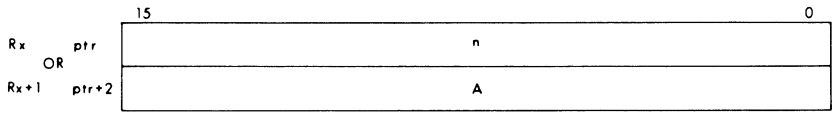
The character is an 8-bit byte:



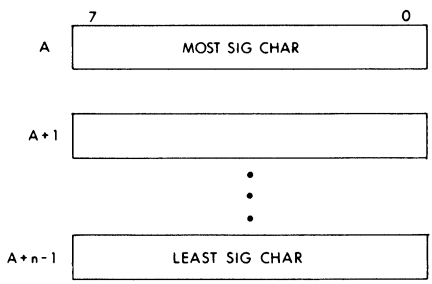
The character is used as an operand by CIS11 instructions. When it appears in a general register, the character is in the low-order half; the high-order half of the register must be zero. When it appears in the instruction stream, the character is in the low-order half of a word; the high-order half of the word must be zero. If the high-order half of a word which contains a character is nonzero, the effect of the instruction which uses it will be UNPREDICTABLE.

A character string is a contiguous sequence of bytes in memory that begins and ends on a byte boundary. It is addressed by its most significant character (lowest address). The highest address is the least significant character. It is specified by a two-word descriptor with the attributes of length and lowest address. The length is an unsigned binary integer which represents the number of characters in the string and may range from 0 to 65,535. A character string with zero length is said to be vacant: its address is ignored. A character string with nonzero length is said to be occupied.

The character string descriptor is used as an operand by CIS11 instructions. It appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. The following figure shows the descriptor for a character string of length "n" starting at address "A" in memory:



The following figure shows the character string in memory:

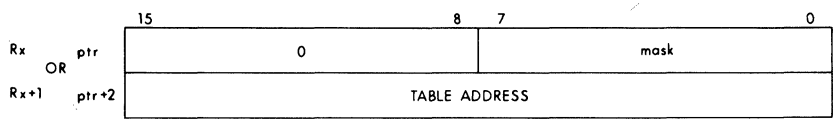


A "character set" is a subset of the 256 possible characters that can be encoded in a byte. It is specified by a descriptor which consists of the address of a 256-byte table and an 8-bit mask. The address is of the zeroeth byte in the table. Each byte in the table specifies up to eight orthogonal character subsets of which the corresponding character is a member. The mask selects which combinations of these orthogonal subsets constitute the entire character set. In effect, each bit in the mask corresponds to one of eight orthogonal subsets that may be encoded by the table. The mask specifies the union of the selected subsets into the character set. Typical sets would be: uppercase, lowercase, nonzero digits, end of line, etc.

Operationally, a character (char) is considered to be in the character set if the evaluation of (M[table.adr+char] AND mask) is not equal to zero. The character is not in the character set if the evaluation is zero. Each byte in the table indicates which combination of up to eight orthogonal character subsets (i.e., one for each of the eight bit vectors 00000001_2 00000010_2 00000100_2 00001000_2 00010000_2 00100000_2 01000000_2 and 10000000_2) the corresponding character is a member. The mask specifies which union of the eight orthogonal character subsets constitute the total character set. For example, if the eight-bit vector 00000001_2 appearing in the table corresponds to the character subset of all uppercase alphabetic characters, 00000010_2 appearing in

the table corresponds to the character subset of all lowercase alphabetic characters, and 00000100_2 appearing in the table corresponds to the decimal digits, then using the mask 00000011_2 with this table specifies the character set of all alphabetic characters, and using the mask 00000111_2 specifies the character set of all alphanumeric characters.

The character set descriptor is used as an operand by CIS11 intructions. It appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. If the high-order half of the first descriptor word is nonzero, the effect of an instruction which uses a character set will be UNPREDICTABLE.



Character String Instructions

The character string operations conveniently provide most of the common, as well as time-consuming, functions that are encountered in commercial data and text processing applications.

Instructions are provided to move and to search character strings.

Character String Move Instructions

MOVC(I)	Move character
MOVRC(I)	Move reverse justified character
MOVTC(I)	Move translated character

Character String Search Instructions

LOCC(I)	Locate character
SKPC(I)	Skip character
SCANC(I)	Scan character
SPANC(I)	Span character
CMPC(I)	Compare character
MATC(I)	Match character

The character string move instructions use character string descriptors as operands. These descriptors specify a source and a destination character string. The contents of the source are moved to the destination with alignment at either the most significant character as in MOVC(I) and MOVTC(I), or the least signficant character as in MOVRC(I). If the source is longer than the destination, characters are truncated from the side opposite that of the alignment; if the destination is longer than the source, the destination is completed with fill characters on the side opposite that of the alignment. The MOVTC(I) instructions move a translated source string to a destination string.

The character string search instructions use a character string descriptor as one operand. The other operand is either a character, a character string descriptor, or a character set descriptor. These instructions are used to examine the source string to find the presence or absence of characters. The source string is processed from most significant to least significant character.

Conceptually, these instructions may be divided into three classes:

1. Character String Searches — CMPC(I) compares two character strings. The condition codes are set according to the comparison of the corresponding most significant unequal characters. MATC(I) finds an object string within a source string. This is the "instring" function that languages and text processing systems provide.
2. Character Searches — LOCC(I) finds the first occurrence of a given character in a string. SKPC(I) skips to the first nonoccurrence of a given character in a string.
3. Character Set Searches — In these instructions, a string is examined until a member of a character set is either found as a SCANC(I), or not found as in SPANC(I). This aids the search for one of several delimiters such as "/", ",", CR, LF, FF, etc., or the passing of combinations of characters such as blanks, tabs, etc. LOCC(I) and SKPC(I) are optimizations of SCANC(I) and SPANC(I) in which the set consists of a single character.

The setting of condition codes reflects the results of the character string operations. For character string moves, the condition codes indicate whether the source and destination strings were of equal length, the source was shorter than the destination such that fill characters were used, or the source was longer than the destination such that characters were truncated. This is accomplished by setting the condition codes on the result of arithmetically comparing the initial source and destination lengths. For CMPC(I), the condition codes are the result of arithmetically comparing the most significant correspond-

ing pair of unequal characters. For the other search instructions, they show whether or not the operand strings were completely examined.

The condition codes for some character string search instructions may be interpreted according to the notion of success or failure. Success is the accomplishment of the instruction's task; failure is the inability to accomplish the task. Since the condition codes are set based on the results of the instruction, there is an indirect correspondence between these settings and success or failure. This correspondence is invariant within an instruction, but it is not the same for all search instructions. Therefore, different branch instructions must be used to test the operation of each instruction. They are summarized in the following table:

Instruction	Success	Failure
LOCC(I)	BNE	BEQ
SCANC(I)	BNE	BEQ
CMPC(I)	BEQ	BNE
MATC(I)	BNE	BEQ

The "register form" of character string instructions implicitly finds operands in the general registers. These operands include character, character string descriptor, character set descriptor, and translation table address. If an instruction does not use a register, its contents will be undisturbed. R0-R1 generally contain a source character string descriptor; R2-R3 generally contain a second source character string descriptor, or the destination string descriptor. The low-order half of R4 is used as an explicit character. R4-R5 is used to contain a character set descriptor. R5 contains the starting address of a 256-byte table which is used for character translation.

When move instructions terminate, R0 contains the number of unmoved source characters, and R1, R2, and R3 are cleared. For search instructions, the registers are updated to represent descriptors for the resulting strings.

The "in-line form" of character string instructions finds operands, or pointers to operands, in the instruction stream immediately following the opcode word. Operands which appear directly in the instruction stream include characters and translation table addresses. Descriptors are represented in the instruction stream by a single word whose contents are interpreted as a word address pointer to the two-word descriptor. These descriptors specify character strings and character sets. Some instructions return a character string descriptor in R0-R1.

In general, all character string instructions are unaffected by the overlapping of source or destination strings. The result of the move instructions is equivalent to having read the entire source string before

storing characters in the destination. If the destination string of the MOVTC(I) instructions overlaps the translation table, the characters stored in the destination string will be UNPREDICTABLE.

Decimal String Data Types

Two classes of decimal string data types—numeric strings and packed strings—are defined. Both have similar arithmetic and operational properties; they primarily differ in the representation of signs and the placement of digits in memory.

The numeric string data types are signed zoned, unsigned zoned, trailing overpunch, leading overpunched, trailing separate and leading separate. The packed string data types are signed packed and unsigned packed. Instructions which operate on numeric strings permit each numeric string operand to be separately specified; similarly, packed string instructions permit each packed string operand to be separately specified. Thus, within each of the two classes of decimal strings, the operands of an instruction may be of any data type within the appropriate class.

Decimal strings exist in memory as contiguous bytes which begin and end on a byte boundary. They represent numbers consisting of 0 to 31_{10} digits, in either sign-magnitude or absolute-value form. Sign-magnitude strings (SIGNED) may be positive or negative; absolute-value strings (UNSIGNED) represent the absolute value of the magnitude. Decimal numbers are whole integer values with an implied decimal radix point immediately beyond the least significant digit; they may be conceptually extended with zero digits beyond the most significant digit.

A four-bit binary coded decimal representation is used for most digits in decimal strings. A four-bit half byte is called a "nibble" and may be used to contain a binary bit pattern which represents the value of a decimal digit. The following table shows the binary nibble contents associated with each decimal digit:

digit	nibble	digit	nibble
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Each decimal string data type may have several representations. These representations permit a certain latitude when accepting source operands. Decimal string data types have a PREFERRED representation, which is a valid source representation and which is used

to construct the destination string. Additional ALTERNATE representations are provided for some decimal data types when accepting source operands.

Decimal strings used as source operands will not be checked for validity. Instructions will produce UNPREDICTABLE results if a decimal string used as a source operand contains an invalid digit encoding, invalid sign designator, or, in the case of overpunched numbers, an invalid sign/digit encoding.

When used as a source, decimal strings with zero magnitude are unique, regardless of sign. Thus, both positive and negative zero have identical interpretations.

Conceptually, decimal string instructions first determine the correct result, and then store the decimal string representation of the correct result in the destination string. A result of zero magnitude is considered to be positively signed. If the destination string can contain more digits than are significant in the result, the excess most significant destination string digits have zero digits stored in them. If the destination string cannot contain all significant digits of the result, the excess most significant result digits are not stored; the instruction will indicate decimal overflow. Note that negative zero is stored in the destination string as a side effect of decimal overflow where the sign of the result is negative and the destination is not large enough to contain any nonzero digits of the result.

If the destination string has zero length, no resulting digits will be stored. The sign of the result will be stored in separate and packed strings, but not in zoned and overpunched strings. Decimal overflow will indicate a nonzero result.

Decimal String Descriptors

Decimal strings are represented by a two-word descriptor. The descriptor contains the length, data type, and address of the string. It appears in two consecutive general registers (register form of instructions), or in two consecutive words in memory pointed to by a word in the instruction stream (in-line form of instructions). The unused bits are reserved by the architecture and must be 0. The effect of an instruction using a descriptor will be unpredictable if any nonzero reserved field in the descriptor contains nonzero values or a reserved data type encoding is used. The design of the numeric and packed string descriptors are identical:

First Word

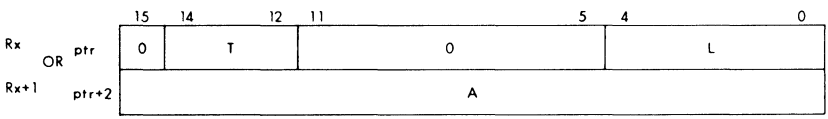
length <4:0>	Number of digits specified as an unsigned binary integer

data type <14:12>	Specifies which decimal data type representation is used

Second Word

address <15:0>	Specifies the address of the byte which contains the most significant digit of the decimal string

The following figure shows the descriptor for a decimal string of data type "T" whose length is "L" digits and whose most significant digit is at address "A":



The encodings (in binary) for the NUMERIC string data type field are:

000	signed zoned
001	unsigned zoned
010	trailing overpunch
011	leading overpunch
100	trailing separate
101	leading separate
110	—reserved to DIGITAL
111	—reserved to DIGITAL

The encodings (in binary) for the PACKED string data type field are:

000	—reserved to DIGITAL
001	—reserved to DIGITAL
010	—reserved to DIGITAL
011	—reserved to DIGITAL
100	—reserved to DIGITAL
101	—reserved to DIGITAL
110	signed packed
111	unsigned packed

Packed Strings

Packed strings can store two decimal digits in each byte. The least significant (highest addressed) byte contains the sign of the number in bits <3:0> and the least significant digit in bits <7:4>.

Signed Packed Strings — The preferred positive sign designator is 1100_2; alternate positive sign designators are 1010_2, 1110_2 and 1111_2. The preferrred negative sign designator is 1101_2; the alternate negative sign designator is 1011_2. Source strings will properly accept both the preferred and alternate designators; destination strings will be stored with the preferred designator.

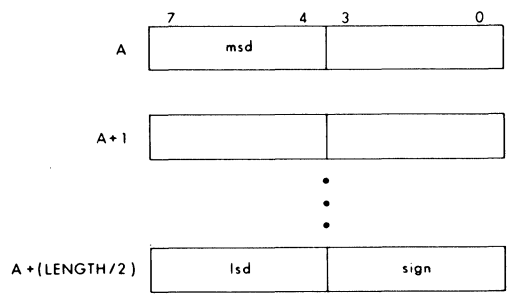
Unsigned Packed Strings — The unsigned sign designator is 1111_2.

PACKED SIGN NIBBLE:

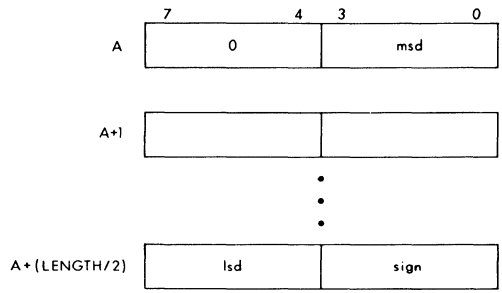
sign nibble	**preferred designator**	**alternate designators**
positive	1100_2	1010_2, 1110_2, 1111_2
negative	1101_2	1011_2
unsigned	1111_2	

For other than the least significant byte, bytes contain two consecutive digits—the one of lower significance in bits <3:0> and the one of higher significance in bits <7:4>. For numbers whose length is odd, the most significant digit is in bits <7:4> of the lowest addressed bytes. Numbers with an even length have their most significant digit in bits <3:0> of the lowest addressed byte; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000_2 for destination strings. Numbers with a length of one occupy a single byte and contain their digit in bits <7:4>. The number of bytes which represent a packed string is [length/2]+1 (integer division where the fractional portion of the quotient is discarded).

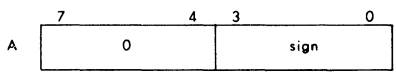
The following is a packed string with an odd number of digits:



The following is a packed string with an even number of digits:



A zero length packed string occupies a single byte of storage; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000_2 for destination strings. Bits <3:0> must be a valid sign for source strings, and are used to store the sign of the result for destination strings. When used as a source, zero length strings represent operands with zero magnitude. When used as a destination, they can only reflect a result of zero magnitude without indicating overflow. The following is a zero length packed string:



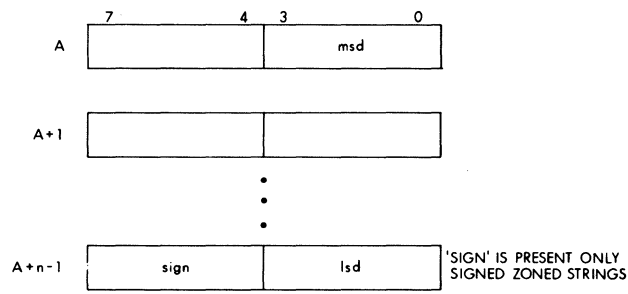
A valid packed string is characterized by:

1. A length from 0 to 31_{10} digits.
2. Every digit nibble is in the range 0000_2 to 1001_2.
3. For even length sources, bits <7:4> of the lowest addressed byte are 0000_2.
4. Signed packed strings—sign nibble is either 1010_2, 1011_2, 1100_2, 1101_2, 1110_2 or 1111_2.
5. Unsigned packed strings—sign nibble is 1111_2.

Zoned Strings

Zoned strings represent one decimal digit in each byte. Each byte is divided into two portions—the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low-order nibble contains the value of the corresponding decimal digit.

Signed Zoned Strings — When used as a source string, the high-order nibble of the least significant byte contains the sign of the num-



ber; the high-order nibbles of all other bytes are ignored. Destination strings are stored with the sign in the high-order nibble of the least significant byte, and 0011_2 in the high-order nibble of all other bytes. 0011_2 in the high-order nibble corresponds to the ASCII encoding for numeric digits. The positive sign designator is 0011_2; the negative sign designator is 0111_2.

Unsigned Zoned Strings — When used as a source string, the high-order nibbles of all bytes are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all bytes.

The number of bytes needed to contain a zoned string is identical to the length of the decimal number.

A zero length zoned string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a nonzero result will be indicated by setting overflow.

A valid zoned string is characterized by:

1. A length from 0 to 31_{10} digits.
2. The low-order nibbles of each byte are in the range 0000_2 to 1001_2.
3. Signed zoned strings—The high order nibble of the least significant byte is either 0011_2 or 0111_2.

Overpunch Strings

Overpunch strings represent one decimal digit in each byte. Trailing overpunch strings combine the encoding of the sign and the least significant digit; leading overpunch strings combine the encoding of the sign and the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions—the high-

order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low-order nibble contains the value of the corresponding decimal digit. When used as a source string, the high-order nibble of all bytes which do not contain the sign are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all bytes which do not contain the sign. 0011_2 in the high-order nibble corresponds to the ASCII encoding for numeric digits.

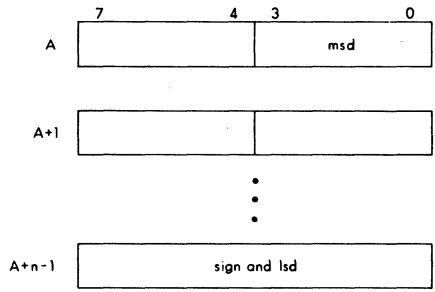
The following table shows the sign of the decimal string and the value of the digit which is encoded in the sign byte. Source strings will properly accept both the preferred and alternate designators; destination strings will store the preferred designator. The preferred designators correspond to the ASCII graphics "A" to "R," "{," and "}." The alternate designators correspond to the ASCII graphics "0" to "9," "[," "?," "]," "!" and ":".

OVERPUNCH SIGN/DIGIT BYTE:

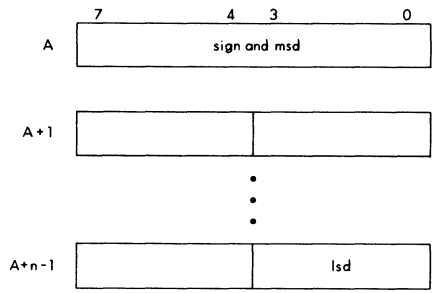
overpunch sign/digit	**preferred designator**	**alternate designators**
+0	01111011_2	00110000_2, 01011011_2, 00111111_2
+1	01000001_2	00110001_2
+2	01000010_2	00110010_2
+3	01000011_2	00110011_2
+4	01000100_2	00110100_2
+5	01000101_2	00110101_2
+6	01000110_2	00110110_2
+7	01000111_2	00110111_2
+8	01001000_2	00111000_2
+9	01001001_2	00111001_2
−0	01111101_2	01011101_2, 00100001_2, 00111010_2
−1	01001010_2	
−2	01001011_2	
−3	01001100_2	
−4	01001101_2	
−5	01001110_2	
−6	01001111_2	
−7	01010000_2	
−8	01010001_2	
−9	01010010_2	

The number of bytes needed to contain an overpunch string is identical to the length of the decimal number.

The following is a trailing overpunch string:



The following is a leading overpunch string:



A zero length overpunch string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a nonzero result will be indicated by setting overflow.

A valid overpunch string is characterized by:

1. A length from 0 to 31_{10} digits.
2. The low-order nibble of each digit byte is in the range 0000_2 to 1001_2.
3. The encoded sign/digit byte contains values from the above table of preferred and alternate overpunch sign/digit values.

Separate Strings

Separate strings represent one decimal digit in each byte. Trailing separate strings encode the sign in a byte immediately beyond the least significant digit; leading separate strings encode the sign in a

byte immediately beyond the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions—the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low order nibble contains the value of the corresponding decimal digit.

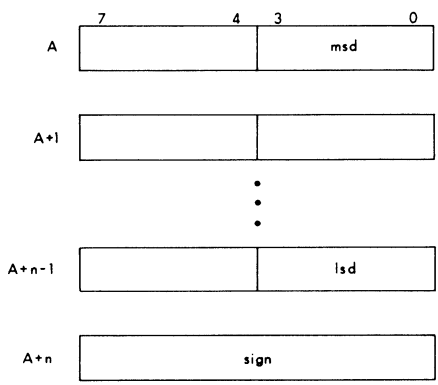
When used as a source string, the high-order nibbles of all digit bytes are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all digit bytes. 0011_2 in the high-order nibble corresponds to the ASCII encoding for numeric digits. The preferred positive sign designator is 00101011_2 and the alternate positive sign designator is 00100000_2. The negative sign designator is 00101101_2. These designators correspond to the ASCII encoding for "+," "space," and "−."

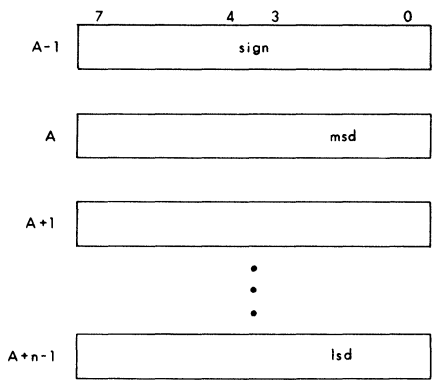
SEPARATE SIGN BYTE:

sign byte	**preferred designator**	**alternate designator**
positive	00101011_2	00100000_2
negative	00101101_2	

The number of bytes needed to contain a leading or trailing separate string is identical to (length+1).

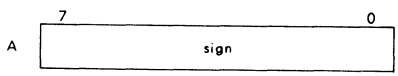
The following is a trailing separate string:



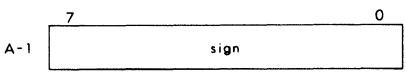


A zero length separate string occupies a single byte of memory which contains the sign. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only reflect a result of zero magnitude without indicating overflow; the sign of the result is stored.

The following is a zero length trailing separate string:



The following is a zero length leading separate string:



A valid separate string is characterized by:

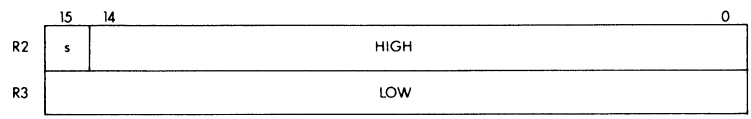
1. A length from 0 to 31_{10} digits.
2. The low-order nibble of each digit byte is in the range 0000_2 to 1001_2.
3. The sign byte is either 00100000_2, 00101011_2 or 00101101_2.

Long Integer

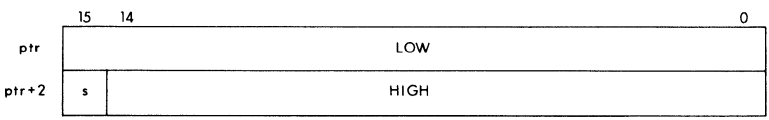
Long integers are 32-bit binary 2's complement numbers organized as two words in consecutive registers or in memory—no descriptor is

used. One word contains the high-order 15 bits. The sign is in bit <15>; bit <14> is the most significant. The other word contains the low-order 16 bits with bit <0> the least significant. The range of numbers that can be represented is −2,147,483,648 to +2,147,483,647.

The register form of decimal convert instructions uses a restricted form of long integer with the number in the general register pair R2-R3:



The in-line form of decimal convert instructions reference the long integer by a word address pointer which is part of the instruction stream:



Note that these two representations of long integers differ. There is no single representation of long integer among EAE, EIS, FPP and software. The "register form" was selected to be compatible with EIS; the "in-line form" was selected to be compatible with current standard software usage.

Decimal String Instructions

The decimal string instruction groups aid manipulation of decimal data. Several numeric (byte) and packed decimal data types are supported. Instructions are provided for basic arithmetic operations, as well as for compare, shift, and convert functions.

Instructions

Each arithmetic, shift and compare instruction operates on a single class of data type. Both numeric and packed string instructions are provided for most operations. Convert instructions have a source operand of one data type and a destination operand of another data type. Decimal string instructions specify to which class each of their decimal string operands belong. The data type supplied as part of each oper-

and's descriptor may be any valid data type of the class. This permits a general mixing of data types within numeric and packed classes.

The data types on which an instruction operates are designated by the last letter(s) of the opcode mnemonic. "N" denotes numeric strings, "P" denotes packed strings, and "L" denotes long binary integers.

The arithmetic instructions are ADDN(I), ADDP(I), SUBN(I), SUBP(I), MULP(I) and DIVP(I). ASHN(I) and ASHP(I) shift a decimal string by a specified number of digit positions (either direction) with optional rounding, and store the result in the destination string. Thus, they effectively multiply or divide by a power of ten. If the shift count is zero, these shift instructions can be used simply to move decimal strings (destinations are stored with preferred representation). Move negated may be accomplished by using SUBN(I) or SUBP(I). Arithmetic comparison instructions, CMPN(I) and CMPP(I), are provided to examine the relative difference between two decimal strings.

CVTNL(I) and CVTPL(I) convert a decimal string to a long (32-bit) 2's complement integer. CVTLN(I) and CVTLP(I) convert a long integer to a decimal string. CVTNP(I) and CVTPN(I) convert between numeric and packed decimal strings.

The instructions are:

Numeric String Instructions

ADDN(I)	Add numeric
SUBN(I)	Subtract numeric
ASHN(I)	Arithmetic shift numeric
CMPN(I)	Compare numeric

Packed String Instructions

ADDP(I)	Add packed
SUBP(I)	Subtract packed
MULP(I)	Multiply packed
DIVP(I)	Divide packed
ASHP(I)	Arithmetic shift packed
CMPP(I)	Compare packed

Convert Instructions

CVTNL	Convert numeric to long
CVTLN	Convert long to numeric
CVTPL	Convert packed to long
CVTLP	Convert long to packed
CVTNP	Convert numeric to packed
CVTPN	Convert packed to numeric

Condition Codes

For instructions which store a value in a destination string, the N and Z bits reflect the value stored. The N bit indicates a negative destination; the Z bit indicates a destination having zero magnitude. A destination string with zero magnitude is considered to be positive (even if a negative zero was stored as a consequence of decimal overflow). Thus, the setting of N and Z are mutually exclusive.

The V bit will indicate whether the destination string accurately represents the result of the instruction. It is also set if division by zero was attempted. If the V bit is set, the destination string will represent the least significant portion of the result (truncated). If the V bit is cleared, the destination represents the true result.

For DIVP(I), C indicates division by zero. Otherwise, C is always cleared.

For comparisions using the CMPN(I) and CMPP(I) instructions, the N and Z bits reflect the signed relationship between the source strings. The signed branch instructions can test the result. V and C are cleared.

For instructions which return a long integer value, N reflects the sign of the 2's complement integer, and Z indicates whether it was zero. V indicates whether the long integer could not contain all significant digits and sign of the result. CVTNL(I) and CVTPL(I) also use C to represent a borrow from a more significant portion of the long binary result. Otherwise, C is cleared.

Operand Delivery

The "register form" of decimal string instructions implicitly finds the operands in the general registers. These operands include decimal string descriptors, long binary integers, and shift descriptor words. If an instruction does not use a register, its contents will be undisturbed. R0-R1 generally contain the first source descriptor, R2-R3 generally contain the second source descriptor, and R4-R5 generally contain the destination descriptor. ASHN and ASHP use R4 to contain a shift descriptor word. CVTLN, CVTLP, CVTNL and CVTPL use R0-R1 to contain a decimal string descriptor, and R2-R3 for the long integer. When an instruction is completed, the source descriptor registers are cleared.

The "in-line form" of decimal string instructions finds the operands, or pointers to descriptors, in the instruction stream immediately following the opcode word. Operands which appear directly in the instruction stream are shift descriptor words. Operands which are represented in the instruction stream by a pointer containing the word address of the

descriptor are decimal string descriptors and long binary integers. No in-line form of decimal string instructions modify R0-R6.

Data Overlap

The operation of decimal string instructions is unaffected by any overlap of the source operands provided that each source operand is a valid representation of the specified data type.

The overlap of the destination string and any of the source strings will, in general, produce UNPREDICTABLE results. However, ADDN(I), ADDP(I), SUBN(I) and SUBP(I) will permit the destination string to overlap either or both source strings only if all corresponding digits of the strings are in coincident bytes in memory. This facilitates two-address arithmetic.

Commercial Load Descriptor Instructions

The commercial load descriptor instructions augment the character and decimal string instructions by efficiently loading the general registers with string descriptors. Two forms of instructions are provided. The L2Dr instructions load two string descriptors into the general registers. The first descriptor is loaded into R0-R1 and the second descriptor is loaded into R2-R3. This instruction supports equal length character string move, equal length character string compare, character string matching, and decimal string compare.

The second form, the L3Dr instructions, take three descriptors. The first is loaded into R0-R1, the second into R2-R3, and the third into R4-R5. The instruction supports three-address arithmetic.

The condition codes are not affected.

Words containing the addresses of the descriptors (two for L2Dr and three for L3Dr) are in consecutive locations in memory. The descriptor addresses are found by applying the addressing mode @(Rr)+ once for each descriptor. The value of r is encoded as the low order three bits of the instruction's opcode. If $0 \leq r \leq 5$, then r can be thought of as the base address of a small table in memory, where each entry in the table contains the address of a descriptor. If $r = 6$, then the instructions effectively pop the addresses of descriptors off of the stack. If $r = 7$, then the descriptor addresses are contiguous with the instruction's opcode word.

The string descriptors are two words long. The address of the descriptor is that of the low-order word. It is loaded into the corresponding even register. The high-order word of the descriptor is loaded into the corresponding odd register. Note that although these instructions are described in terms of string descriptors, they are applicable for other

instances where two consecutive words in memory referenced by a pointer are to be copied into even-odd general register pairs.

The instructions are:

L2D0	Load 2 descriptors using @(R0)+
L2D1	Load 2 descriptors using @(R1)+
L2D2	Load 2 descriptors using @(R2)+
L2D3	Load 2 descriptors using @(R3)+
L2D4	Load 2 descriptors using @(R4)+
L2D5	Load 2 descriptors using @(R5)+
L2D6	Load 2 descriptors using @(R6)+
L2D7	Load 2 descriptors using @(R7)+

L3D0	Load 3 descriptors using @(R0)+
L3D1	Load 3 descriptors using @(R1)+
L3D2	Load 3 descriptors using @(R2)+
L3D3	Load 3 descriptors using @(R3)+
L3D4	Load 3 descriptors using @(R4)+
L3D5	Load 3 descriptors using @(R5)+
L3D6	Load 3 descriptors using @(R6)+
L3D7	Load 3 descriptors using @(R7)+

INSTRUCTION SUSPENSION

The intent of defining instruction suspendability is to establish a means for providing reasonable interrupt latency and does not presume to endow CIS11 instructions with an ability to recover from trap conditions from which sequences of basic instructions cannot recover.

Suspension-events refer primarily to events which occur asynchronously to the instruction's execution; these are specifically the interrupts generated by I/O peripheral devices, power-fail traps, and floating point processor exceptions. Secondarily, suspension-events can refer also to those synchronous trap events which occur only for information notification purposes and do not imply that the integrity of the instruction's execution is in jeopardy. Such suspension events include "yellow zone" traps.

Potentially suspendable instructions have a defined architectural mechanism, (PS<8> as described below), by which they can be suspended in mid-execution to allow the processor to service suspension-events and then subsequently to be resumed from the point where they had been suspended.

The presence of suspension-events may cause certain CIS11 instructions to be suspended on some processors. If the instruction is sus-

pended, PS<8> will be set, R7 will be backed up to address the opcode word, and the suspension-event will be serviced. When the instruction is resumed, PS<8> indicates that execution of the instruction has previously begun.

In order to make these instructions suspendable on all processors, the instruction state is part of the user state which is saved by interrupt handling routines. This includes the general registers, condition codes and memory. This state is processor dependent when suspended. Software should not attempt to interpret or modify this state; it must only be saved and restored. Up to 64_{10} words of internal instruction state may also have been pushed onto the stack. This state must not be modified by software. The instruction will remove this state from the stack when it is resumed.

If PS<8> is set prior to executing a potentially suspendable instruction, the effect of the instruction is UNPREDICTABLE.

At the normal completion of an potentially suspendable instruction, PS<8> will be cleared.

The name of the bit PS<8> will be "Instruction Suspension" with the corresponding mnemonic "IS."

All suspendable instructions use PS<8> to indicate instruction suspension. If, when a potentially suspendable instruction is executed, PS<8> is clear, it means that the instruction is being commenced; if it is set, it means that the instruction is being resumed. PS<8> is cleared when:

1. A suspended instruction successfully completes.
2. The processor powers up.
3. A new PS is fetched from vector location with PS<8> clear.
4. RTI or RTT is executed with new PS<8> clear.
5. It is explicitly cleared by an instruction.

PS<8> is set when:

1. A potentially suspendable instruction is interrupted and wishes to be suspended.
2. A new PS is fetched from vector location with PS<8> set.
3. RTI or RTT is executed with PS<8> set.
4. It is explicitly set by an instruction.

The setting of this bit will have no effect on instructions which are not potentially suspendable; such instructions will not implicitly modify this bit.

When an instruction is suspended, the following state may contain

information vital to the resumption of the instruction. The information must be preserved and restored prior to restarting the suspended instruction. This information may vary from one execution of the instruction to another.

1. General registers R0 through R5.
2. Condition code bits (PS<3:0>).
3. Up to 64_{10} words on the stack of the context in which the suspended instruction was executing.
4. Any destinations used by the instruction.

Stack Utilization

CIS11 instructions may use the R6 stack for temporary "scratch" state storage.

The maximum number of additional words which an extended instruction may claim on the R6 stack is 64_{10}. The reason for imposing a limit is to ensure that system software can adequately provide for worst-case stack allocation requirements. In addition to the above restriction, the normal PDP-11 stack-limit mechanism remains in effect for extended instructions just as it does for any other instruction.

If insufficient stack space exists, the instruction will terminate by a memory management abort in such a way that if additional stack space is allocated, the instruction will successfully restart.

NOTATION

dst	destination string
src1	source string 1
src2	source string 2
dscr	descriptor

ADDN/ADDP/ADDNI/ADDPI

Purpose: Add Decimal

Operation: dst ← src2 + src1

Condition Codes:

N: set if dst < 0; cleared otherwise

Z: set if dst = 0; cleared otherwise

V: set if dst cannot contain all significant digits of the result; cleared otherwise

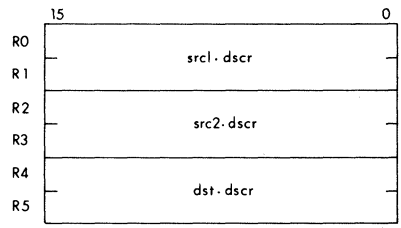
C: cleared

Opcodes:

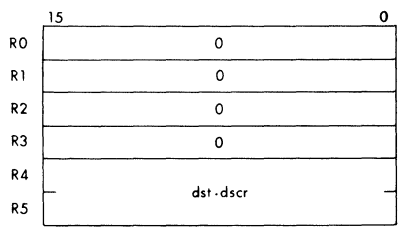
ADDN	076050
ADDP	076070
ADDNI	076150
ADDPI	076170

Description: Src1 is added to src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form—ADDN and ADDP
When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—ADDNI and ADDPI
Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of

the source strings provided that each source string is a valid representation of the specified data type.

2. Source strings may overlap the destination string only if all corresponding digits of the strings are in coincident bytes in memory.

ASHN/ASHP/ASHNI/ASHPI

Purpose: Arithmetic Shift Decimal

Operation: dst ← src * (10 ** shift count)

Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result; cleared otherwise
- C: cleared

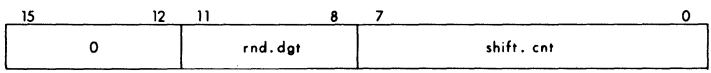
Opcodes:

ASHN	076056
ASHP	076076
ASHNI	076156
ASHPI	076176

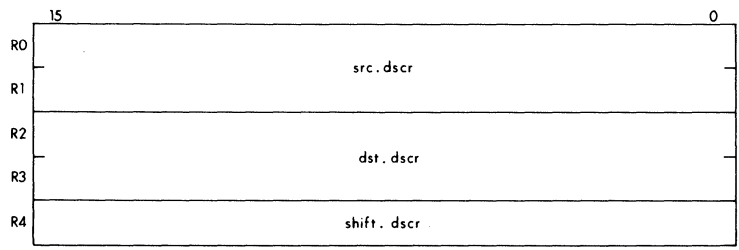
Description: The decimal number specified by the source descriptor is arithmetically shifted and stored in the area specified by the destination descriptor. The shifted result is aligned with the least significant digit position in the destination string. The shift count is a 2's complement byte whose value ranges from -128_{10} to $+127_{10}$. If the shift count is positive, a shift in the direction of least to most significant digits is performed. A negative shift count performs a shift from most to least significant digit. Thus, the shift count is the power of ten by which the source is multiplied; negative powers of ten effectively divide. Zero digits are supplied for vacated digit positions. A zero shift count will move the source to the destination. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

A negative shift count invokes a rounding operation. The result is constructed by shifting the source the specified number of digit positions. The rounding digit is then added to the most significant digit which was shifted out. If this sum is less than 10_{10} the shifted result is stored in the destination string. If the sum is 10_{10} or greater, the magnitude of the shifted result is increased by 1 and then stored in the destination string. If no rounding is desired, the rounding digit should be zero.

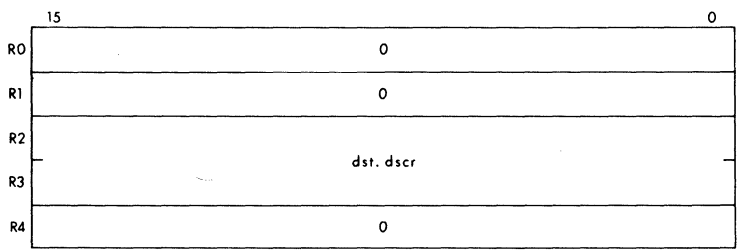
The shift count and rounding digit are represented in a single word referred to as the shift descriptor. Bits <15:12> of this word must be zero.



Register Form—ASHN and ASHP
When the instruction starts, the operands must have been placed in the general registers. The source descriptor is placed in R0-R1, the destination descriptor is placed in R2-R3, and the shift descriptor is placed in R4.



When the instruction is completed, the source descriptor registers and shift descriptor register are cleared.



In-line Form—ASHNI and ASHPI
The words which follow the opcode word in the instruction stream are a word address pointer to a two-word decimal string source descriptor, a word address pointer to a two-word decimal string desti-

nation descriptor, and a shift descriptor word. R0-R6 are unchanged when the instruction is completed.

Notes:

1. If bits <15:12> of the shift descriptor word are not zero, the effect of the instruction is unpredictable.
2. If bits <11:8> of the shift descriptor are not a valid decimal digit, the results of the instruction are unpredictable.
3. Any overlap of the source and destination strings will produce unpredictable results.

CMPC/CMPCI

Purpose: Compare Character

Operation: Src1 is compared with src2 (src1-src2)

Condition Codes: The condition codes are based on the arithmetic comparison of the most significant pair of unequal src1 and src2 characters (src1.byte-src2.byte)

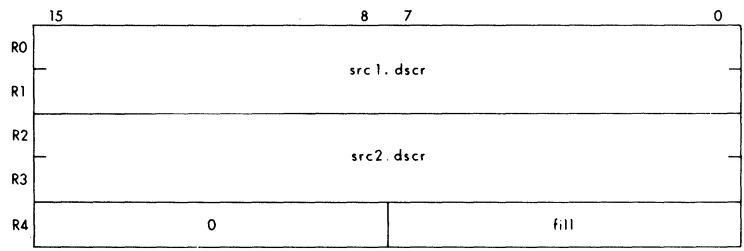
- N: set if result < 0; cleared otherwise
- Z: set if result = 0: cleared otherwise
- V: set if there was arithmetic overflow, that is, src1.byte<7> and src2.byte<7> were different, and src2.byte<7> was the same as bit <7> of (src1.byte-src2.byte); cleared otherwise
- C: cleared if there was a carry from the most significant bit of the result; set otherwise

Opcodes:		
	CMPC	076044
	CMPCI	076144

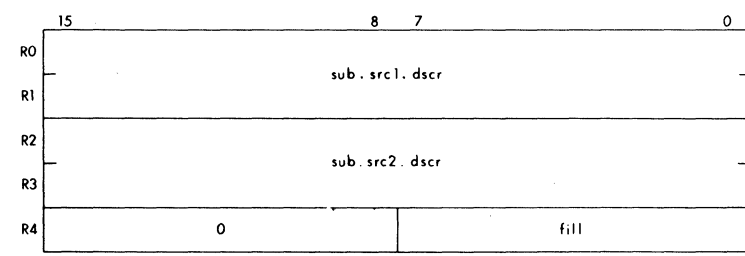
Description: Each character of src1 is compared with the corresponding character of src2 by examining the character strings from most significant to least significant characters. If the character strings are of unequal length, the shorter character string is conceptually extended to the length of the longer character string with fill characters beyond its least significant character. The instruction terminates when the first corresponding unequal characters are found or when both character strings are exhausted. The condition codes reflect the last comparison, permitting the unsigned branch instructions to test the result.

Register Form—CMPC

When the instruction starts, the operands must have been placed in the general registers. The first source character string descriptor is placed in R0-R1, the second source character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero.



The instruction terminates with substring descriptors in R0-R1 and R2-R3 which represent the portion of each source character string beginning with the most significant corresponding unequal characters. R0-R1 contain a descriptor for the unequal portion of the original src1 string; R2-R3 contain a descriptor for the unequal portion of the original src2 string. A vacant character string descriptor indicates that the entire source character string was equal to the corresponding portion of the other source character string, including extension by the fill character; its address is one greater than that of the least significant character of the character string.



In-line Form—CMPCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string src1 descriptor,

a word address pointer to a two-word character string src2 descriptor, and a word whose low-order half contains the fill character and whose high-order half must be zero. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source character strings.
2. If the src1 character string is vacant, the fill character will be compared with src2. If the src2 character string is vacant, the fill character will be compared with src1. If both character strings are vacant, the condition codes will indicate equality.
3. CMPC—If an initial source character string descriptor is vacant, the resulting substring descriptor is the same as the original character string descriptor.
4. A test for success is BEQ; a test for failure is BNE.
5. When the instruction terminates, the condition codes will be set as if a CMPB instruction operated on the most significant unequal characters. If both strings are initially vacant or are identical, the condition codes will be set as if the last characters to be compared were identical. This results in equality with N cleared, Z set, V cleared, and C cleared.
6. Both CMPC and CMPCI update the condition codes. CMPC returns substring descriptors.

CMPN/CMPP/CMPNI/CMPPI

Purpose: Compare Decimal

Operation: Src1 is compared with src2 (src1-src2)

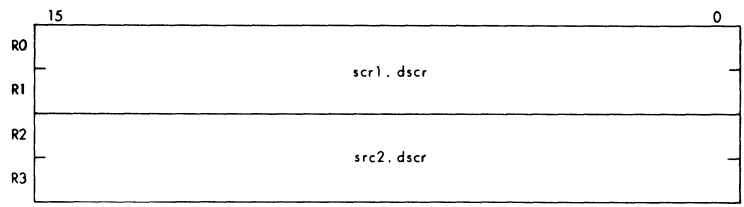
Condition Codes:
- N: set if src1 < src2; cleared otherwise
- Z: set if src1 = src2; cleared otherwise
- V: cleared
- C: cleared

Opcodes:

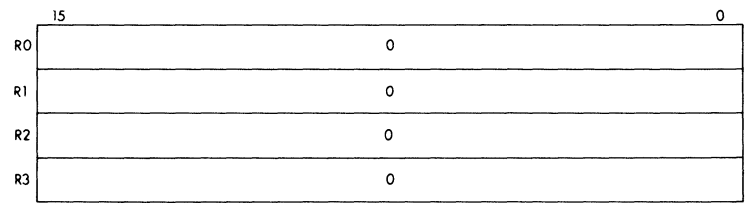
CMPN	076052
CMPP	076072
CMPNI	076152
CMPPI	076172

Description: Src1 is arithmetically compared with src2. The condition codes reflect the comparison. The signed branch instruction can be used to test the result.

Register Form—CMPN and CMPP
When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, and the second source descriptor is placed in R2-R3.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—CMPNI and CMPPI
Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Note:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.

CVTLN/CVTLP/CVTLNI/CVTLPI

Purpose: Convert Long to Decimal

Operation: decimal string ← long integer

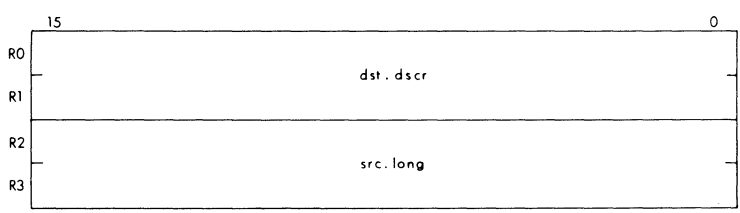
Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result; cleared otherwise
- C: cleared

Opcodes:

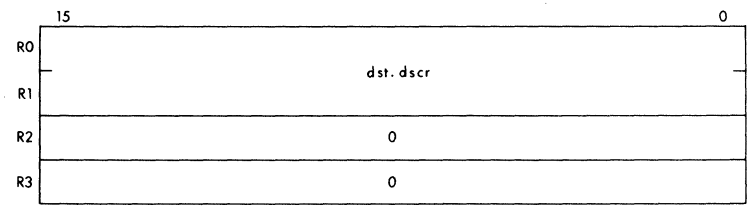
CVTLN	076057
CVTLP	076077
CVTLNI	076157
CVTLPI	076177

Description: The source long integer is converted to a decimal string. The condition codes reflect the result stored in the destination decimal string, and whether all significant digits were stored.

Register Form—CVTLN and CVTLP
When the instruction starts, the operands must have been placed in the general registers. The destination descriptor is placed in R0-R1, and the source long integer is placed in R2-R3.



When the instruction is completed, the source long integer registers are cleared.



In-line Form—CVTLNI and CVTLPI
The words which follow the opcode word in the instruction stream are a word address pointer to a two-word decimal string descriptor, and a word address pointer to a two-word long integer source. R0-R6 are unchanged when the instruction is completed.

Notes:

1. Register forms use a long integer oriented with the sign and high-order portion in R2, and the low-order portion in R3.
2. In-line forms use a long integer oriented with the low-order portion in src.long, and the sign and high-order portion in src.long + 2.

CVTNL/CVTPL/CVTNLI/CVTPLI

Purpose: Convert Decimal to Long

Operation: long integer ← decimal string

Condition Codes: The condition codes are based on the long integer destination and on the sign of the source decimal string.

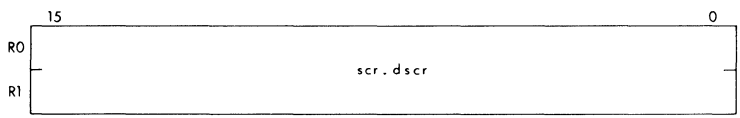
- N: set if long.integer < 0; cleared otherwise
- Z: set if long.integer = 0; cleared otherwise
- V: set if long.integer dst cannot correctly represent the 2's complement form of the result; cleared otherwise
- C: set if src < 0 and long.integer ≠ 0; cleared otherwise

Opcodes:

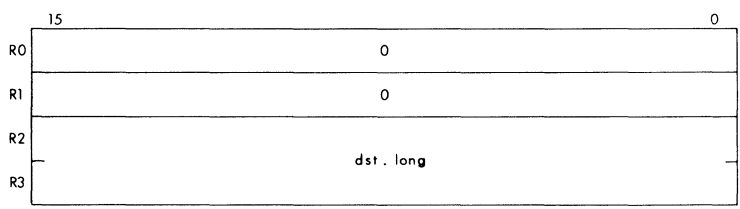
CVTNL	076053
CVTPL	076073
CVTNLI	076153
CVTPLI	076173

Description: The source decimal string is converted to a long integer. The condition codes reflect the result of the operation, and whether significant digits were not converted.

Register Form—CVTNL and CVTPL
When the instruction starts, the operands must have been placed in the general registers. The source decimal string descriptor is placed in R0-R1.



When the instruction is completed, the source decimal string descriptors are cleared, and the destination long integer is returned in R2-R3.



In-line Form—CVTNLI and CVTPLI
The words which follow the opcode word in the instruction stream are a word address pointer to a two-word decimal string source descriptor, and a word address pointer to a two-word long integer destination. R0-R6 are unchanged when the instruction is completed.

Notes:

1. Register forms use a long integer oriented with the sign and high-order portion in R2, and the low-order portion in R3.
2. In-line forms use a long integer oriented with the low-order portion in dst.long, and the sign and high-order portion in dst.long + 2.
3. If the V bit is set, the contents of the long integer destination are the least significant 32 bits of the result.
4. A source whose value is $+2^{31}$ can be represented as a 32-bit binary integer. However, since the destination is a 2's complement

long integer, the resulting condition codes will be: N set, Z cleared, V set, and C cleared.

CVTNP/CVTPN/CVTNPI/CVTPNI

Purpose: Convert Decimal

Operation:

CVTNP/CVTNPI	packed string ← numeric string
CVTPN/CVTPNI	numeric string ← packed string

Condition Codes:

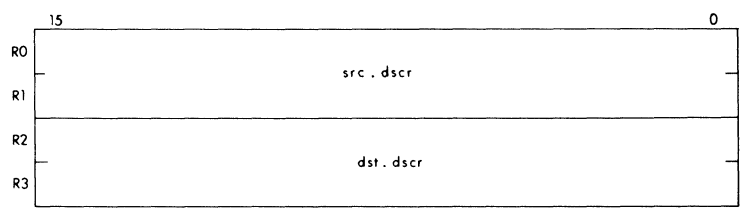
N: set if dst < 0; cleared otherwise
Z: set if dst = 0; cleared otherwise
V: set if dst cannot contain all significant digits of the result; cleared otherwise
C: cleared

Opcodes:

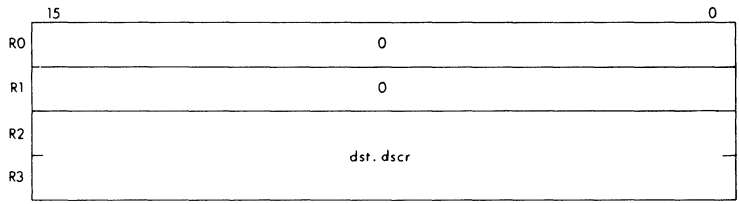
CVTNP	076055
CVTPN	076054
CVTNPI	076155
CVTPNI	076154

Description: These instructions convert between numeric and packed decimal strings. The source decimal string is converted and moved to the destination string. The condition codes reflect the result of the operation, and whether all significant digits were stored.

Register Form—CVTNP and CVTPN
When the instruction starts, the operands must have been placed in the general registers. The source descriptor is placed in R0-R1 and the destination descriptor is placed in R2-R3.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—CVTNPI and CVTPNI
Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The results of the instruction are unpredictable if the source and destination strings overlap.
2. These instructions use both a numeric and a packed decimal string descriptor.

DIVP/DIVPI

Purpose: Divide Decimal

Operation: dst ← src2/src1

Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result or if src1 = 0; cleared otherwise
- C: set if src1 = 0; cleared otherwise

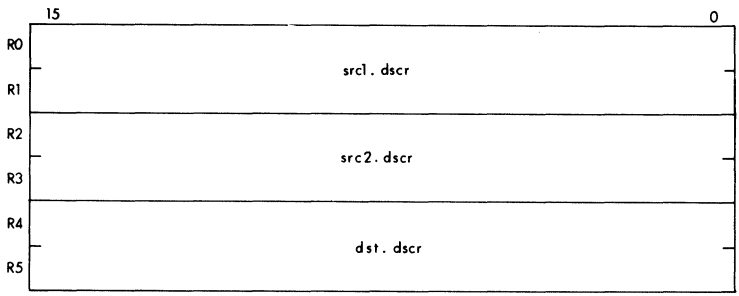
Opcodes:

DIVP	076075
DIVPI	076175

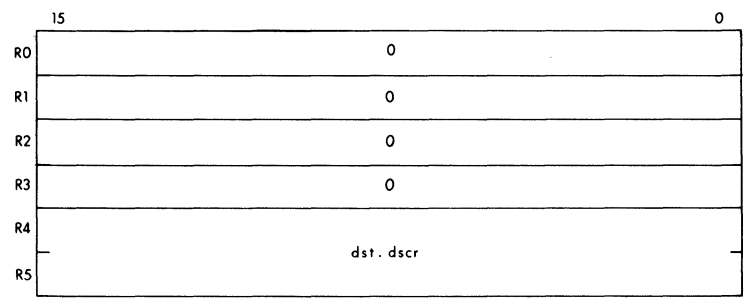
Description: Src2 is divided by src1, and the quotient (fraction truncated) is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form—DIVP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—DIVPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.

2. The results of the instruction are UNPREDICTABLE if the source and destination strings overlap.
3. Division by zero will set the V and C bits. The destination string, and the N and Z condition code bits will be UNPREDICTABLE.
4. No numeric string divide instruction is provided.

LOCC/LOCCI

Purpose: Locate Character

Operation: Search source character string for a character

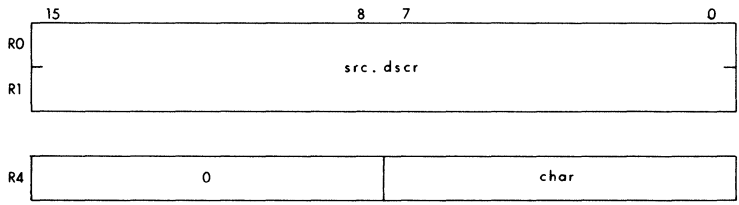
Condition Codes: The condition codes are based on the final contents of R0.

N: set if R0<15> set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

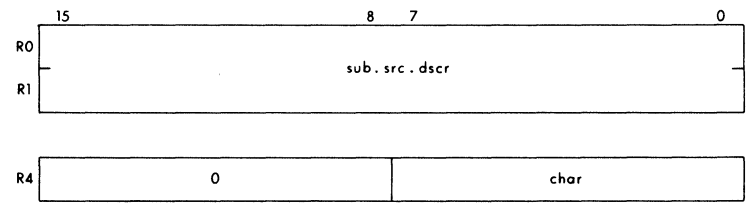
Opcodes:		
	LOCC	076040
	LOCCI	076140

Description: The source character string is searched from most significant to least significant character until the first occurrence of the search character. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with the located character. If the source character string contains only characters not equal to the search character, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form—LOCC
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the search character is placed in R4<7:0>, and R4<15:8> must be zero.

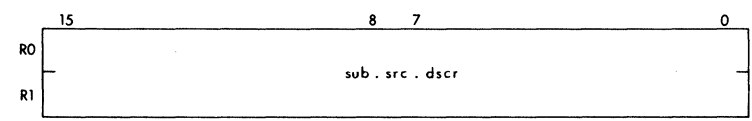


When the instruction is completed, R0-R1 contain a character set descriptor which represents the substring of the source character string beginning with the located character.



In-line Form—LOCCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word whose low-order half contains the search character and whose high-order half must be zero. When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the located character. R2-R6 are unchanged.



Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating no match

was found. The original source character string descriptor is returned in R0-R1.

2. A test for success is BNE; a test for failure is BEQ.
3. The condition codes will be set as if this instruction were followed by TST R0.

L2DR

Purpose: Load Two Descriptors

Operation: Load word pairs into R0-R1 and R2-R3

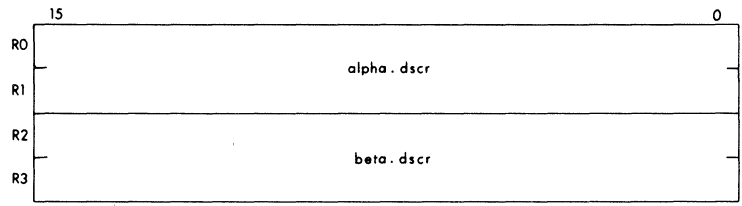
Condition Codes:
N: not affected
Z: not affected
V: not affected
C: not affected

Opcodes: L2DR 07602r

Description: This instruction augments the character and decimal string instructions by efficiently loading string descriptors into the general registers.

A descriptor "alpha" is loaded into R0-R1; a second descriptor "beta" is loaded into R2-R3. The address of the descriptors is determined by the addressing mode @(Rr)+ where r is the low-order three bits of the opcode word. The address of the descriptor "alpha" is derived by applying this addressing mode once; the address of the descriptor "beta" is derived by applying this addressing mode a second time. The addressing mode autoincrements the indicated register by two. The addressing mode computation is not affected by the descriptors which are loaded into the general registers. The words which contain the addresses of the descriptors are in consecutive words in memory; the descriptions themselves may be anywhere in memory. The condition codes are not affected.

When the instruction is completed, the "alpha" descriptor is in R0-R1 and the "beta" descriptor is in R2-R3.



L3DR

Purpose: Load Three Descriptors

Operation: Load word pairs into R0-R1, R2-R3, and R4-R5

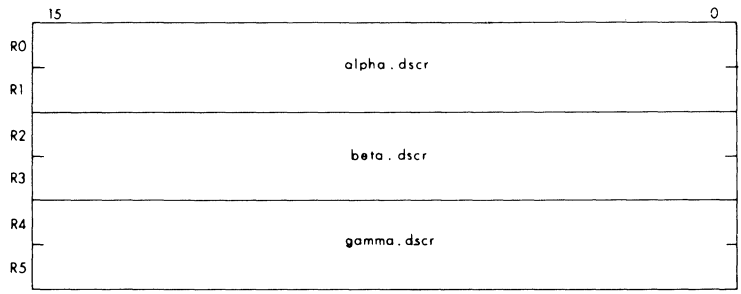
Condition Codes:
N: not affected
Z: not affected
V: not affected
C: not affected

Opcodes: L3DR 07606r

Description: This instruction augments the character and decimal string instructions by efficiently loading string descriptors into the general registers.

A descriptor "alpha" is loaded into R0-R1; a second descriptor "beta" is loaded into R2-R3; a third descriptor "gamma" is loaded into R4-R5. The address of the descriptors is determined by the addressing mode @(Rr)+ where r is the low-order three bits of the opcode word. The address of the descriptor "alpha" is derived by applying this addressing mode once. The address of the descriptor "beta" is derived by applying this addressing mode a second time. The address of the descriptor "gamma" is derived by applying this addressing mode a third time. The addressing mode autoincrements the indicated register by two. The addressing mode computation is not affected by the descriptors which are loaded into the general registers. The words which contain the addresses of the descriptors are in consecutive words in memory; the descriptors themselves may be anywhere in memory. The condition codes are not affected.

When the instruction is completed, the "alpha" descriptor is in R0-R1, the "beta" descriptor is in R2-R3 and the "gamma" descriptor is in R4-R5.



MATC/MATCI

Purpose: Match Character

Operation: Search source character string for object character string

Condition Codes: The condition codes are based on the final contents of R0.

N: set if R0<15> set; cleared otherwise

Z: set if R0 = 0; cleared otherwise

V: cleared

C: cleared

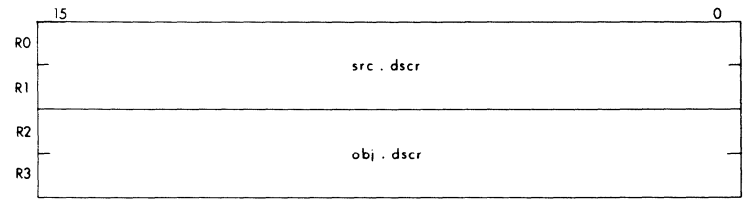
Opcodes:

MATC	076045
MATCI	076145

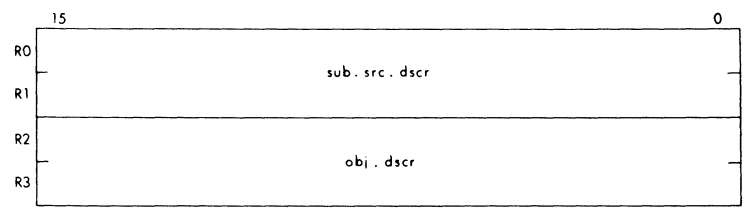
Description: The source character string is searched from most significant to least significant character for the first occurrence of the entire object character string. A character string descriptor is returned in R0-R1 which represents the portion of the original source character string from the most significant character which completely matches the object character string to the end of the source character string. If the object character string did not completely match any portion of the source character string, the character descriptor returned in R0-R1 is vacant with an address one greater than the least significant character in the source string. The condition codes reflect the resulting value in R0. If the Z bit is cleared, the entire object was successfully matched with the source character string; if the Z bit is set, the match failed.

Register Form—MATC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the object character string descriptor is placed in R2-R3.

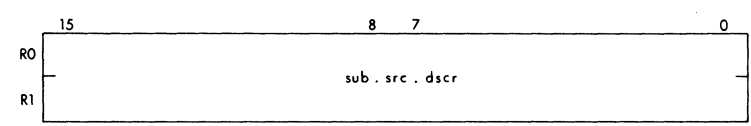


The instruction terminates with a character substring descriptor returned in R0-R1 which represents the portion of the original source character string beginning with the most significant character to competely match the object character string.



In-line Form—MATCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word address pointer to a two-word character string object descriptor. The instruction terminates with a character substring descriptor returned in R0-R1 which represents the portion of the original source character string beginning with the most significant character to completely match the object character string. R2-R6 are unchanged when the instruction is completed.



Notes:

1. The operation of this instruction is unaffected by any overlap of the source and object character strings.
2. A vacant object character string matches any nonvacant source character string. A vacant source character string will not match any object character string. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating no match was found. The original source character string descriptor is returned in R0-R1.
3. If the length of the object character string is greater than that of the source character string, no match is found; R0-R1 and the condition codes will be updated.
4. A test for success is BNE; a test for failure is BEQ.
5. The condition codes will be set as if this instruction were followed by TST R0.

MOVC/MOVCI

Purpose: Move Character

Operation: dst ← src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character string lengths (result = src.len−dst.len).

- N: set if result < 0; cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len−dst.len); cleared otherwise
- C: cleared if there was a carry from the most significant bit of the result; set otherwise

Opcodes:

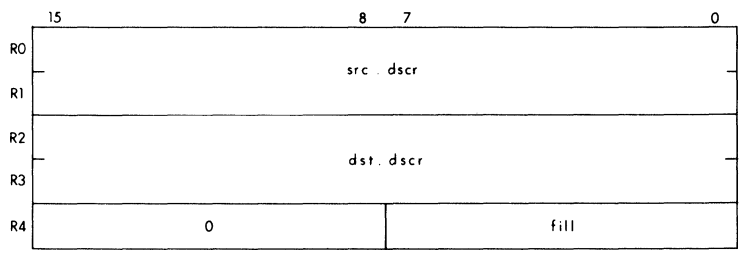
MOVC	076030
MOVCI	076130

Description: The character string specified by the source descriptor is moved into the area specified by the destination descriptor. It is aligned by the most significant character. The condition codes reflect

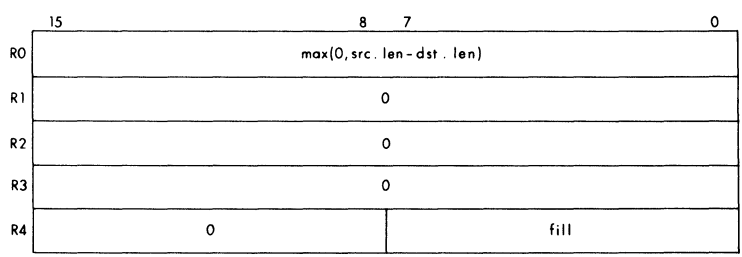
an arithmetic comparison of the original source and destination lengths. If the source string is shorter than the destination string, the fill character is used to complete the least significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form—MOVC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero.



When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared.



In-line Form—MOVCI

The words which follow the opcode word in the instruction stream are

a word address pointer to a two-word character string source descriptor, a word address pointer to a two-word character string destination descriptor, and a word whose low-order half contains the fill character and whose high-order half must be zero. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. The condition codes will be updated. MOVC will update the general registers.
3. MOVC — When the instruction terminates, R0 is zero only if Z or C is set.
4. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.

MOVRC/MOVRCI

Purpose: Move Reverse Justified Character

Operation: dst ← reverse justified src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character string lengths (result = src.len−dst.len).

- N: set if result < 0; cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len−dst.len); cleared otherwise
- C: cleared if there was a carry from the most significant bit of the result; set otherwise

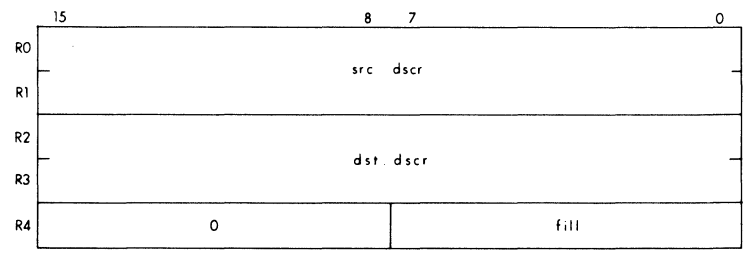
Opcodes:

MOVRC	076031
MOVRCI	076131

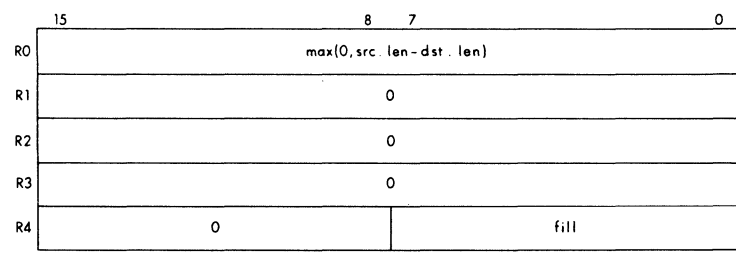
Description: The character string specified by the source descriptor is moved into the area specified by the destination descriptor. It is aligned by the least significant character. The condition codes reflect an arithmetic comparison of the original source and destination lengths. If the source string is shorter than the destination string, the fill character is used to complete the most significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the most significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form—MOVRC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero.



When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared.



In-line Form—MOVRCI
The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, a word address pointer to a two-word character string destination descriptor, and a word whose low-order half contains the fill character and whose high-order half must be zero. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVRC will update the general registers.
3. MOVRC—When the instruction terminates, R0 is zero only if Z or C are set.
4. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.

MOVTC/MOVTCI

Purpose: Move Translated Character

Operation: dst ← translated src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character string lengths (result = src.len−dst.len).

- N: set if result < 0; cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len−dst.len); cleared otherwise
- C: cleared if there was a carry from the most significant bit of the result; set otherwise

Opcodes:

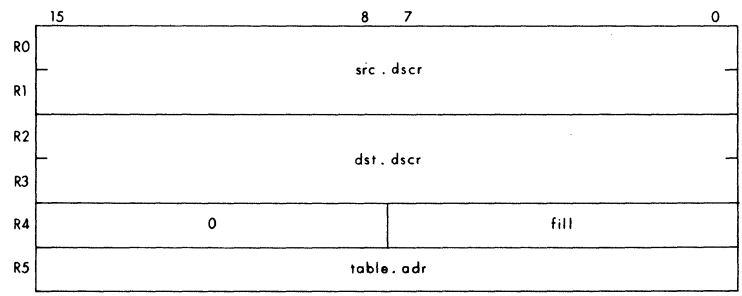
MOVTC	076032
MOVTCI	076132

Description: The character string specified by the source descriptor is translated and moved into the area specified by the destination descriptor. It is aligned by the most significant character. Translation is accomplished by using each source character as an 8-bit positive integer index into a 256-byte table, the address of which is an operand of the instruction. The byte at the indexed location in the table is stored in the destination string. The condition codes reflect an arithmetic comparison of the original source and destination lengths.

If the source string is shorter than the destination string, the untranslated fill character is used to complete the least significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are translated and moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form—MOVTC

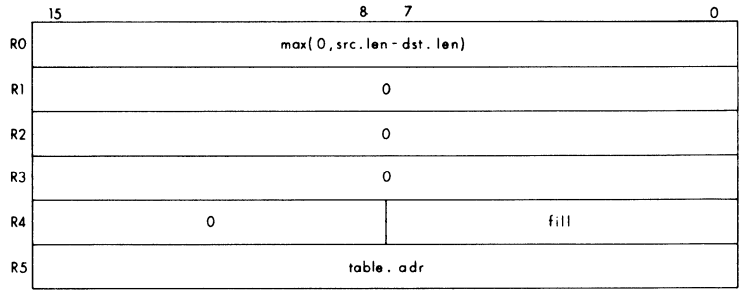
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, R4<15:8> must be zero, and the translation table address is placed in R5.



When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared.

In-line Form—MOVTCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, a word address pointer to a two-word character string destination



descriptor, a word whose low-order half contains the fill character and whose high-order half must be zero, and a word containing the address of the translation table. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.
2. If the destination string overlaps the translation table in any way, the results of the instruction will be UNPREDICTABLE.
3. If the source string is vacant, the untranslated fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVTC will update the general registers.
4. MOVTC—When the instruction terminates, R0 is zero only if Z or C are set.
5. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.
6. The effect of the instruction is UNPREDICTABLE if the entire 256-byte translation table is not in readable memory.

MULP/MULPI

Purpose: Multiply Decimal

Operation: dst ← src2 * src1

Condition N: set if dst < 0; cleared otherwise

Codes:

Z: set if dst = 0; cleared otherwise

V: set if dst cannot contain all significant digits of the result; cleared otherwise

C: cleared

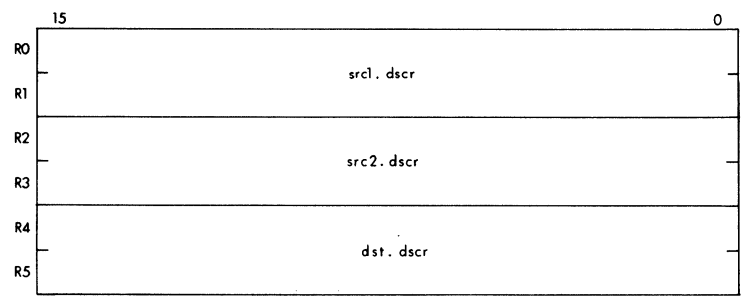
Opcodes:

MULP	076074
MULPI	076174

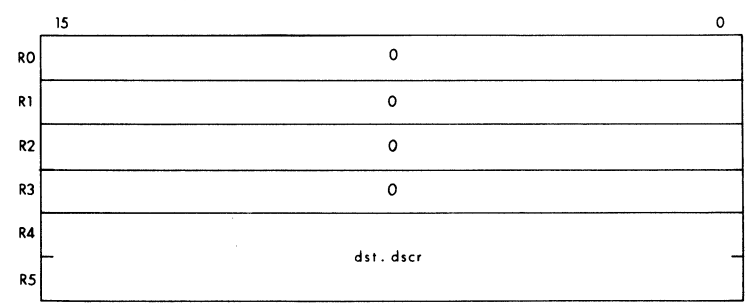
Description: Src1 and src2 are multiplied, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form—MULP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—MULPI
Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.
2. The results of the instruction are UNPREDICTABLE if the source and destination strings overlap.
3. No numeric string multiply instruction is provided.

SCANC/SCANCI

Purpose: Scan Character

Operation: Search source character string for a member of the character set

Condition Codes: The condition codes are based on the final contents of R0.

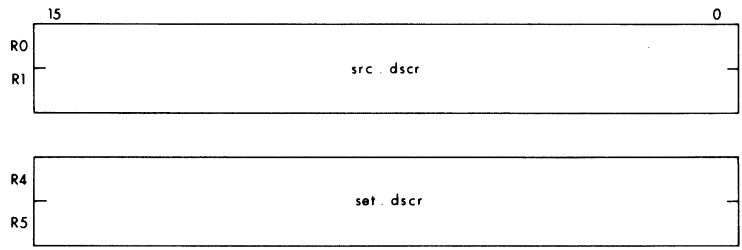
N: set if R0<15> set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

Opcodes:		
	SCANC	076042
	SCANCI	076142

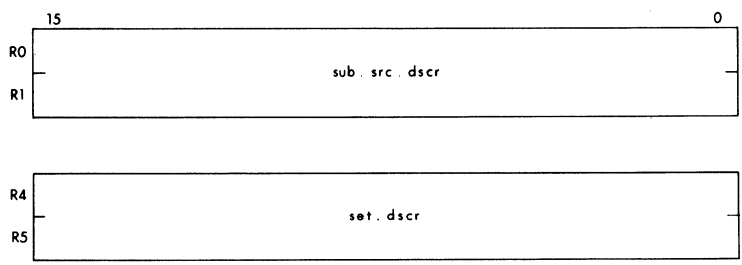
Description: The source character string is searched from most significant to least significant character until the first occurrence of a character which is a member of the character set. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with the located member of the character set. If the source character string contains only characters which are not in the character set, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form—SCANC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the character set descriptor is placed in R4-R5.

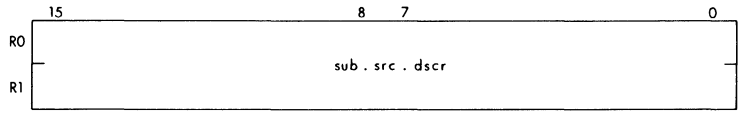


When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is a member of the character set.



In-line Form—SCANCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word address pointer to a two-word character set descriptor. When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is a member of the character set. R2-R6 are unchanged.



Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that no characters in the set were found. The original source character string descriptor is returned in R0-R1.
2. The source character string and character set table may overlap in any way.
3. A test for success is BNE; a test for failure is BEQ.
4. The condition codes will be set as if this instruction were followed by TST R0.
5. The effect of the instruction is UNPREDICTABLE if the entire 256-byte character set table is not in readable memory.

SKPC/SKPCI

Purpose: Skip Character

Operation: Search source character string until a character other than the search character is found

Condition Codes: The condition codes are based on the final contents of R0.

- N: set if R0<15> set; cleared otherwise
- Z: set if R0 = 0; cleared otherwise
- V: cleared
- C: cleared

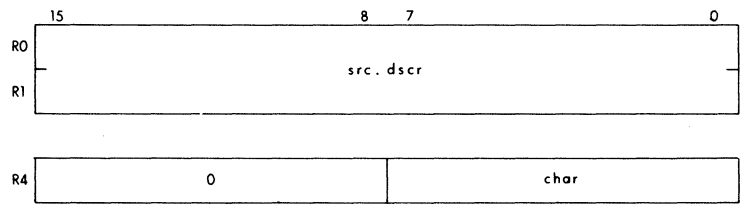
Opcodes:		
	SKPC	076041
	SKPCI	076141

Description: The source characer string is searched from most significant to least significant character until the first occurrence of a character which is not the search character. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning which the most significant character

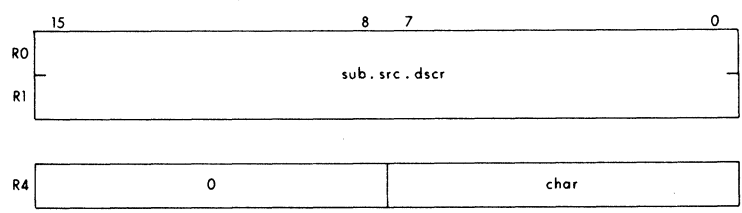
which was not equal to the search character. If the source character string contains only characters equal to the search character, the instruction returns a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form—SKPC

When the instruction starts, the operands must have been placed in the general registers. The source character string decriptor is placed in R0-R1, the search character is placed in R4<7:0>, and R4<15:8> must be zero.



When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which was not equal to the search character.

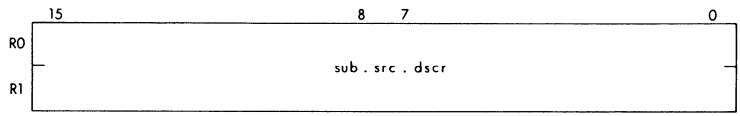


In-line Form—SKPCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word whose low-order half contains the search character and whose high-order half must be zero. When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which was not equal to the search character. R2-R6 are unchanged.

Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating the character string only contained search characters. The original source character string descriptor is returned in R0-R1.
2. The condition codes will be set as if this instruction were followed by TST R0.



SPANC/SPANCI

Purpose: Span Character

Operation: Search source character string for a character which is not a member of the character set.

Condition Codes: The condition codes are based on the final contents of R0.

- N: set if R0<15> set; cleared otherwise
- Z: set if R0 = 0; cleared otherwise
- V: cleared
- C: cleared

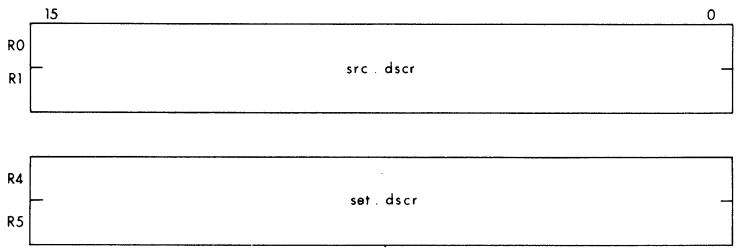
Opcodes:

SPANC	076043
SPANCI	076143

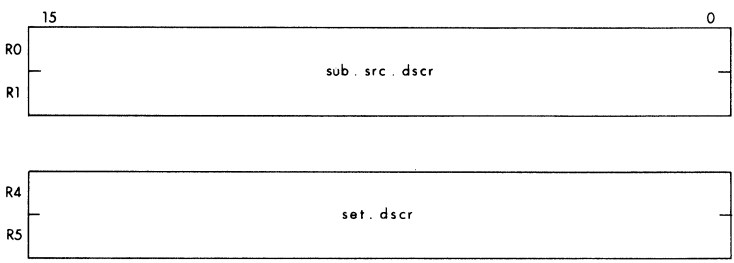
Description: The source character string is searched from most significant to least significant character until the first occurrence of character which is not a member of the character set. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with the character which is not a member of the character set. If the source character string contains only characters which are in the character set, the instruction returns a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form—SPANC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the character set descriptor is placed in R4-R5.

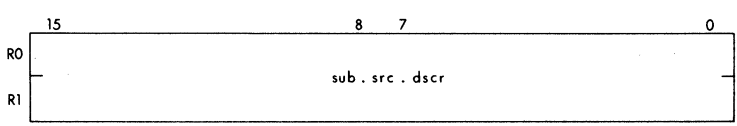


When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is not a member of the character set.



In-line Form—SPANCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word address pointer to a two-word character set descriptor. When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is not a member of the character set. R2-R6 are unchanged.



Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that only characters in the set were found. The original source character string descriptor is returned in R0-R1.
2. The source character string and character set table may overlap in any way.
3. The condition codes will be set as if this instruction were followed by TST R0.
4. The effect of the instruction is UNPREDICTABLE if the entire 256-byte character set table is not in readable memory.

SUBN/SUBP/SUBNI/SUBPI

Purpose: Subtract Decimal

Operation: dst ← src2 − src1

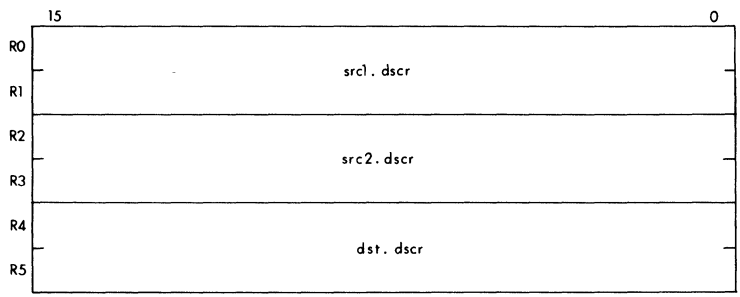
Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result; cleared otherwise
- C: cleared

Opcodes:

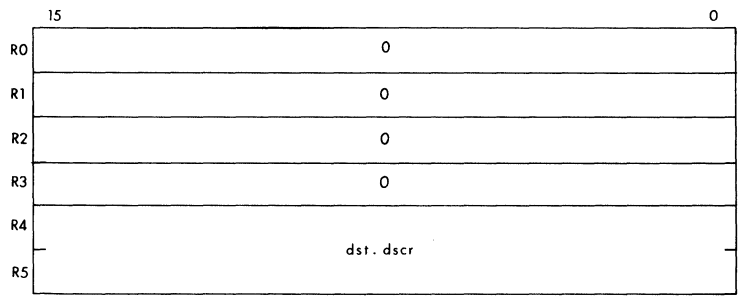
SUBN	076051
SUBP	076071
SUBNI	076151
SUBPI	076171

Description: Src1 is subtracted from src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form—SUBN and SUBP
When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—SUBNI and SUBPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.
2. Source strings may overlap the destination string only if all corresponding digits of the strings are in coincident bytes in memory.

APPENDIX A

UNIBUS I/O PAGE DEVICE ADDRESSES AND VECTOR ASSIGNMENTS

Device	Address	Size in Words	Number of Devices	
AA11	776750	8	1	(first unit)
AA11	776400	8	4	(extra units)
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	
Customer	764000	1024	1	
DC11	774000	4	32	
DC14-D	777360	8	1	
Diagnostics	760000	4	1	
DL11-A	777560	4	1	(console)
DL11-A	776500	4	16	
DL11-B	777560	4	1	(console)
DL11-B	776500	4	16	
DL11-C	775610	4	31	
DL11-D	775610	4	31	
DL11-E	775610	4	31	
DL11-W	777546	1	1	(line clock, first unit only)
DL11-W	777560	4	1	(console)
DL11-W	776500	4	16	
DM11	775000	4	16	
DM11-BB	770500	4	16	(modem control for DM11)
DN11-AA	775200	4	16	

Device	Address	Size in Words	Number of Devices	
DN11-DA	775200	1	64	
DP11	774400	4	−32	(assigned backwards)
DR11-A/C	767600	4	−16	(assigned backwards)
DR11-B(1)	772410	4	1	
DR11-B(2)	772430	4	1	
DS11	775400	67	1	
DT11	777420	1	8	
DV11	775000	16	4	
DX11	776200	16	2	
Floating CSRs	760010	1020	1	
FP11	772160	8	1	
GT40	772000	4	4	
ICR/ICS11	771000	256	1	
IP11/IP300	771000	128	2	
KE11	777300	8	2	
KG11	770700	4	8	
KL11	776500	4	16	
KL11	777560	4	1	(console)
KT11	772200	64	1	
KT11-SR3	772516	1	1	
KU116-AA	777540	1	1	
KW11-L	777546	1	1	
KW11-P	772540	4	1	
KW11-W	772400	4	1	
LP11	777514	2	1	(LP0)
LP11	764004	2	1	(LP1)
LP11	764014	2	1	(LP2)
LP11	764024	2	1	(LP3)
LP11	764034	2	1	(LP4)
LP11	764044	2	1	(LP5)
LP11	764054	2	1	(LP6)
LP11	764064	2	1	(LP7)
LP20	775400	32	2	
LPA11-K	770460	8	1	
LPS11	770400	16	1	
LS11	777514	2	1	
LV11	777514	2	1	
M792	773000	32	8	

Device	Address	Size in Words	Number of Devices	
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_readers	772600	32	1	(2 per PA611)
PA611_punches	772700	32	1	(2 per PA611)
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-11/24	777570	68	1	
PDP-11/34A	777570	68	1	
PDP-11/35	777570	68	1	
PDP-11/40	777570	68	1	
PDP-11/44	777570	68	1	
PDP-11/45	777570	68	1	
PDP-11/55	777570	68	1	
PDP-11/60	777570	68	1	
PDP-11/70	777570	68	1	
PR11	777550	4	1	
RC11	777440	8	1	
Reserved	770100	32	1	
Reserved	770440	8	1	
Reserved	772150	4	1	
Reserved	772420	4	1	
Reserved	772514	1	1	
Reserved	772550	8	1	
Reserved	775606	1	1	
Reserved	777000	56	1	
Reserved	777200	32	1	
Reserved	777510	2	1	
Reserved	777520	4	1	
Reserved	777540	3	1	
RF11	777460	8	1	

Device	Address	Size in Words	Number of Devices	
RH70/11_alt	776300	32	1	(Alternate RS/RP/RM/TJ)
RK611	777440	16	1	
RK11	777400	8	1	
RL11	774400	4	1	
RM03/04/05	776700	22	1	(RH70/RH11)
RP04/05/06	776700	22	1	(RH70/RH11)
RP11	776700	16	1	(RH70/RH11)
RS04	772040	16	1	(RH70/RH11)
RX11/RX211	777170	4	1	
TA11/DIP11-A	777500	4	1	
TC11	777340	8	1	
Testers	770000	32	1	
TM11/TMB11	772520	8	1	
TR79	764000	4	1	
TS11	772520	2	4	
TU16/45/77	772440	16	1	(RH70/RH11)
TU58	776500	4	4	
UDC-Units	771000	1	256	
UDC11	771774	2	1	
UET	772140	4	1	
Unibus-Map	770200	64	1	
VSV11	772000	4	4	
VT48	772000	16	1	
VTV01	772600	112	2	
XY11	777530	4	1	

PDP-11 INTERRUPT AND TRAP VECTORS

000	PDP-11 Reserved
004	PDP-11 CPU Errors (Illegal instructions, Bus Errors, Stack Limit, Illegal Internal Address, Microbreak)
010	PDP-11 Reserved Instructions
014	PDP-11 Breakpoint/Trace traps
020	PDP-11 IOT Trap
024	PDP-11 Power Fail
030	PDP-11 EMT Trap
034	PDP-11 TRAP Trap
040	Reserved for System Software

044	Reserved for System Software
050	Reserved for System Software
054	Reserved for System Software
060	DL11(1), KL11(1)
064	DL11(1), KL11(1)
070	PC11, paper tape reader
074	PC11, paper tape punch
100	KW11-L, line clock
104	KW11-P, programmable clock
110	Reserved for System Software
114	CPU
120	XY11, Plotter
124	DR11-B, DMA interface
130	AD01, A/D subsystem
134	AFC11, analog subsystem
140	AA11, display
144	AA11, RSTS/E (crash-dump)
150	alternate RS/RP/RM/TJ
154	UNUSED - Reserved for Digital
160	RL11, disk
164	UNUSED - Reserved for Digital
170	LP/LS/LV11 (#1), USER RESERVED
174	LP/LS/LV11 (#2), USER RESERVED
200	LP/LS/LV11 (#0), LP20 (1), lineprinter
204	RF11, RS03/04 (RH11/RH70), MASSBUS fixed head disk
210	LP20(2), RC11, RK611/RK711
214	TC11, DECtape
220	RK11, disk
224	TM11, TS11, TU16/45, TE16, TU77, MASSBUS Magnetic tape
230	CD11, CM11, CR11
234	ICS/ICR11, IP11/IP300, UDC11
240	PDP-11-PIRQ
244	Floating Point exception
250	Memory Management error
254	RM02/03/50 (RH11/RH70), RP04/5/6 (RH11/RH70), RP11
260	DIP11, TA11
264	RX11, floppy disk
270	LP/LS/LV11 (#3), USER RESERVED
274	LP/LS/LV11 (#4), USER RESERVED
300	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 DC11s (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	Option	Decimal Size (words)	Octal Modulus (address)
1	DC11	4	10
1	TU58	4	10 (See Note 1)
2	KL11(extra)	4	10
2	DL11-A(extra)	4	10
2	DL11-B(extra)	4	10
3	DP11	4	10
4	DM11-A	4	10
5	DN11	2	4
6	DM11-BB	2	4
7	DH11 modem control	2	4
8	DR11-A	4	10
9	DR11-C	4	10
10	PA611(reader+punch)	8	10
11	LPD11	4	10
12	DT11	4	10
13	DX11	4	10
14	DL11-C	4	10
14	DL11-D	4	10
14	DL11-E	4	10
15	DJ11	4	10

Rank	Option	Decimal Size (words)	Octal Modulus (address)
16	DH11	4	10
17	GT40	8	10
17	VSV11	8	10
18	LPS11	12	10
19	DQ11	4	10
20	KW11-W	4	10
21	DU11	4	10
22	DUP11	4	10
23	DV11+modem control	6	10
24	LK11-A	4	10
25	DWUN	4	10
26	DMC11	4	10
26	DMR11	4	10 (DMC before DMR)
27	DZ11	4	10
28	KMC11	4	10
29	LPP11	4	10
30	VMV21	4	10
31	VMV31	4	10
32	VTV01	4	10
33	DWR70	4	10
34	RL11/RLV11	2	4 (after the first)
35	RX02	2	4
36	TS11	2	4 (after the first)
37	LPA11-K	4	10
38	IP11/IP300	2	4
39	KW11-C	4	10
40	RX11	2	4 (after the first)
41	DR11-W	2	4
42	DR11-B	2	4 (after the first)

1 There is no standard configuration for systems with both DC11 and TU58.

FLOATING CSR ADDRESS DEVICES

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	Option	Decimal Size (words)	Octal Modulus (address)
1	DJ11	4	10
2	DH11	8	20
3	DQ11	4	10
4	DU11	4	10
5	DUP11	4	10
6	LK11A	4	10
7	DMC11/DMR11	4	10 (DMC before DMR)
8	DZ11[1] and DZV11	4	10
9	KMC11	4	10
10	LPP11	4	10
11	VMV21	4	10
12	VMV31	8	20
13	DWR70	4	10
14	RL11 and RLV11	4	10 (extra only)
15	LPA11-K	8	20 (extra only)
16	KW11-C	4	10
17	Reserved	4	10
18	RX11	4	10 (extra only)
19	DR11-W	4	10
20	DR11-B	4	10 (after second)

1 DZ11E and DZ11F are dual DZ11s and are treated by the algorithm as two DZ11s.

DEVICE ADDRESSES

776 000 – 760 006 Diagnostics

760 010 (Start of floating addresses)

763 776 (Top of floating addresses)

764 000	TR79	
764 004 – 764 066	LP11(#0-7)	Customer
765 000 – 765 776	M9301	Customer
767 600 – 767776	DR11-A/C	Customer
770 000 – 770 076	Testers	
770 100 – 770 176	Reserved	
770 200 – 770 376	UNIBUS Map	
770 400 – 770 416	AR11	LPS11
770 436		LPS11
770 440 – 770 456	Reserved	
770 460 – 770 476	ADF11/LPA11-K	
770 500 – 770 676	DM11-BB	#1 – #16

770 700 – 770 776	KG11 #1 – #8
771 000 – 771 776	UDC Functional I/O Units ICR/ICS11, IP11/IP300
771 774 – 771 776	UDC11 ICR/ICS11, IP11/IP300
772 000 – 772 036	GT40 (#1-#4), VSV11 (#1-#4), VT48
772 040 – 772 076	RS04
772 100 – 772 136	UNIBUS Memory Parity MM11-LP #1 – MS11-LP #16
772 140 – 772 146	UNIBUS Tester
772 150 – 772 156	Reserved
772 160 – 772 176	FP11 Registers
772 200 – 772 216	Supervisor Instruction Descriptor PDR, reg 0-7
772 220 – 772 236	Supervisor Data Descriptor PDR, reg 0-7

772 240 – 772 256	Supervisor Instruction PDR, reg 0-7
772 260 – 772 276	Supervisor Data PAR, reg 0-7
772 300 – 772 316	Kernel Instruction PDR, reg 0-7
772 320 – 772 336	Kernel Data PDR, reg 0-7
772 340 – 772 356	Kernel Instruction PAR, reg 0-7
772 360 – 772 376	Kernel Data PAR, reg 0-7
772 400 – 772 406	KW11-W
772 410 – 772 416	DR11-B (#1)
772 420 – 772 426	Reserved
772 430 – 772 436	DR11-B (#2)
772 440 – 772 476	TU16/45/77

772 500 – 772 512	OST	
772 514	Reserved	
772 516	Memory Mgt. reg (MMR3)	
772 520 – 772 536	TM11/TMB11/TS11	
772 540 – 772 546	KW11-P	
772 550 – 772 566	Reserved	
772 570 – 772 576	AFC11	
772 600 – 772 676	PA611 Typeset Readers	VTV01
772 700 – 772 776	PA611 Typeset Punches	VTV01
772 760 – 772 776	NCV11	VTV01
773 476		

773 000 – 773 076	BM792-YA	BM873-YA	BM873-YB BM873-YC M792 M9301-XX
773 100 – 773 276	MR11-DB		
773 376			
773 776			
774 000 – 774 376	DC11,	#1 – #32	
774 400 – 774 406	RL11	DP11, #1	
774 776		#32	
775 000 – 775 176	DM11,	#1 – #16	DV11, #1-#4
775 200 – 775 376	DN11-AA/DN11-DA		#1 – #16
775 400 – 775 576	LP20	DS11	
775 604			
775 606	Reserved		
775 610 – 776 176	DL11-C, -D, -E	#1 – #31	
776 200 – 776 276	DX11		

776 300 776 376	alternate RH70/RH11
776 400 776 476	#2 AA11, #5
776 500 776 676	KL11, #1 TU58 DL11-A, -B, -W #16
776 700 776 736 776 752	RP11 — RM03/04/05, RP04/05/06
776 750 776 766	AA11, #1
776 770 776 776	AD01
777 000 777 156	Reserved
777 160 777 166	CM11, CD11 CR11
777 170 777 176	RX11/RX211
777 200 777 276	Reserved

777 300 – 777 336	KE11, #2	
777 340 – 777 356	TC11	
777 360 – 777 376	DC14-D	
777 400 – 777 416	RK11	
777 420 – 777 436	DT11	
777 440 – 777 456	RC11	RK611
777 460 – 777 476	RF11	RK611
777 500 – 777 506	TA11/DIP11-A	
777 510 – 777 512	Reserved	
777 514 – 777 516	LP11/LS11/LV11	
777 520 – 777 526	Reserved	

777 530 777 536	XY11
770 540 777 544	KV116-AA
777 546	DL11-W/KW11-L, line clock
777 550 777 556	PC11/PR11
777 560	DL11-A/DL11-B Console Terminal
777 566	DL11-W/KL11
777 570	Console Switch & Display Register
777 572	(MMR0)
777 574	Memory Mgt. reg (MMR1)
777 576	(MMR2)
777 600 777 616	User Instruction PDR, reg 0-7
777 620 777 636	User Data PDR, reg 0-7
777 640 777 656	User Instruction PAR, reg 0-7
777 660 777 676	User Data PAR, reg 0-7

777 700		R0
777 701		R1
777 702	General registers,	R2
777 703	Set 0	R3
777 704		R4
777 705		R5
777 706	Kernel	R6 (SP)
777 707		R7 (PC)
777 710		R0
777 711		R1
777 712	General registers	R2
777 713	Set 1	R3
777 714		R4
777 715		R5
777 716	Supervisor	R6(SP)
777 717	User	R6 (SP)
777 740	Low Error Address (PDP-11/70)	
777 742	High Error Address (PDP-11/70)	
777 744	Memory System Error (PDP-11/70)	
777 746	Cache Control	
777 750	Maintenance	
777 752	Hit/Miss	
777 754		
777 756		
777 760	Lower Size	(PDP-11/70)
	System Size	
777 762	Upper Size	(PDP-11/70)
777 764	System I/D	(PDP-11/70)
777 766	CPU Error	
777 770	Microprogram Break (PDP-11/70)	
777 772	Program Interrupt Request (PIR)	
777 774	Stack Limit (SL) (PDP-11/70)	
777 776	Processor Status Word (PS)	

NOTE

All presently unused UNIBUS addresses are reserved by Digital.

APPENDIX B

INSTRUCTION TIMING

PDP-11/04 CENTRAL PROCESSOR

INSTRUCTION EXECUTION TIME

The execution time for an instruction depends on the instruction itself and the modes of addressing used. In the most general case, the Instruction Execution Time is the sum of a Basic Time, a Source Address Time, and a Destination Address Time.

$$\text{Instr Time} = \text{Basic Time} + \text{SRC Time} + \text{DST Time}$$

Double Operand instructions require all 3 of these Times, Single Operand instructions require a Basic Time and a DST Time, and with all other instructions the Basic Time is the Instr Time.

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

BASIC TIMES

Double Operand Instruction	Basic Time (μ sec) MOS	Parity MOS
ADD, SUB, BIC, BIS	3.17	3.33
CMP, BIT	2.91	3.07
MOV	2.91	3.07
Single Operand		
CLR, COM, INC, DEC, NEG, ADC, SBS	2.65	2.81
ROR, ROL, ASR, ASL	2.91	3.07
TST	2.39	2.55
SWAB	2.91	3.07
All Branches (branch true)	2.65	2.81
All Branches (branch false)	1.87	2.03
Jump Instructions		
JMP	0.91	0.88
JSR	3.27	3.27
Control, Trap, and Miscellaneous Instructions		
RTS	4.11	4.43
RTI, RTT	5.31	5.79
Set N,Z,V,C	2.39	2.55
Clear N,Z,V,C	2.39	2.55
HALT	1.46	1.62
WAIT	2.13	2.29
RESET	100 ms	100 ms
IOT, EMT, TRAP, BPT	7.95	8.49

ADDRESSING TIMES

ADDRESSING FORMAT **Time (μsec)**

Mode	Description	Symbolic	SRC Time*		DST Time**	
			MOS	Parity MOS	MOS	Parity MOS
0	REGISTER	R	0	0	0	0
1	REGISTER DEFERRED	@R or (R)	0.94	1.10	1.48	1.67
2	AUTO-INCREMENT	(R)+	1.20	1.36	1.76	1.95
3	AUTO-INCREMENT DEFERRED	@(R)+	2.66	2.98	3.20	3.55
4	AUTO-DECREMENT	−(R)	1.20	1.36	1.76	1.95
5	AUTO-DECREMENT DEFERRED	@−(R)	2.66	2.98	3.20	3.55
6	INDEX	X(R)	2.92	3.24	3.46	3.81
7	INDEX DEFERRED	@X(R)	4.38	4.86	4.92	5.43

* For Source time, add the following for odd byte addressing: 0.52 (μsec)

** For Destination time, modify as follows:

a) Add for odd byte addressing with a non-modifying instruction: 0.52 (μsec)

b) Add for odd byte addressing with a modifying instruction modes 1-7: 1.04 (μsec)

c) Subtract for all non-modifying instructions except Mode 0:
MOS: 0.54 Parity MOS: 0.57 (μsec)

d) Add for MOVE instructions Mode 1-7: 0.26 (μsec)

e) Subtract for JMP and JSR instructions, modes 3, 5, 6, 7: 0.52 (μsec)

B.2 PDP-11/34A CENTRAL PROCESSOR

INSTRUCTION EXECUTION TIME

The execution time for an instruction depends on the instruction itself, the modes of addressing used, and the type of memory being referenced. In the most general case, the Instruction Execution Time is the sum of a Source Address Time, a Destination Address Time, and an Execute, Fetch Time.

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Some of the instructions require only some of these times, and are so noted. All Timing information is in microseconds, unless otherwise noted. Times are typical; processor timing can vary ± 10%.

BASIC INSTRUCTION SET TIMING

Double Operand

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Single Operand

$$\text{Instr Time} = \text{DST Time} + \text{EF Time}$$

Branch, Jump, Control, Trap, & Misc

$$\text{Instr Time} = \text{EF Time}$$

NOTES

1) The times specified apply to both word and byte instructions whether odd or even byte.
2) Timing is given without regard for NPR or BR servicing.
3) If the memory management is enabled execution times increase by 0.12 μsec for each memory cycle used.
4) All timing is based on memory with the following performance characteristics:

Memory	Access Time	Cycle Time
MOS (MS11-JP)	.635	.775

- Instruction timings with MOS MS11-L memory for the PDP-11/34A were not available at the time of publication. These timings will be available in the next PDP-11 Processor Handbook. Instruction times with MS11-L memory are approximately 15% faster than with MS11-J memory.

I. SOURCE ADDRESS TIME

Instruction	Source Mode	Memory Cycles	MOS (MS11-JP)
Double Operand	0	0	0.00 μsec
	1	1	1.26
	2	1	1.46
	3	2	2.62
	4	1	1.41
	5	2	2.82
	6	2	2.82
	7	3	4.18

II. DESTINATION TIME

Instruction	Destination Mode	Memory Cycles	MOS
Modifying Single Operand and Modifying Double Operand (Except MOV, SWAB, ROR, ROL ASR ASL)	0	0	0.00
	1	2	1.74
	2	2	1.89
	3	3	3.15
	4	2	1.89
	5	3	3.25
	6	3	3.35
	7	4	4.66
MOV	0	0	0.00
	1	1	0.93
	2	1	0.93
	3	2	2.29
	4	1	1.13
	5	2	2.34
	6	2	2.49
	7	3	3.75
MTPS	0	0	0.00
	1	1	0.95
	2	1	1.26
	3	2	2.51
	4	1	1.26
	5	2	2.51
	6	2	2.69
	7	3	4.20

	Destination Mode	Memory Cycles	MOS
	0	0	0.00
	1	1	0.64
	2	1	0.64
MFPS	3	2	2.08
	4	1	0.82
	5	2	2.08
	6	2	2.26
	7	3	3.51

III. EXECUTE, FETCH TIME

DOUBLE OPERAND

Instruction	Memory Cycles	MOS
ADD, SUB, CMP, BIT, BIC, BIS, XOR	1	2.16
MOV	1	1.96

SINGLE OPERAND

CLR, COM, INC, DEC, ADC, SBC, TST	1	1.96
SWAB, NEG	1	2.16
ROR, ROL, ASR, ASL	1	2.31
MTPS	2	3.12
MFPS	2	2.12

EIS INSTRUCTIONS (use with DST times)

MUL	1	*8.95
DIV (overflow)	1	2.91
		12.61
ASH	1	**4.31
ASHC	1	**4.31

MEMORY MANAGEMENT INSTRUCTIONS

MFPI (D)	2	3.14
MTPI (D)	2	3.34

* Add 200ns for each bit transition in serial data from LSB to MSB
** Add 200ns per shift

Instruction	Destination Mode	Memory Cycles	MOS
SWAB, ROR, ROL, ASR, ASL	0	0	0.00
	1	2	1.54
	2	2	1.69
	3	3	2.95
	4	2	1.74
	5	3	3.05
	6	3	3.15
	7	4	4.46
Non-Modifying Single Operand and Double Operand	0	0	0.00
	1	1	1.26
	2	1	1.41
	3	2	2.67
	4	1	1.46
	5	2	2.77
	6	2	2.87
	7	3	4.18
MFPI (D) MTPI (D)	0	0	0.00
	1	1	1.24
	2	1	1.44
	3	2	2.45
	4	1	1.44
	5	2	2.45
	6	2	2.65
	7	3	3.96

BRANCH INSTRUCTIONS

Instruction	Memory Cycles	MOS
BR, BNE, BEQ, (Branch) BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLOS, BHIS, BLO	1	2.31
(No Branch)	1	1.76
SOB (Branch)	1	2.51
(No Branch)	1	2.11

JUMP INSTRUCTIONS

	Destination Mode	Memory Cycles	MOS
JMP	1	1	1.96
	2	1	2.31
	3	2	3.37
	4	1	2.16
	5	2	3.32
	6	2	3.32
	7	3	4.78
JSR	1	2	3.44
	2	2	3.59
	3	3	4.65
	4	2	3.44
	5	3	4.65
	6	3	4.85
	7	4	6.06

Instruction	Memory Cycles	MOS
RTS	2	3.57
MARK	2	4.52
RTI, RTT	3	4.98
Set or Clear C,V,N,Z	1	2.16
HALT	1	1.81
WAIT	1	1.81
RESET	1	100 msec
IOT, EMT, TRAP, BPT	5	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 7.7 μsec for MOS.

NPR (DMA) latency, which is the time from request to bus mastership for the first NPR device, is 2.5 μsec, max.

NOTES

1. Add 0.84 μseconds when in rounding mode (FT = 0).
2. Add 0.24 μseconds per shift to align binary points and 0.24 μseconds per shift for normalization. The number of alignment shifts is equal to the exponent difference for exponent differences bounded as follows:

   $1 \leq |EXP\ (AC) — EXP\ (FSRC)| \leq 24$ single precision
   $1 \leq |EXP\ (AC) — EXP\ (FSRC)| \leq 56$ double precision

   The number of shifts required for normalization is equivalent to the number of leading zeroes of the result.
3. Add .24 μseconds times the exponent of the product if the exponent of the product is:

   $1 \leq EXP\ (PRODUCT) \leq 24$ single-precision
   $1 \leq EXP\ (PRODUCT) \leq 56$ double-precision

   Add 0.24 μseconds per shift for normalization of the fractional result. The number of shifts required for normalization is equivalent to the number of leading zeroes in the fractional result.
4. Add 0.24 μseconds per shift for normalization of the integer being converted to a floating point number. For positive integers, the number of shifts required to normalize is equivalent to the number of leading zeroes; for negative integers, the number of shifts required for normalization is equivalent to the number of leading ones.
5. Add 0.24 μseconds per shift to convert the fraction and exponent to integer form, where the number of shifts is equivalent to 16 minus the exponent when converting to short integer or 32 minus the exponent when converting to long integer for exponents bounded as follows:

   $1 \leq EXP\ (AC) \leq 15$ short integer
   $1 \leq EXP\ (AC) \leq 31$ long integer

B.3 PDP-11/44 CENTRAL PROCESSOR

Timing for the instructions assumes the following conditions:

1. Times specified are typical and may vary by ±10%. They apply to both byte and word instructions, whether odd or even byte.
2. Timing is given without regard to NPR or BR servicing and assumes that no service states are used except where explicitly forced by the microstructures.
3. Cache times assume 100% hits. Non-cache times assume 0% hits.
4. If memory management is used, add 0.09 μsec per memory to the instruction time.
5. The memory timing is assumed to be the following:

MS11-M	DATI (P)	490 ns taa
	DATO (B)	230 ns taa

6. All times are expressed in μsec.

MOV, CMP, BIT, BIS, BIC, ADD, SUB

REGISTER TO REGISTER INSTRUCTION TIMES

INST	UNCACHED TIME	SMØ CACHED TIME	DMØ # MEM CYCLE
MOV (0,0)	1.23	.60	1
ADD, BIS, BIC (0 0) CMP, BIT, SUB	1.41	.78	1

For the following instructions, use time indicated for any combination other than register to register.

To figure time, add SRC time from the first table to DST time from the second table for the appropriate instruction.

SRC MODE TIMES FOR ALL INSTRUCTIONS LISTED (INCLUDING FETCH)

SRC MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.23	.64	1
1	1.92	.66	2
2	2.10	.84	2
3	2.97	1.08	3
4	2.10	.84	2
5	2.97	1.08	3
6	3.15	1.26	3
7	4.02	1.50	4

MOV DST MODE TIMES

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	.18	.18	0
1	.77	.77	0

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
2	.77	.77	0
3	1.82	1.19	1
4	.95	.95	0
5	1.82	1.19	1
6	2.00	1.37	1
7	2.87	1.60	2

ADD, BIS, BIC

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	.36	.36	0
1	1.46	.83	1
2	1.64	1.01	1
3	2.51	1.25	2
4	1.64	1.01	1
5	2.51	1.25	2
6	2.69	1.43	2
7	3.56	1.67	3

CMP, BIT

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	.36	.36	0
1	1.05	.42	1

INST	UNCACHED TIME	CACHED TIME	# MEM CYCLE
2	1.23	.60	1
3	2.10	.84	2
4	1.23	.60	1
5	2.10	.84	2
6	2.28	1.02	2
7	3.15	1.26	3

SUB

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	.18	.18	0
1	1.46	.83	1
2	1.64	1.01	1
3	2.51	1.25	2
4	1.64	1.01	1
5	2.51	1.25	2
6	2.69	1.43	2
7	3.56	1.67	3

For the following instructions, use time indicated directly.

XOR, NEG

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.59	.96	1
1	2.69	1.43	2

SRC MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
2	2.89	1.61	2
3	3.74	1.85	3
4	2.87	1.61	2
5	3.74	1.85	3
6	3.92	2.03	3
7	4.79	2.27	4

CLR, COM, INC, DEC, SBL, ADL, SXT

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.23	.60	1
1	2.51	1.25	2
2	2.69	1.43	2
3	3.56	1.67	3
4	2.69	1.43	2
5	3.56	1.67	3
6	3.74	1.85	3
7	4.61	2.09	4

TST

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.23	.60	1

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	2.60	.84	2
2	2.28	1.02	2
3	3.15	1.26	3
4	2.28	1.02	2
5	3.15	1.26	3
6	3.33	1.44	3
7	4.20	1.68	4

ROL, ROR, ASR, ASL

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.59	.96	1
1	2.69	1.43	2
2	2.87	1.61	2
3	3.74	1.85	3
4	2.87	1.61	2
5	3.74	1.85	3
6	3.92	2.03	3
7	4.79	2.27	4

SWAB

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.41	.78	1

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	2.51	1.25	2
2	2.69	1.43	2
3	3.56	1.67	3
4	2.69	1.43	2
5	3.56	1.67	3
6	3.74	1.85	3
7	4.61	2.09	4

MFPI (D)

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	2.18	1.55	1
1	3.23	1.97	2
2	3.41	2.15	2
3	4.10	2.21	3
4	3.41	2.15	2
5	4.10	2.21	3
6	4.28	2.39	4
7	5.15	2.64	4

MTPI (D)

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	2.64	1.38	2

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	3.59	2.26	2
2	3.27	2.51	2
3	4.46	2.57	3
4	3.27	2.51	2
5	4.46	2.57	3
6	4.64	2.75	3
7	5.51	2.99	4

JMP

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	1.23	.60	1
2	1.59	.96	1
3	2.28	1.02	2
4	1.41	.78	1
5	2.28	1.02	2
6	2.28	1.02	2
7	3.33	1.44	3

JSR

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	2.47	1.91	1
2	2.65	2.09	1

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
3	3.34	2.15	2
4	2.47	1.91	1
5	3.34	2.15	2
6	3.52	2.40	2
7	4.39	2.57	3

CALL TO SUPERVISOR MODE

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	7.46	6.20	2
1	8.15	6.44	3
2	8.33	6.45	3
3	9.20	6.68	4
4	8.33	6.44	3
5	9.20	6.68	4
6	9.38	6.86	4
7	10.25	7.10	5

BRANCHES

TYPE BNE, ETC.		UNCACHED	CACHED	# MEM CYCLE
	FAILED	1.05	.42	1
	PASSED	1.59	.96	1
SOB	NO BRANCH	1.41	.78	1
	BRANCH	1.77	1.14	1

TRAP, SUBROUTINES

		UNCACHED	CACHED	# MEM CYCLE
TRAP INST.		5.68	3.93	3
RTS		2.46	1.20	2
RTI, RTT	NOT KERNEL	3.61	1.92	3
	KERNEL	4.35	2.46	3

MISCELLANEOUS

	UNCACHED	CACHED	# MEM
SET, CLR CC's	1.41	.78	(1)
WAIT (LOOP)	1.53	.90	(1)
(EXIT)	5.56	3.67	(3)
RESET (NOP)	1.23	.60	1
	90 μs In Kernel Mode		
MARK	3.36	2.10	2
MFPT	1.41	.78	1
SPL	2.85	2.22	1

EIS

ASH DM	0	3.93	1	3.30	ADD 180 ns.
	1	4.62	2	3.36	FOR TRANS.
	2	4.80	2	3.54	FOR RIGHT SHIFTS
	3	5.67	2	3.78	SUBTRACT 600ns.
	4	4.80	2	3.36	
	5	5.60	3	3.78	
	6	5.78	3	3.89	
	7	6.65	4	4.13	

ASHC DM	0	3.51	1	2.88	ADD 180 ns.
	1	4.20	2	2.94	FOR TRANS.
	2	4.38	2	3.12	
	3	5.25	3	3.36	
	4	4.38	2	3.12	
	5	5.25	3	3.36	
	6	5.43	3	3.54	
	7	6.30	4	3.78	
MUL DM	0	6.63	1	6.00	ADD 180 ns. PER
	1	7.32	2	6.06	BIT TRANSITION
	2	7.50	2	6.24	
	3	8.37	3	6.48	
	4	7.50	2	6.24	
	5	8.37	3	6.48	
	6	8.55	3	6.66	
	7	9.42	4	6.90	
DIV DM	0	11.01	1	10.28	
	1	11.07	2	10.44	
	2	11.88	2	10.62	
	3	12.75	3	10.86	
	4	11.88	1	10.62	
	5	12.75	3	10.86	
	6	12.93	3	11.04	
	7	13.08	4	11.28	

B.4 PDP-11/70 CENTRAL PROCESSOR

The execution time for an instruction depends on the instruction itself, the modes of addressing used, and the type of memory being referenced. In the most general case, the Instruction Execution Time is the sum of a Source Address Time, and an Execute, Fetch Time.

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Some of the instructions require only some of these times, and are so noted. Times are typical; processor timing, with core memory, may vary +15% to −10%.

BASIC INSTRUCTION SET TIMING

Double Operand

all instructions,
except MOV: Instr Time = SRC Time + DST Time
(but including MOVB) + EF Time
MOV Instruction: Instr Time = SRC Time + EF Time
(word only)

Single Operand

all instructions: Instr Time = DST Time + EF Time or
Instr Time = SRC Time + EF Time

Branch, Jump, Control, Trap & Misc

all instructions: Instr Time = EF Time

USING THE CHART TIMES

To compute a particular instruction time, first find the instruction "EF" Time. Select the proper EF Time for the SRC and DST modes. Observe all "NOTES" to the EF Time by adding the correct amount to basic EF number.

Next, note whether the particular instruction requires the inclusion of SRC and DST Times, if so, add the appropriate amounts to correct EF number.

USING THE CHART TIMES

The times given in the chart for Cache "hits"; that is, all the read cycles are assumed to be in the Cache. The number of read cycles in each subset of the instruction is also included so that timing can be calculated for a specific case of hits and misses, or timing can be calculated based on an average hit rate.

a) Specific hits and misses
Add 1.02 μsec for each read cycle which is a miss instead of a hit.

b) Average hit rate
If P_H is the percent of reads that are hits, add $1.02 \times (1 - P_H) \times$ (Number of read cycles) to the instruction timing.

For example, an ADD A,B instruction using Mode 6 (indexed) address modes:

1) All Hits:

SRC time	= 0.60 μsec	2 read cycles
DST time	= 0.60 μsec	2 read cycles
EF time	= 1.35 μsec	1 read cycle
TOTAL	= 2.55 μsec	5 read cycles

2) 4 Hits, 1 Miss
Total = 2.55 + 1.02
= 3.57 μsec

3) Read hit rate of 90%
Total = 2.55 + (1.02) (.1) (5)
= 3.06 μsec

NOTES

1. The times specified generally apply to Word instructions. In most cases Even Byte instructions have the same time, with some Odd Byte instructions taking longer. All exceptions are noted.
2. Timing is given without regard for NRP or BR serving. Core memory is assumed to be iocated within the first 128K memory unit.

3. Times are not affected if Memory Management is enabled.
4. All times are in microseconds.

SOURCE ADDRESS TIME

Instruction	Source Mode	SRC Time	Read Memory Cycles
Double Operand	0	.00	0
	1	.30	1
	2	.30	1
	3	.75	2
	4	.45	1
	5	.90	2
	6	.60	2
	7	1.05	3

DESTINATION ADDRESS TIME

Instruction	DST Mode	DST Time (A)	Read Memory Cycles
Single Operand and Double Operand (except MOV, MTPI, MTPD, JMP, JRS	0	.00	0
	1	.30	1
	2	.30	1
	3	.75	2
	4	.45	1
	5	.90	2
	6	.60	2
	7	1.05	3

NOTE (A): Add .15 μsec for odd byte instructions, except DST Mode 0.

EXECUTE, FETCH TIME

Double Operand

Instruction (Use with SRC Time and DST Time)	EF Time (SRC Mode 0) (DST Mode 0)	Read Mem Cyc	EF Time (SRC Mode 1-7) (DST Mode 0)	Read Mem Cyc	EF Time (SRC Mode 0-7) (DST Mode 1-7)	Read Mem Cyc
ADD, SUB, BIC, BIS MOVB	.30 (D)	1	.45 (D)	2	1.20 (C)	1
CMP, BIT	.30 (D)	1	.45 (D)	1	.45 (C)	1
XOR	.30 (D)	1	.30 (D)	1	1.20	1

NOTE (C): Add 0.15 μsec if SRC is R1 to R7 and DST is R6 or R7.
NOTE (D): Add 0.3 μsec if DST is R7.

Instruction (Use with SRC Time)	DST Mode	DST Register	EF Time (SRC Mode = 0)	EF Time (SRC Mode = 1-7)	Read Memory Cycles
	0	0-6	.30	.45	1
	0	7	.60	.75	1
	1	0-7	1.20	1.20	1
	2	0-7	1.20	1.20	1
MOV	3	0-7	1.65	1.65	2
	4	0-7	1.35	1.35	1
	5	0-7	1.80	1.80	2
	6	0-7	1.50	1.65	2
	7	0-7	1.95	2.10	3

Single Operand

Instruction (Use with DST Time)	EF TIME (DST Mode = 0)	Memory Cycles	EF Time (DST Mode 1 to 7)	Read Memory Cycles
CLR, COM, INC, DEC, ADC, SBC, ROL, ASL, SWAB, SXT	.30 (J)	1	1.20	1
NEG	.75	1	1.50	1
TST	.30 (J)	1	.45	1
ROR, ASR	.30 (J)	1	1.20 (H)	1
ASH, ASHC	.75 (I)	1	.90 (I)	1

NOTE (H): Add 0.15 μsec if odd byte.
NOTE (I): Add 0.15 μsec per shift.
NOTE (J): Add 0.30 μsec if DST is R7.

Instruction (Use with SRC Times)	EF Time	Read Memory Cycles
MUL	3.30	1
DIV		
by zero	.90	1
shortest	7.05	1
longest	8.55	1

Instruction	EF Time	Read Memory Cycles	
MFPI	1.50	1	use with SRC times
MFPD	1.50	1	

Instruction	DST Mode	Instruction Time	Read Memory Cycles
MTPI	0	.90	1
MTPD	1	1.65	2
	2	1.65	2
	3	2.10	3
	4	1.80	2
	5	2.25	3
	6	2.10	3
	7	2.55	4

Branch Instructions

Instruction	Instr Time (Branch)	Instr Time (No Branch)	Read Memory Cycles
BR, BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLOS, BHIS, BLO	.60	.30	1
SOB	.60	.75	1

Jump Instructions

Instruction	DST Mode	Instr Time	Read Memory Cycles
JMP	1	.90	1
	2	.90	1
	3	1.20	2
	4	.90	1
	5	1.35	2
	6	1.05	2
	7	1.50	3
JSR	1	1.95	1
	2	1.95	1
	3	2.25	2
	4	1.95	1
	5	2.40	2
	6	2.10	2
	7	2.55	3

Control, Trap & Miscellaneous Instructions

Instruction	Instr Time	Read Memory Cycles
RTS	1.05	2
MARK	.90	2
RTI, RTT	1.50	3
SET N, Z, V. C		
CLR, N, Z, V, C	.60	1
HALT	1.05	0
WAIT	.45	0
WAIT Loop for a BR is .3 μsec.		
RESET	10ms	1
IOT, EMT, TRAP, BRT	3.30	3
SPL	.60	1
INTERRUPT First Device	2.31	2

EFFECTIVE MEMORY CYCLE TIME

The overall effective cycle time of the CPU can be calculated from the following formula:

$$TC_E = P_R \times [(P_H \times TC_H) + (1 - P_H)\, TC_M] + (1 - P_R)\, TC_W$$

Where TC_E = Effective cycle time
TC_H = Cycle time for a read hit = 0.30 μsec
TC_M = Cycle time for a read miss = 1.32 μsec
TC_W = Cycle time for a write = 0.75 μsec
P_R = Percent of cycles that are reads
P_H = Percent of reads that are hits

Thus, for an average PDP-11/70 program which has a read rate of 91% and a read hit rate of 93%, the effective cycle time is:

$TC_E = .91 \times [(.93 \times .30) + (.07 \times 1.32)] + (.09 \times .75) = .41$ μsec

APPENDIX C
FLOATING POINT TIMING

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-F. Timings for the KEF11-AA (PDP-11/24) were not available at the time of publication. Also, the FP11-A (PDP-11/34a) timings utilizing MOS MS11-L memory were not available. Both timing sets will be available in the next PDP-11 processor handbook.

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

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

Operation (register-to-register)	11/34A FP11-A	11/70 FP11-C	11/44 FP11-F
Single Precision			
Add/Subtract	8.91	1.65	8.91
Multiply	16.2	3.27	16.2
Divide	16.2	4.29	16.2
Double Precision			
Add/Subtract	8.91	1.68	8.91
Multiply	25.36	5.43	25.36
Divide	35.36	6.73	35.36

FLOATING POINT INSTRUCTION TIMING: FP11-A

Instruction Execution Time

The execution time of an FP11-A floating point instruction is dependent on the following conditions:

- type of instruction
- type of addressing mode specified
- type of memory
- memory management facility enabled or disabled

Additionally, the execution time of certain instructions, such as ADD, is dependent on the data.

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

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

NOTE

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

Table C-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 C-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 C-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 C-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 C-6

Instr.	Mode 0 (Reg. to Reg.)	Notes	Modes 1 thru 7
LDCIF	7.5	1,4	to determine
LDCID	7.5	1,4	memory-to-register times
LDCLF	7.5	1,4	for these instructions
LDCLD	7.5	1,4	
STFPS	2.8		
STST	2.6		Use Table C-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 C-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 C-4 Floating Destination Store Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	4	1.38	2.94
2	2	4	1.56	3.12
2 Immediate	1	1	0.60	0.60
3	3	5	2.38	3.94
4	2	4	1.56	3.12
5	3	5	2.38	3.94
6	3	5	2.56	4.12
7	4	6	3.56	5.12

Table C-5 Floating Destination Fetch And Store Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	2	1.42	1.42
2	2	2	1.60	1.60
2 Immediate	2	2	1.60	1.60
3	3	3	2.42	2.42
4	2	2	1.60	1.60
5	3	3	2.60	2.60
6	3	3	2.60	2.60
7	4	4	3.60	3.60

Table C-6 Source Fetch Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	1	2	1.00	1.18
2	1	2	1.18	1.36
2 Immediate	1	1	1.18	1.18
3	2	3	2.00	2.18
4	1	2	1.18	1.36
5	2	3	2.00	2.18
6	2	3	2.18	2.36
7	3	4	3.18	3.36

Table C-7 Destination Store Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	1	2	0.60	1.38
2	1	2	0.96	1.68
2 Immediate	1	1	0.96	0.96
3	2	3	1.60	2.38
4	1	2	0.96	1.68
5	2	3	1.60	2.38
6	2	3	1.78	2.56
7	3	4	2.78	3.56

NOTES:

- Add 0.84 μsec when in rounding mode (FT = 0).
- Add 0.24 μsec per shift to align binary points and 0.24 μsec per shift for normalization. The number of alignment shifts is equal to the exponent difference for exponent differences bounded as follows:

  $1 \leq |EXP(AC) - EXP(FSRC)| \leq 24$ single precision
  $1 \leq |EXP(AC) - EXP(FSRC)| \leq 56$ double precision

The number of shifts required for normalization is equivalent to the number of leading zeros of the result.

- Add .24 μsec times the exponent of the product if the exponent of the product is:

  $1 \leq$ EXP (PRODUCT) ≤ 24 single precision
  $1 \leq$ EXP (PRODUCT) ≤ 56 double precision

  Add 0.24 μsec per shift for normalization of the fractional result. The number of shifts required for normalization is equivalent to the number of leading zeros in the fractional result.
- Add 0.24 μsec per shift for normalization of the integer being converted to a floating point number. For positive integers, the number of shifts required to normalize is equivalent to the number of leading zeros; for negative integers, the number of shifts required for normalization is equivalent to the number of leading ones.
- Add 0.24 μsec per shift to convert the fraction and exponent to integer form, where the number of shifts is equivalent to 16 minus the exponent when converting to short integer or 32 minus the exponent when converting to long integer for exponents bounded as follows:

  $1 \leq$ EXP (AC) ≤ 15 short integer
  $1 \leq$ EXP (AC) ≤ 31 long integer

FLOATING POINT INSTRUCTION TIMING: FP11-C

Floating point instruction times are calculated in a manner similar to the calculation of CPU instruction timing. Since the FP11-C is a separate processor operating in parallel with the main processor, however, the calculation of floating point instruction times must take this parallel processing or overlap into account. The following is a description of the method used to calculate the effective floating point instruction execution times.

TERM	DEFINITION
Instruction Decode Preinteraction Time	CPU time required to decode a floating point instruction opcode and to store the general register referred to in the floating point instruction in a temporary floating point register (FPR). This time is fixed at 450 ns.

TERM	DEFINITION
Address Calculation Time	CPU time required to calculate the address of the operand. This time is dependent on the addressing mode specified. Refer to Table C-8.
Wait Time	CPU time spent waiting for completion by the floating point processor of a previous floating point instruction, in the case of load class instructions. For store class instructions, the wait time is the sum of time during which the floating point completes a previous floating point instruction and floating point execution time for the store class instruction. Wait time is calculated as follows:
(Load Class Instructions)	Wait time = [floating point execution time (previous FP instruction)] − [disengage and fetch time (previous FP instruction)] − [CPU execution time for interposing non-floating point instruction] − [preinteraction time] — [address calculation time]. If the result is ≤ 0, the wait time is zero.
(Store Class Instructions)	Wait time = [floating point execution time (previous floating point instruction)] − [CPU execution time for interposing non-FP instruction] − [disengage and fetch time (previous FP instruction)] − [preinteraction] + [floating point execution time] — [address calculation time]. If the result is ≤ 0, the wait time is zero.
Resync Time	If the CPU must wait for the floating point processor (i.e., wait time = 0), an additional 450 ns must be added to the effective execu-

TERM	DEFINITION
	tion time of the instruction. If wait time = 0, then resync time =0.
Interaction Time	CPU time required actually to initiate floating point processor operation.
Argument Transfer Time	CPU time required to fetch and transfer to the floating point processor the required operand. This time is 300 ns × the number of 16-bit words read from memory (load class floating point instructions), or 1200 ns × the number of 16-bit words written to memory (store class instructions).
Disengage and Fetch Time	CPU time required to fetch the next instruction from memory. This time is 300 ns.
Floating Point Execution Time	Time required by the floating point processor to complete a floating point instruction once it has received all arguments (load class instructions). Execution times are contained in Tables C-2 through C-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 C-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 C-9 FP11-C Execution Times

Instruction	Minimum nsec	Maximum nsec	Typical
LDF	360	360	
LDD	360	360	
ADDF	900	2520	950
ADDD	900	4140	980
SUBF	900	1980	1130
SUBD	900	4140	1160
MULF	1800	3440	2520
MULD	3060	6220	4680
DIVF	1920	6720	3540
DIVD	3120	14400	6000
MODF	2880	5990	
MODD	3780	9770	
LDCFD	420	420	
LDCDF	540	540	
STF*	0		
STD*	0		
CMPF	540	1080	
CMPD	540	1080	
STCFD*	720	720	720
STCDF*	540	720	540
LDCIF	1260	1440	1440
LDCID	1260	1440	1440

Instruction	Minimum nsec	Maximum nsec	Typical
LDCLF	1260	1980	
LDCLD	1260	1980	
LDEXP	540	900	
STCFI*	1260	1620	
STCFL*	1260	2160	
STCDI*	1260	1620	
STCDL*	1260	2160	
STEXP*	360	360	
	MO	Not MO	
CLRD	180	2150	
CLRD	180	14350	
NEGF	360	2400	
NEGD	360	2400	
ABSF	360	2400	
ABSD	360	2400	
TSTF	180	180	
TSTD	180	180	
LDFPS	180	0	
STFPS*	0		
STST*	0		
CFCC	0		
SETF	180		
SETD	180		
SETI	180		
SETL	180		

* Store Class Instructions

Load class instructions are those which do not deposit results in a memory location.

Execution of a load class floating point instruction by the floating point occurs in parallel with CPU operation and can be overlapped. Figure C-2 gives a simplified picture of how a load class floating point instruction is executed.

Store class instructions are those which store a result from the floating point into a memory location. Execution of a store class instruction by the floating point processor must occur before the result can be stored, hence parallel processing cannot occur for store class floating point instructions.

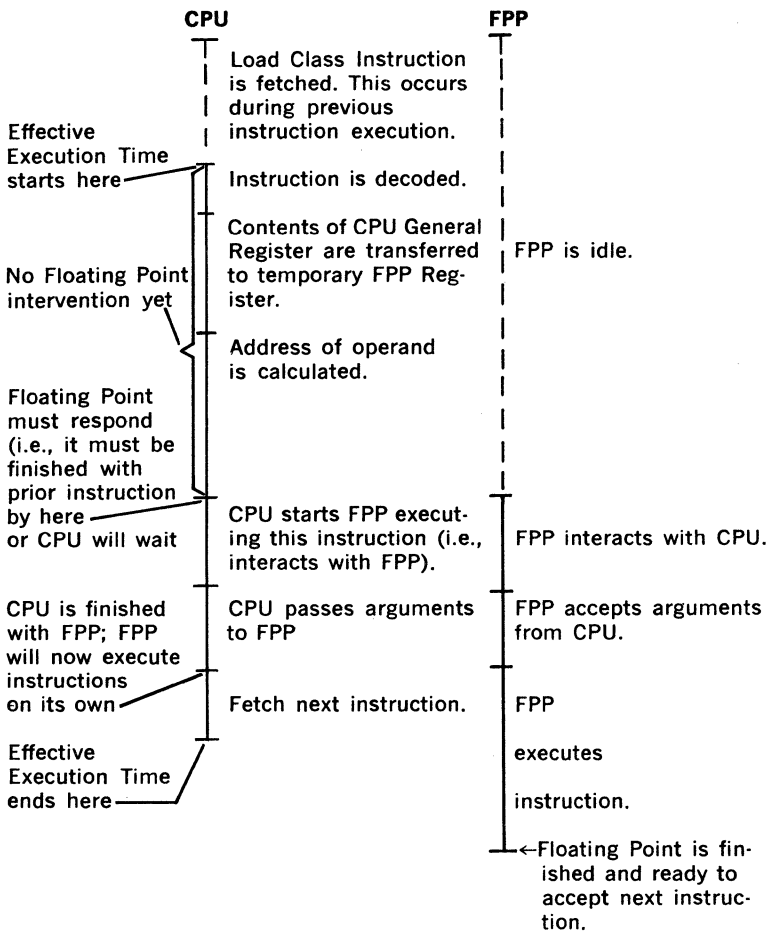


Figure C -2 Load Class Floating Point Instruction

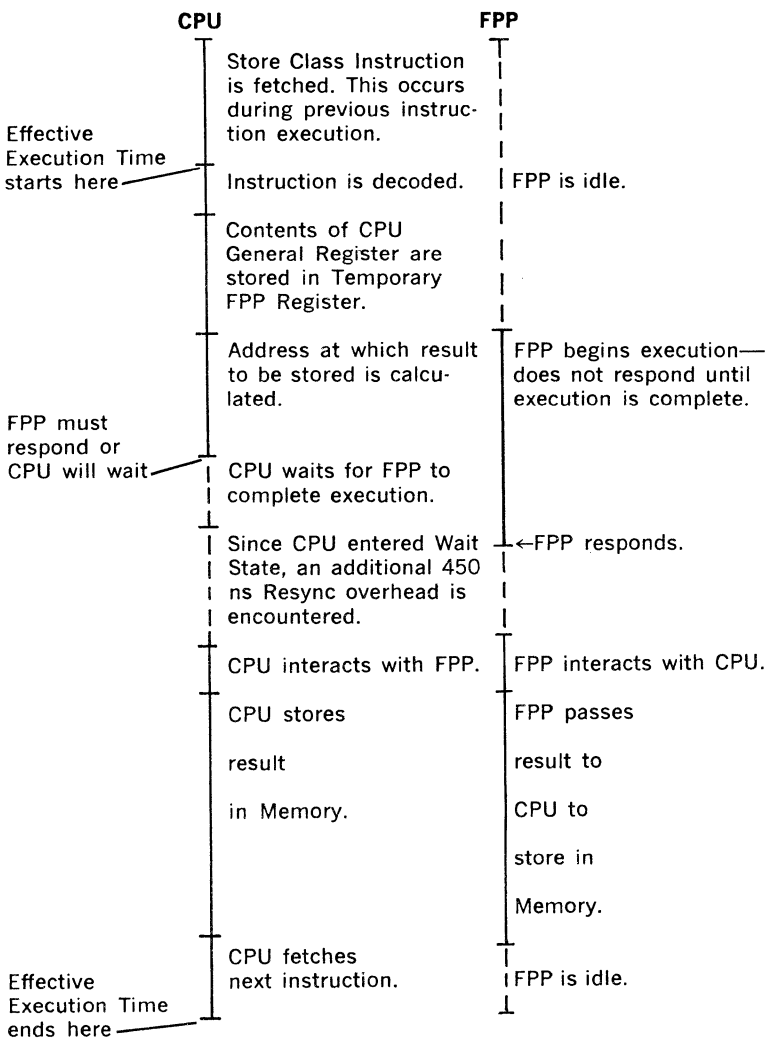


Figure C-3 Store Class Floating Instruction

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

Figure C-4 shows 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 C-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 C-8)	300 ns
Wait Time (Since FPP is idle, Wait = 0)	0 ns
Resync Time (Since Wait = 0, Resync = 0)	0 ns
Interaction Time	300 ns
Argument Transfer Time (Transfer 2 words @ 300 ns/word)	600 ns
Disengage and Fetch Time	300 ns
Effective Execution Time	1950

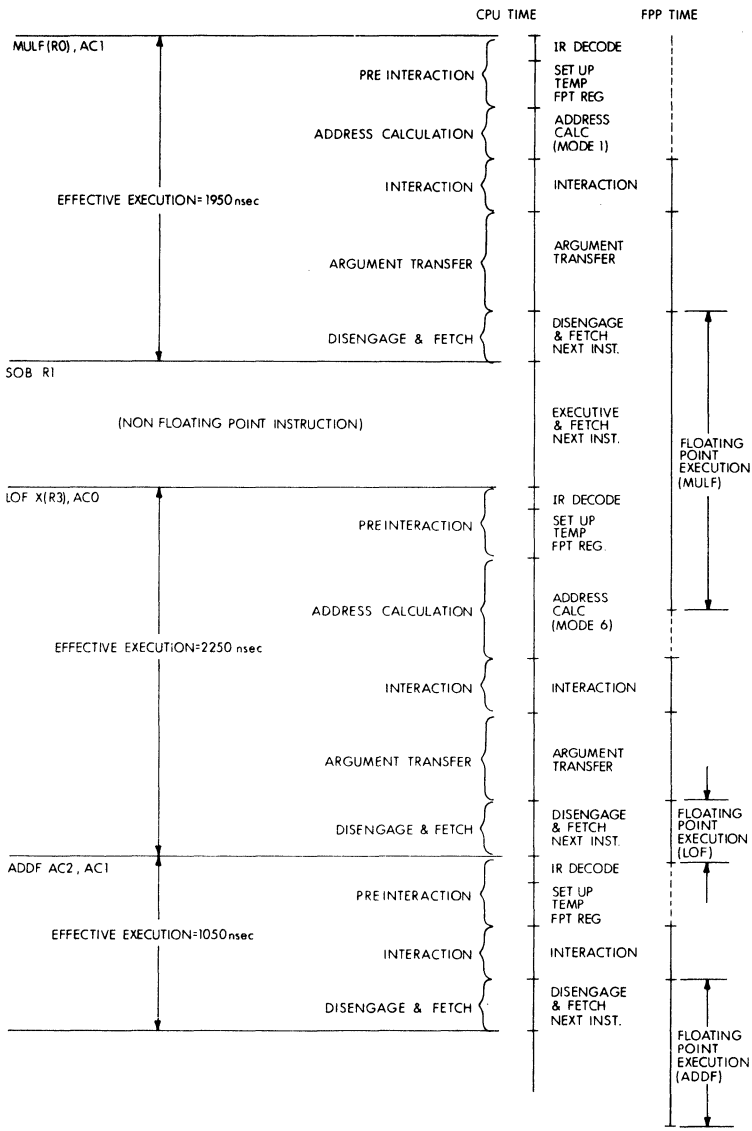
For LDF X(R3),ACLO (Ref. Figure C-4), first we calculate Wait Time:

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

Since calculation resulted in a negative number, Wait Time = 0.

...so effective execution time is the summation of the following:

Preinteraction Time	450 ns
Address Calculation Time (Mode 6 from Table C-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



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

FLOATING POINT INSTRUCTION TIMING: FP11-F

Instruction Execution Time

The execution time of an FP11-F floating point instruction is dependent on the following conditions:

- type of instruction
- type of addressing mode specified
- type of memory
- memory management facility enabled or disabled

Additionally, the execution time of certain instructions, such as ADD, is dependent on the data.

Table C-10 provides the basic instruction times for mode 0. Tables C-11 through C-15 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 C-10 which gives MULF, mode 0, execution time of 12.4 μsec.
2. Refer to Note 1 as specified in the notes column of Table C-10. Note 1 specifies an additional 0.84 μsec is to be added if rounding mode is specified. This yields 13.24 μsec.
3. The Modes 1 through 7 column of Table C-10 refers to Table C-11 to determine the additional time required for mode 1 through 7 instructions. In this example, mode 3 specifies an additional 3 μsec for single precision yielding 16.24 μsec.

All timing information is in microseconds unless otherwise noted. Times are typical; processor timing can vary ±10%. All instructions assume 100% cache hits.

NOTE

Add .090 μsec for each DATI memory cycle if memory management is enabled.
Add .630 μsec for each DATI memory cycle if a cache miss is encountered.

Table C-10 FP11-F Instruction Execution Times

Instr.	Mode 0 (Reg. to Reg.)	Notes	Modes 1 thru 7
LDF	3.0		
LDD	3.0		
LDCFD	4.8	1	
LDCDF	4.8	1	
CMPF	4.5		
CMPD	4.5		
DIVF	12.3	1	Use Table C-11
DIVD	19.6	1	to determine
ADDF	6.5	1,2	memory-to-register times
ADDD	6.5	1,2	for these instructions
SUBF	6.9	1,2	
SUBD	6.9	1,2	
MULF	12.4	1	
MULD	19.7	1	
MODF	16.4	1,3	
MODD	23.7	1,3	
STF	1.4		
STD	1.4		Use Table C-12
STCDF	4.2		to determine
STCFD	4.2		memory-to-register times
CLRF	1.6		for these instructions
CLRD	1.6		
ABSF	2.5		
ABSD	2.5		Use Table C-13
NEGF	2.6		to determine
NEGD	2.6		memory-to-memory times
TSTF	2.6		for these instructions
TSTD	2.6		
LDFPS	1.5		
LDEXP	3.4		Use Table C-14
LDCIF	6.5	1,4	to determine
LDCID	6.5	1,4	memory-to-register times
LDCLF	6.5	1,4	for these instructions
LDCLD	6.5	1,4	

Instr.	Mode 0 (Reg. to Reg.)	Notes	Modes 1 thru 7
STFPS	1.8		
STST	1.6		Use Table C-15
STEXP	2.4		to determine
STCFI	3.5	5	register-to-memory times
STCDI	3.5	5	for these instructions
STCFL	3.5	5	
STCDL	3.5	5	
The following instructions do not reference memory			
CFCC	1.0		
SETF	1.2		
SETD	1.2		Execution times
SETI	1.2		are as shown.
SETL	1.2		

Table C-11 Floating Source Fetch Time

	Memory Cycles		Time (μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	4	0.60	1.4
2	2	4	0.80	1.6
2 Immediate	1	1	0.30	0.3
3	3	5	0.90	1.7
4	2	4	0.80	1.6
5	3	5	0.90	1.7
6	3	5	1.10	1.9
7	4	6	1.40	2.2

Table C-12 Floating Destination Store Time

	Memory Cycles		Time (μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	4	1.38	2.94
2	2	4	1.56	3.12
2 Immediate	1	1	0.60	0.60
3	3	5	1.68	3.24
4	2	4	1.56	3.12
5	3	5	1.68	3.24
6	3	5	1.86	3.42
7	4	6	2.16	3.72

Table C-13 Floating Destination Fetch And Store Time

	Memory Cycles		Time (μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	2	0.72	0.72
2	2	2	0.90	0.90
2 Immediate	2	2	0.80	0.80
3	3	3	1.02	1.02
4	2	2	0.90	0.90
5	3	3	1.20	1.20
6	3	3	1.20	1.20
7	4	4	1.50	1.50

Table C-14 Source Fetch Time

	Memory Cycles		Time (μs)	
Addressing Mode	Short Integer	Long Integer	Short Integer	Long Integer
1	1	2	0.30	0.70
2	1	2	0.48	1.28
2 Immediate	1	1	0.48	0.48
3	2	3	0.60	1.0
4	1	2	0.48	1.28
5	2	3	0.60	1.0
6	2	3	0.78	1.18
7	3	4	1.08	1.48

Table C-15 Destination Store Time

	Memory Cycles		Time (μs)	
Addressing Mode	Short Integer	Long Integer	Short Integer	Long Integer
1	1	2	0.60	1.38
2	1	2	0.96	1.68
2 Immediate	1	1	0.96	0.96
3	2	3	0.90	1.68
4	1	2	0.96	1.68
5	2	3	0.90	1.68
6	2	3	1.08	1.86
7	3	4	1.38	2.16

NOTES:

1. Add 0.84 μsec when in rounding mode (FT = 0).
2. Add 0.24 μsec per shift to align binary points and 0.24 μsec per shift for normalization. The number of alignment shifts is equal to the exponent difference for exponent differences bounded as follows:

   $1 \leq EXP(AC) - EXP(FSRC) \leq 24$, single precision
   $1 \leq EXP(AC) - EXP(FSRC) \leq 56$, double precision

The number of shifts required for normalization is equivalent to the number of leading zeros of the result.

3. Add 0.24 μsec times the exponent of the product if the exponent of the product is:

   $1 \leq$ EXP(PRODUCT) ≤ 24, single precision
   $1 \leq$ EXP(PRODUCT) ≤ 56, double precision

   Add 0.24 μsec per shift for normalization of the fractional result. The number of shifts required for normalization is equivalent to the number of leading zeros in the fractional result.

4. Add 0.24 μsec per shift for normalization of the integer being converted to a floating point number. For positive integers, the number of shifts required to normalize is equivalent to the number of leading zeros; for negative integers, the number of shifts required for normalization is equivalent to the number of leading ones.

5. Add 0.24 μsec per shift to convert the fraction and exponent to integer form, where the number of shifts is equivalent to 16 minus the exponent when converting to short integer, or 32 minus the exponent when converting to long integer for exponents bounded as follows:

   $1 \leq$ EXP(AC) ≤ 15, short integer
   $1 \leq$ EXP(AC) ≤ 31, long integer

APPENDIX D
CONVERSION TABLE

Decimal	Octal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001
10	12	1010
11	13	1011
12	14	1100
13	15	1101
14	16	1110
15	17	1111
16	20	10000
17	21	10001
18	22	10010
19	23	10011
20	24	10100
21	25	10101
22	26	10110
23	27	10111
24	30	11000
25	31	11001
26	32	11010
27	33	11011
28	34	11100
29	35	11101
30	36	11110
31	37	11111
32	40	100000
33	41	100001
34	42	100010
35	43	100011

Decimal	Octal	Binary
36	44	100100
37	45	100101
38	46	100110
39	47	100111
40	50	101000
41	51	101001
42	52	101010
43	53	101011
44	54	101100
45	55	101101
46	56	101110
47	57	101111
48	60	110000
49	61	110001
50	62	110010
51	63	110011
52	64	110100
53	65	110101
54	66	110110
55	67	110111
56	70	111000
57	71	111001
58	72	111010
59	73	111011
60	74	111100
61	75	111101
62	76	111110
63	77	111111
64	100	1000000
65	101	1000001
66	102	1000010
67	103	1000011
68	104	1000100
69	105	1000101
70	106	1000110
71	107	1000111
72	110	1001000
73	111	1001001
74	112	1001010
75	113	1001011
76	114	1001100
77	115	1001101

Decimal	Octal	Binary
78	116	1001110
79	117	1001111
80	120	1010000
81	121	1010001
82	122	1010010
83	123	1010011
84	124	1010100
85	125	1010101
86	126	1010110
87	127	1010111
88	130	1011000
89	131	1011001
90	132	1011010
91	133	1011011
92	134	1011100
93	135	1011101
94	136	1011110
95	137	1011111
96	140	1100000
97	141	1100001
98	142	1100010
99	143	1100011
100	144	1100100
101	145	1100101
102	146	1100110
103	147	1100111
104	150	1101000
105	151	1101001
106	152	1101010
107	153	1101011
108	154	1101100
109	155	1101101
110	156	1101110
111	157	1101111
112	160	1110000
113	161	1110001
114	162	1110010
115	163	1110011
116	164	1110100
117	165	1110101
118	166	1110110
119	167	1110111

Decimal	Octal	Binary
120	170	1111000
121	171	1111001
122	172	1111010
123	173	1111011
124	174	1111100
125	175	1111101
126	176	1111110
127	177	1111111
128	200	10000000

aborts
clearing status registers after 168 to 169
Fault Recovery Registers and 161 to 162
Memory Management Register #0 and 164 to 165
parity and 284 to 285

ABSD floating point instruction 320 to 321

ABSF floating point instruction 320 to 321

absolute addressing mode 25, 33, 37, 40
position-independent coding and 95

absolute-value string data 351

access control field (ACF) 146

access information bits 147

accumulators 23
in Floating Point Processor 307, 308, 316, 320

accuracy, of floating point 316 to 318

ACF (access control field) 146

Active Page Field (APF) 150

Active Page Registers (APRs) 138, 142 to 147
virtual bus address and 150 to 152

ADCB instruction 57

ADC instruction 57

ADDD floating point instruction 321 to 322

ADDER console command 237

ADDF floating point instruction 321 to 322

ADD instruction 24, 57 to 58

ADDN(I) commercial instruction 367 to 369

ADDP(I) commercial instruction 367 to 369

address display lights 301

addresses
assignments of 208, 250
of I/O device registers 212, 256 to 257
mapping of 153 to 161
odd addressing errors in 117
physical address construction and 150 to 152
physical address space and 136
relocation of 136 to 139
specification of 200 to 202
stack limit 279
UNIBUS and 11
vitural address space and 135
see also vector address

addressing, by floating point instructions 315

addressing modes 23 to 40
for floating point 316
position-independent coding and 93 to 94

address registers, in Floating Point Processor 315 to 316

address select switch 300 to 301

address space
in PDP-11/70 processors 267
virtual 135, 150
see also physical address space

APF (Active Page Field) 150

APRs *see* Active Page Registers

/A qualifier 237

architecture
of Floating Point Processor 307 to 308
of PDP-11/24 systems 184
of PDP-11/44 systems 218, 225 to 226
of PDP-11/70 cache 280 to 282
of PDP-11/70 systems 261 to 262
UNIBUS and 11 to 13

ASCII consoles
for PDP-11/24 processors 192 to 193
for PDP-11/44 processors 232 to 233
ASCII conversions 123
ASHC instruction 58 to 59
ASH instruction 58
ASHN(I) commercial instruction 362, 369 to 371
ASHP(I) commercial instruction 362, 369 to 371
ASLB instruction 59 to 60
ASL instruction 59 to 60
ASRB instruction 60
ASR instruction 60
assembler language
conversion routines for 120
see also MACRO-11 assembler language
autodecrement addressing mode 25, 29, 35, 39
stacking and 98
autodecrement deferred addressing mode 25, 28 to 29, 36, 39
autoincrement addressing mode 25, 27 to 28, 35, 38
autoincrement deferred addressing mode 25, 29 to 30, 36, 38
availability *see* reliability, availability, and maintainability program

backing up 168
backplanes
on PDP-11/04 and PDP-11/34A processors 171 to 172, 178 to 180
on PDP-11/24 processors 191
on PDP-11/44 processors 224, 231
base addresses 136, 137
battery backups
in PDP-11/04 and PDP-11/34A processors 173
in PDP-11/24 processors 190, 210
in PDP-11/44 processors 223 to 224, 254
in PDP-11/70 processors 269
BBSY (Bus Busy) signal 14 to 15
BCC instruction 61
BCS instruction 61
BEQ instruction 61
BGE instruction 61
BGs (bus grants) 14
BGT instruction 62
BHI instruction 62 to 63
BHIS instruction 63
BICB instruction 63
BIC instruction 63
binary dump ODT command 199 to 200
BINARY LOAD console command 242
binary loading 243
binary number system 8
binary unloading 243
BISB instruction 63
BIS instruction 63
BITB instruction 63 to 64
BIT instruction 63 to 64
BLE instruction 64 to 65
Block Number (BN) 151
BLO instruction 65
BLOS instruction 65
BLT instruction 65 to 66
BMI instruction 66
BN (Block Number) 151
BNE instruction 66 to 67
BOOT console command 237
BOOT console emulator function 176

boot module (M9312) 177
bootstrap loaders
  on PDP-11/04 and PDP-11/34A processors 171, 177
  on PDP-11/70 processors 303 to 304
BPL instruction 67
BPT instruction 67
branch instructions 43, 46 to 47, 52 to 53, 350
break generation 209, 251
BR instruction 67
BRs (bus requests) 14
bus arbitrators 14
Bus Busy signal (BBSY) 14 to 15
bus cycles 13 to 14
buses
  communications on 13
  control of 14
  I/O 291
  see also UNIBUS
bus grants (BGs) 14
bus request levels 16
bus requests (BRs) 14
BVC instruction 67
BVS instruction 68

cabinets 272 to 273
cache data register (CDR) 226 to 229
cache hit register (CHR) 230 to 231
cache maintainance register (CMR) 229 to 230
cache registers 285 to 291
caches
  on PDP-11/34A processors 172, 173
  on PDP-11/44 processors 224 to 226
  on PDP-11/70 processors 261, 268, 280 to 283
calls, to coroutines 110 to 111
CARRIAGE RETURN (<CR>) command 235, 242
CARRIAGE RETURN (<CR>) command 196 to 197
C bit 49 to 50
/CB qualifier 236
CCC instruction 68
CDR (cache data register) 226 to 229
central processing units *see* CPUs
CFCC floating point instruction 322
character data types 346 to 348
characters 346 to 348
character searches 349
character sets 346 to 348
character set searches 349
character strings 346 to 347
  instruction used with 348 to 351
check bits (parity bits) 268
chip select (CSEL) 186
chopping, in floating point 317
chop/round floating point modes 312
CHR (cache hit register) 230 to 231
C instruction 68
CIS11 (Commercial Instruction Set) 345 to 402
CLC instruction 69
clearing of status registers 168 to 169
CLN instruction 69
clock status register (LKS) 208, 249 to 250
CLRB instruction 24, 68
CLRD floating point instruction 322 to 323
CLRF floating point instruction 322 to 323

CLR instruction 24, 68
CLV instruction 69
CLZ instruction 69
CMPB instruction 24, 69 to 70
CMPC(I) commercial instruction 349 to 350, 371 to 373
CMPD floating point instruction 323
CMPF floating point instruction 323
CMP instruction 69 to 70
CMPN(I) commercial instruction 362, 363, 373 to 374
CMPP(I) commercial instruction 362, 363, 373 to 374
CMR (cache maintenance register) 229 to 230
COMB instruction 71
COM instruction 24, 70
commands
  console 233 to 243
  console ODT command set 194 to 200
Commercial Instruction Set (CIS11) 345 to 402
commercial load descriptor instructions 364 to 365
communications, on UNIBUS 11 to 13
Computer Special Systems (CSS) 6
condition code instructions 43, 49 to 51
condition codes
  character string operations and 349 to 350
  decimal string instructions and 363
  in PDP-11/24 processors 189 to 190
  in PDP-11/44 processors 222
  in PDP-11/70 processors 266
Console Break character 233
console commands 233 to 243
console emulation 175 to 176
console emulator functions 175 to 176
Console Mode 177
console ODT 193 to 194
  address specification in 200 to 202
console ODT command set 194 to 200
consoles
  for PDP-11/04 and PDP-11/34A processors 174 to 177
  for PDP-11/24 processors 192 to 193
  for PDP-11/44 processors 232 to 233
  for PDP-11/70 processors 293 to 295
console state 192, 232, 233
CONTINUE console command 237 to 238
continue (CONT) switch 300
continuing, on PDP-11/70 processors 295
Control C (↑C) command 234
control characters 234 to 235
control chip (DC303) 185 to 186
controller registers 292 to 293
controllers, in PDP-11/70 processors 262, 271, 291 to 292
Control O (↑O) command 235
Control P (↑P) command 233, 235
Control Q (↑Q) command 235
control register 289
Control S (↑S) command 235
Control S ODT command 199 to 200
control switches 299
Control U (↑U) command 235
conversion routines 120 to 123
convert instructions 362
core memories 270

coroutines 110 to 114
corporate cabinets 272 to 273
CPU error register
  in PDP-11/24 processors 209 to 210
  in PDP-11/44 processors 251 to 253
  in PDP-11/70 processors 275
CPU registers
  in PDP-11/24 processors 212
  in PDP-11/44 processors 256 to 257
CPUs (central processing units)
  bus priority of 13, 18
  Floating Point Processors in 308 to 309
  PDP-11/04 and PDP-11/34A 171 to 180
  PDP-11/24 184 to 187
  PDP-11/44 217 to 258
  PDP-11/70 261 to 305
  processor traps and 116 to 117
  UNIBUS interrupts and 15
CSEL (chip select) 186
CSM instruction 70 to 71
CSS (Computer Special Systems) 6
CVTLN(I) commercial instruction 375 to 376
CVTLP(I) commercial instruction 375 to 376
CVTNL(I) commercial instruction 362, 363, 376 to 378
CVTNP(I) commercial instruction 378 to 379
CVTPL(I) commercial instruction 362, 363, 376 to 378
CVTPN(I) commercial instruction 378 to 379
data
  character, type of 346 to 348
  decimal, types of 351 to 361
  floating point 310 to 311
  memory management and 142 to 143
  in PDP-11/70 cache 280 to 282
data chip (DC302) 185, 186
data display lights 301
data formats, for floating point 309 to 311
data overlap 364
data select switch 301
data transactions, on UNIBUS 18 to 19
data transfers 292
DATI data transactions 18, 19
DATIP data transactions 18
DATOB data transactions 18
DATO data transactions 18
DC303 control chip 185 to 186
DC302 data chip 185, 186
DC304 memory management chip 186, 187
DECB instruction 71
decimal number system 8
decimal string data types 351 to 361
decimal string descriptors 352 to 353
decimal string instructions 361 to 364
DEC instruction 71
deferred addressing modes 25, 36 to 37
DEPOSIT console command 238 to 239
DEPOSIT console emulator function 175
deposit (DEP) switch 299
DF (Displacement Field) 150 to 151
diagnostics
  M9312 for 303 to 304
  M9301-YC, -YH for 304 to 305
  on PDP-11/04 and PDP-11/34A processors 171, 177
DIN (Displacement in the Block) 151

Direct Memory Access (DMA) 156, 291

Displacement in the Block (DIN) 151

Displacement Field (DF) 150 to 151

DIVD floating point instruction 324 to 325

DIVF floating point instruction 324 to 325

DIV instruction 71 to 72

division 120 to 122

DIVP(I) commercial instruction 363, 379 to 381

DMA (Direct Memory Access) 156, 291

documentation 8

dollar sign ($) ODT command 197 to 198

DONE/READY flag 104

double-operand instructions 43 to 46, 51 to 52

ECC, *see* error correcting code

ED (expansion direction) 147

EIS (Extended Integer Instructions) 56, 172

EMT instruction 72 to 73, 117, 118

ENABLE/HALT switch 294, 295, 300

environment
- for PDP-11/04 and PDP-11/34A processors 180
- for PDP-11/24 processors 214
- for PDP-11/44 processors 258
- for PDP-11/70 processors 274 to 275

/E qualifier 237

error correcting code (ECC)
- parity and 283
- in PDP-11/44 processors 223 to 224
- in PDP-11/70 processors 267 to 269

error flags, in Memory Management Register #0 162, 164

errors
- in Floating Point Processors 315, 317 to 318
- parity 284 to 285
- in PDP-11/44 processors 254
- in PDP-11/70 processors 277 to 279
- processor traps and 116 to 117
- recovery from 305
- status indicator lights for 301 to 302
- time-out 211

EXAMINE console command 239 to 240

EXAMINE console emulator function 175

examine (EXAM) switch 299

exceptions, in Floating Point Processor 312 to 316

executive programs 135, 141

exits
- from console state 233
- from main programs 48

expansion direction (ED) 147

Extended Integer Instructions (EIS) 56, 172

Fault Recovery (Status) registers 161 to 169

faults, multiple 169

FEA (floating exception address register) 307, 315 to 316

FEC (floating exception code register) 315 to 316, 318

FILL console command 240

FIS (Floating Instruction Set) 307

flags, in Memory Management Register #0 162, 164

floating exception address register (FEA) 307, 315 to 316

floating exception code 315 to 316
floating exception code register (FEC) 315 to 316, 318
Floating Instruction Set (FIS) 307
Floating Point Processor (FPP; PDP-11 floating point) 307 to 342
  on PDP-11/34A processors 172, 174
  on PDP-11/44 processors 231
  on PDP-11/70 processors 263, 270 to 271
floating point status register (FPS) 311 to 315
formats
  for addressing mode instructions 25
  for branch instructions 46 to 47
  for condition code operators 50 to 51
  for double-operand instructions 45
  for floating point 309 to 311
  for floating point instructions 319 to 320
  for jump and subroutine instructions 47 to 48
  for single-operand instructons 44
FP11 instruction set 307, 318 to 342
  accuracy in 317
FPP, *see* Floating Point Processor
FPS (floating point status register) 311 to 315
front panels 171

general purpose registers 23
  commercial instructions and 350
  commercial load descriptor instructions and 364 to 365
  Memory Management Register #1 and 166
  in PDP-11/44 processors 220 to 221
  in PDP-11/70 processors 263 to 265, 297 to 298
  Program Counter as 32
  reentrant code and 109
  stacks and 99 to 100
  see also registers
G ODT command 198 to 199
GO ODT command 198 to 199
/G qualifier 235 to 236

HALT console command 240
HALT instruction 73
HALT ODT command 200
hardware, Floating Point Processor as 307
hardware interrupt requests 271
H960 cabinets 272
hidden bit 309
high error address register 286 to 287
high-speed I/O controllers 262, 271, 291 to 292
high-speed mass storage 271
hit/miss register 291
H ODT command 200

immediate addressing mode 25, 32 to 33, 37, 40
  position-independent coding and 94
INCB instruction 24, 73
INC instruction 24, 73
index addressing modes 25, 30 to 31, 36, 39
  position-independent coding and 94
index deferred addressing mode 25, 31, 37, 39
index registers 23
indicators, on PDP-11/70 processors 299 to 302
indirect addressing modes 25, 36 to 37

INITIALIZE console command 240
"in-line" form (Commercial Instruction Set) 345, 350, 363 to 364
instruction cycles 13 to 14
instructions and instruction sets 43 to 91
  for addressing modes 24
  back-up/restart recovery for 168
  Commercial Instruction Set 345 to 402
  for Floating Point Processor 174, 307, 308, 316 to 342
  memory management and 142 to 143
  reserved 211, 254, 277
  trap 117 to 119
instruction suspension 365 to 367
interfaces
  for high-speed I/O controllers 291 to 292
  UNIBUS map as 160 to 161
internal register designators (ODT commands) 197 to 198
interrupt exits 48
interrupts 104 to 107
  in Floating Point Processor 310, 315
  instructions for 48
  memory management and 141 to 142
  in PDP-11/70 processors 271 to 272
  on UNIBUS 15 to 16
interrupt service routines (ISRs) 15, 104
interrupt servicing 16
invalid characters, in console ODT 202
I/O addresses 200
I/O bus 291
I/O controllers, in PDP-11/70 processors 262, 271, 291 to 292
I/O device registers
  in PDP-11/24 processors 212
  in PDP-11/44 processors 256 to 257
I/O page registers 226 to 231
IOT instruction 73 to 74
ISRs (interrupt service routines) 15, 104

JMP instruction 74 to 75
JSR instruction 75 to 77, 99
  in coroutine calls 110
  format for 47
  subroutine linkage and 103, 104
jump and subroutine instructions 43, 47 to 48

KEF11-AA (FP11 instruction set) 307, 317
kernel mode 141, 142
  in PDP-11/04 and PDP-11/34A processors 172
  in PDP-11/24 processors 185, 189, 201
  in PDP-11/44 processors 218, 221, 222
  in PDP-11/70 processors 263, 266
kernel program 140
Kernel Stack 264
KY11-LA (oerator's console) 174 to 176
KY11-LB (programmer's console) 176 to 177

lamp test switch 300
LDCDF floating point instruction 325 to 326
LDCFD floating point instruction 325 to 326
LDCID floating point instruction 326 to 327
LDCIF floating point instruction 326 to 327

LDCLD floating point instruction 326 to 327

LDCLF floating point instruction 326 to 327

LDD floating point instruction 328 to 329

LDEXP floating point instruction 327 to 328

LDF floating point instruction 328 to 329

LDFPS floating point instruction 329

L2Dr commercial load descriptor instruction 354 to 365, 383 to 384

L3Dr commercial load descriptor instruction 364 to 365, 384 to 385

leading overpunch numeric string data 351

leading separate numeric string data 351

line clock status register (LKS) 208, 249 to 250

line feed (<LF>) ODT command 197

linkage 99
  of coroutines 111
  of subroutines 103 to 104

LKS (clock status register) 208, 249 to 250

LOAD ADDR console emulator function 175

loaders, bootstrap 171, 177, 303 to 304

LOCC(I) commercial instruction 349, 350, 381 to 383

long integer data 360 to 361

looping 132 to 133

low error address register 286

lower size register 276

MACRO-11 assembler language
  recursion in 116
  register operands and 25
  relocatable object modules as output of 93

maintainability, *see* reliability, availability, and maintainability program

Maintenance Mode 177

maintenance register 290 to 291

mapped memory references 295 to 296

mapping 153 to 161
  memory referencing and 295 to 296

MARK instruction 77

masks 347

master/slave communications 11, 13

master sync (MSYNC) signal 19

MATC(I) commercial instruction 349, 350, 385 to 387

M9312 bootstrap loader 177, 303 to 304

memories
  addresses in 11
  in PDP-11/04 and PDP-11/34A processors 172 to 173
  in PDP-11/24 processors 190 to 191
  in PDP-11/44 systems 223 to 224
  in PDP-11/70 systems 267 to 270
  referencing of 295 to 296

memory box options 269 to 270

memory management 135 to 169
  in PDP-11/04 and PDP-11/34A processors 172
  in PDP-11/24 processors 190 to 191
  in PDP-11/44 processors 281, 244
  in PDP-11/70 processors 261, 267

memory management chip (DC304) 186, 187

Memory Management Register #0 (MMR0) 162 to 166

Memory Management Register #1 (MMR1) 166 to 167

Memory Mangement Register #2 (MMR2) 167

Memory Management Register #3 (MMR3) 167 to 168

memory management unit (MMU)
parity and 284
physical address construction and 150
processor I/O addressing of registers in 200

memory system error register 287 to 288

MFPD instruction 77 to 78

MFPS instruction 78

MFPT instruction 78 to 79

microprogram break register 276

MICROSTEP console command 240 to 241

miscellaneous instructions 43, 48 to 49

MK11-B memory boxes 269

MK11-CE memory array modules 269

MK11-CF memory array modules 269

MK11-C memory arrays 269

MMR0 (Memory Management Register #0) 162 to 166

MMR1 (Memory Management Register #1) 166 to 167

MMR2 (Memory Management Register #2) 167

MMR3 (Memory Management Register #3) 167 to 168

MMU, *see* memory management unit

MODD floating point instruction 293 to 332

modes
for addressing, in floating point 316
of operation for Floating Point Processor 311 to 312
in PDP-11/04 and PDP-11/34A processors 172
in PDP-11/24 processors 185, 189, 201
in PDP-11/44 processors 221 to 222
in PDP-11/70 processors 263, 265 to 266
see also addressing modes

MODF floating point instruction 329 to 332

MOS memories 190, 223 to 224, 268 to 270

mounting boxes
for PDP-11/24 processors 213 to 214
for PDP-11/44 processors 258

MOVB instruction 79

MOVC(I) commercial instruction 349, 387 to 389

MOV instruction 79

MOVRC(I) commercial instruction 349, 389 to 391

MOVTC(I) commercial instruction 349, 351, 391 to 393

/M qualifier 236

MSYNC (master sync) signal 19

MTPD instruction 79 to 80

MTPS instruction 80 to 81

MULD floating point instruction 332 to 333

MULF floating point instruction 332 to 333

MUL instruction 81

MULP(I) commercial instruction 393 to 395

multiple-user environments, reentrant programs in 108

multiple faults 169
multiplication 122 to 123
multiprogramming 135, 141
  in PDP-11/70 systems 265
M9301-YC boostrap loader 303 to 304

N bit 49
NEGB instruction 81
NEGD floating point instruction 334
NEGF floating point instruction 334
NEG instruction 81
nesting
  of bus interrupts 16
  of interrupts 105 to 107
  recursion for 115 to 116
NEXT console command 241
nibbles 351
nonexistent memory errors 254, 277
nonprocessor requests (NPRs) 13, 14
  in PDP-11/70 systems 271
  priority level of 17
  used bY UNIBUS map 155
nonvanishing floating point numbers 309 to 310
NOP instruction 81
normal/maintenance floating point modes 312
notation, numerical 8
NPRs, *see* nonprocessor requests
/N qualifier 236
numbers, in floating point 309 to 310
numerical notation 8
numeric string data 351
  instructions for 362

octal number system 8
  entering addresses in 201
odd addressing errors 117, 254, 277
ODT 193 to 194
  address specification in 200 to 202
  command set for 194 to 200
ODT time-out 202
OEMs (original equipment manufacturers) 6
operand delivery 363 to 364
operator's console (KY11-LA) 174 to 176
original equipment manufacturers (OEMs) 6
overflows, in floating point 317, 318
overpunch string data 356 to 358

packaged systems 6 to 7
packaging
  of PDP-11/24 processors 212
  of PDP-11/44 processors 257
  for PDP-11/70 processors 272 to 273
packed string data 351, 353 to 355
  instructions for 362
page address field (PAF) 151, 152
Page Address Register (PAR) 143, 151
Page Descriptor Register (PDR) 143 to 147
pages 138, 140
PAR (Page Address Register) 143, 151
parity 267, 283 to 285
  in high-speed I/O controllers 292
parity bits 268
  in console ODT 195
patching, trap handlers for 118
PC *see* Program Counter
PC absolute mode 25, 33, 37
PC immediate mode 25, 32 to 33, 37

PC relative deferred mode 26, 34 to 35, 37

PC relative mode 26, 33 to 34, 37

PDP-11 family 1 to 2
  UNIBUS and architecture of 11 to 13

PDP-11 floating point, *see* Floating Point Processor

PDP-11/04 processors 171 to 180
  mapping on 153, 158

PDP-11/24 processors 183 to 214
  mapping on 154 to 157, 159
  Memory Management Register #1 in 167

PDP-11/34A processors 171 to 180
  mapping on 154, 158, 159

PDP-11/44 processors 217 to 258
  Commercial Instruction Set on 345
  mapping on 154 to 157, 159

PDP-11/70 processors 261 to 305
  mapping on 154 to 157, 159

PDR (Page Descriptor Register) 143 to 147

peripheral device register addressing 136

peripherals 6
  bus priority of 13

physical address space 136
  16-bit, mapping in 158
  18-bit, mapping in 159
  22-bit, mapping in 159
  construction of 150 to 152

PIC (position-independent coding) 93 to 97

PIR (program interrupt request) register 256, 279

P ODT command 199

pointers 23

POPping stacks 98

position-dependent code 95 to 96

position-independent coding (PIC) 93 to 97

power, starting and stopping 294 to 295

power failures
  cache memory and 282
  in PDP-11/24 processors 210
  in PDP-11/44 processors 254
  in PDP-11/70 processors 269, 277
  processor traps for 116 to 117

power specifications
  for PDP-11/24 processors 213
  for PDP-11/44 processors 258
  for PDP-11/70 processors 274

power switch 299

Priority Arbitration logic 291

priority levels
  in PDP-11/24 processors 189
  in PDP-11/44 processors 222
  in PDP-11/70 processors 266, 271
  for traps 211, 255, 277 to 278
  on UNIBUS 11 to 13, 16 to 18

proceed ODT command 199

processor control registers 275 to 276

processor I/O addresses 200

processors, *see* CPUs

Processor Stack Pointers, *see* Stack Pointers

Processor Status Word (PS; PSW)
  bus interrupts and 15, 16
  condition code bits on 49 to 51
  interrupts and 104, 105
  memory management and 142
  ODT command for 198
  in PDP-11/24 processors 188 to 189, 210
  in PDP-11/24 processors 221, 222, 253 to 254
  in PDP-11/70 processors 265 to 266, 276
  reentrant code and 109
  trap instruction and 117

processor traps 116 to 117

in PDP-11/24 processors 210
in PDP-11/44 processors 254
in PDP-11/70 processors 276 to 278

Program Counter (PC) 23, 32, 99
bus interrupts and 15, 16
in coroutine calls 110
interrupts and 105
memory management and 142
in PDP-11/24 processors 188
in PDP-11/44 processors 220
in PDP-11/70 processors 264
trap instructions and 117, 118

program counter addressing modes 25 to 26, 32 to 35, 37

program interrupt request register (PIR) 256, 279

program interrupt requests 256, 271 to 272, 279 to 280

program I/O state
in PDP-11/24 processors 192 to 193
in PDP-11/44 processors 233

programmer's console (KY11-LB) 176 to 177

programming 93 to 133
on PDP-11 family systems 2 to 6

programs
address relocation of 136 to 139
exits from 48
multiprogramming and 135

protection of memory, memory management for 140

PS, *see* Processor Status Word

PSW, *see* Processor Status Word

pure code 108

PUSHing stacks 98

qualifiers 235 to 237

RAMP *see* reliability, availability, and maintainability program

RBUF, *see* receiver data buffer register

receiver control/status register (RCSR) 205
TERM RCSR 202, 244
TU58 RCSR 247

receiver data buffer register (RBUF) 193, 206
TERM RBUF 203 to 204, 244 to 245
TU58 RBUF 247 to 248

RECEIVER DONE flag 209

receivers, timings for 209, 251

recovery from errors 305

recursion 114 to 116

Red Zone Violations 278 to 279

reentrancy 108 to 110

register addressing mode 25, 26, 35, 38

register deferred addressing mode 25, 27, 36, 38

"register" form (Commercial Instruction Set) 345, 350, 363

registers
Active Page Registers 138, 142 to 148
address specification for 200 to 201
cache 285 to 291
Fault Recovery 161 to 169
in Floating Point Processors 307, 311 to 316
general purpose 23
ODT designators for 197 to 198
in PDP-11/24 processors 187 to 190, 209 to 212
in PDP-11/44 processors 220 to 221, 226 to 231, 251 to 257
in PDP-11/70 processors 263 to 267, 275 to 276, 279, 292 to 293, 297 to 298
peripheral device register addressing of 136
second serial line unit 205 to 208
terminal serial line 202 to 205, 244 to 246
TU58 serial line unit 247 to 250

see also general purpose registers
relative addressing modes 26, 33 to 34, 37, 40
position-independent coding and 94
relative deferred addressing mode 26, 34 to 35, 37, 40
reliability, availability, and maintainability program (RAMP)
of PDP-11/24 processors 184
of PDP-11/70 processors 262 to 263, 283
relocation, address 136 to 139
in UNIBUS map 160 to 161
REPEAT console command 241
reserved instructions 117, 211, 254, 277
RESET instruction 82
R ODT command 197 to 198
ROLB instruction 82
ROL instruction 82
RORB instruction 83
ROR instruction 83
rounding bit 317
RTI instruction 83
for nesting 105
RTS instruction 84
format for 48, 103
for nesting 105
RTT instruction 84 to 85

SACK 14
SBCB instruction 85
SBC instruction 85
SCANC(I) commercial instruction 349, 350, 395 to 397
SCC instruction 86
searches, commercial instructions for 349
SEC instruction 86
second serial line unit registers 205 to 208
SEN instruction 86
separate string data 358 to 360
serial line unit registers 247 to 250
serial line unit timing 251
SETD floating point instruction 335
SETF floating point instruction 334
SETI floating point instruction 335
SETL floating point instruction 335
SEV instruction 86
SEZ instruction 86
short/long floating point modes 311
signed packed string data 354
signed zoned numeric string data 351, 355 to 356
sign-magnitude string data 351
single/double floating point modes 311
single instruction/single bus cycle 298 to 300
S instruction 85
single-operand instructions 43 to 44, 51
SKPC(I) commercial instruction 349, 397 to 399
slash (/) ODT command 195 to 196
slave sync (SSYN) signal 15, 19
SOB instruction 86, 87
S ODT command 198
software
for floating point 307
for memory management 140
software exits 48
Software Services 6
software traps 141
SP, *see* Stack Pointers
SPACE-BAR-STEP FEATURE console command 241

SPANC(I) commercial instruction 349, 399 to 401

special characters 234 to 235

specialized systems 6

special symbols 53 to 54

specifications
- for PDP-11/04 and PDP-11/34A processors 180
- for PDP-11/24 processors 212 to 214
- for PDP-11/44 processors 257 to 258
- for PDP-11/70 processors 272 to 275

SPL instruction 87

SSYN (slave sync) signal 15, 19

stack addressing 23

stack limit register (SL) 266 to 267, 278, 279

Stack Pointers (SP) 23, 97, 98
- address specification of 201
- in PDP-11/24 processors 188
- in PDP-11/44 processors 221
- in PDP-11/70 processors 264
- subroutines and 100

stacks 97 to 103
- Commercial Instruction Set and 367
- limit violations of 211 to 212, 255
- overflow boundary for 190, 266 to 267, 278 to 279
- in PDP-11/44 processors 221, 223
- in PDP-11/70 processors 263, 264
- reentrant code and 109

START console command 241 to 242

START console emulator function 176

starting, on PDP-11/70 processors 294

START switch 300

status indicator lights 301 to 302

Status (Fault Recovery) Registers 161 to 169

STCDF floating point instruction 336

STCDI floating point instruction 337 to 338

STCDL floating pont instruction 337 to 338

STCFD floating point instruction 336

STCFI floating point instruction 337 to 338

STCFL floating point instruction 337 to 338

STD instruction 337

step operations 296 to 297

STEXP floating point instruction 339

STF instruction 337

STFPS floating point instruction 339

stopping, on PDP-11/70 processors 295

STST floating point instruction 316, 340

SUBD floating point instruction 340 to 341

SUBF floating point instruction 340 to 341

SUB instruction 87 to 88

SUBN(I) commercial instruction 362, 401 to 402

SUBP(I) commercial instruction 362, 401 to 402

subroutine instructions 43, 47 to 48

subroutines
- coroutines and 112, 113
- linkage for 103 to 104
- recursion in 114 to 115
- stacks and 99 to 100

supervisor mode 141
- in PDP-11/44 processors 221, 222

in PDP-11/70 processors 263
Supervisor Stack 264
suspension of instructions 365 to 367
swapping, of coroutines 110
switches, on PDP-11/70 processors 299 to 302
switch register 300
SXT instruction 88 to 89
symbols 53 to 54
system I/D register 276
system terminals (consoles) 192, 232

tags 282
/TB qualifier 237
terminal serial line unit registers
in PDP-11/24 processors 202 to 205
in PDP-11/44 processors 244 to 246
TERM RBUF (receiver data buffer) register 203 to 204, 244 to 245
TERM RCSR (receiver control/status) register 202, 205, 244
TERM XBUF (transmitter data buffer) register 204 to 205, 246
TERM XCSR (transmitter control/status) register 204, 245 to 246
TEST console command 242
time-out errors 117, 211, 254, 277
timing
in PDP-11/24 processors 209
in PDP-11/44 processors 251
trace traps 190, 222 to 223
trailing overpunch numeric string data 351
trailing separate numeric string data 351
transmitter control/status register (XCSR) 207
TERM XCSR 204, 245 to 246
TU58 XCSR 248 to 249
transmitter data buffer register (XBUF) 207 to 208
TERM XBUF 204 to 205, 246
TU58 XBUF 249
TRANSMITTER READY flag 209
transmitters, timings for 209, 251
transmitter status register *see* transmitter control/status register

trap error 117
trap exits 48
trap handlers 118 to 119
trap handling
in PDP-11/24 processors 211
in PDP-11/44 processors 255
in PDP-11/70 processors 277
TRAP instruction 89, 117, 118
trap instructions 43, 48, 117 to 119
traps
clearing status registers after 168 to 169
Fault Recovery Registers and 161 to 162
Memory Management Register #0 and 165
parity and 284 to 285
in PDP-11/70 processors 266, 276 to 278
priorities for 211
processor 116 to 117, 210, 254
relocation of address of 142
software 141
trace 190, 222 to 223
TSTB instruction 89
TSTD floating point instruction 342
TSTF floating point instruction 342
TST instruction 89
TU58 receiver control/status register (TU58 RCSR) 247
TU58 receiver data buffer register (TU58 RBUF) 247 to 248

TU58 serial line unit registers 247 to 250
TU58 transmitter control/status register (TU58 XCSR) 248 to 249
TU58 transmitter data buffer register (TU58 XBUF) 249

UART (Universal Asynchronous Receiver/Transmitter) 209, 251
undefined variable 310
underflows, in floating point 317 to 318
UNIBUS 1, 11 to 20
  cache bypass of 173
  high-speed I/O controllers and 291 to 292
  mapping of addresses for 153, 157 to 158
  in PDP-11/24 systems 184
  in PDP-11/44 systems 218
  in PDP-11/70 systems 262
  time-out errors in 117
UNIBUS map 153 to 155, 157 to 158, 160 to 161
  in PDP-11/24 processors 184, 191
  in PDP-11/44 processors 218, 224
  in PDP-11/70 processors 261, 267 to 268
Universal Asynchronous Receiver/Transmitter (UART) 209, 251
unmapped memory references 295
UNPREDICTABLE conditions 345 to 346
unsigned packed string data 354 to 355
unsigned zoned numeric string data 351, 356
upper size register 276
user mode 141
  in PDP-11/04 and PDP-11/34A processors 172
  in PDP-11/24 processors 185, 189, 201
  in PDP-11/44 processors 221, 222
  in PDP-11/70 processors 263
User Stack 264

VBA (virtual bus address) 150 to 152
V bit 50
vector addresses 105
  assignments of 208, 250
  memory management control of 142
  trap errors and 117, 119
violations, of stack limits 211 to 212, 255, 278 to 279
virtual address space 135
  physical address construction and 150
virtual bus address (VBA) 150 to 152
volatile information
  in MOS memories 190
  in power failures 116

WAIT instruction 90 to 91

XBUF *see* transmitter data buffer register
XCSR *see* transmitter control/status register
XOR instruction 91

Yellow Zone Violations 278, 279
YH/M9312 bootstrap loader 303 to 304

Z bit 49
zero, in floating point 310
zoned string data 355 to 356

1-800-325-3700

NOTES

PDP-11 PROCESSOR HANDBOOK 1981 **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 handbook? (format, accuracy, completeness, organization, etc.) _____

_____

_____

_____

What features are most useful? _____

_____

_____

_____

Does the publication satisfy your needs? _____

_____

_____

What errors have you found? _____

_____

_____

_____

Additional comments _____

_____

_____

Name _____

Title _____

Company _____ Dept. _____

Address _____

City _____ State _____ Zip _____

(staple here)

(staple here)

(please fold here)

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD, MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
NEW PRODUCTS MARKETING
PK3-1/M92
MAYNARD, MASS. 01754

digital

HANDBOOK SERIES

Microcomputers and Memories
Microcomputer Interfaces
PDP-11 Processor
PDP-11 Software
Peripherals
Terminals and Communications
VAX Architecture
VAX Software
VAX Hardware

digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, MA 01754, Tel. (617) 897-5111 — SALES AND SERVICE OFFICES; UNITED STATES — ALABAMA, Birmingham, Huntsville ARIZONA, Phoenix, Tucson ARKANSAS, Little Rock CALIFORNIA, Costa Mesa, El Segundo, Los Angeles, Oakland, Sacramento, San Diego, San Francisco, Monrovia, Santa Barbara, Santa Clara, Sherman Oaks COLORADO, Colorado Springs, Denver CONNECTICUT, Fairfield, Meriden DELAWARE, Newark FLORIDA, Miami, Orlando, Pensacola, Tampa GEORGIA, Atlanta HAWAII, Honolulu IDAHO, Boise ILLINOIS, Chicago, Peoria INDIANA, Indianapolis IOWA, Bettendorf KENTUCKY, Louisville LOUISIANA, New Orleans MARYLAND, Baltimore MASSACHUSETTS, Boston, Springfield, Waltham MICHIGAN, Detroit, Kalamazoo MINNESOTA, Minneapolis MISSOURI, Kansas City, St. Louis NEBRASKA, Omaha NEW HAMPSHIRE, Manchester NEW JERSEY, Cherry Hill, Parsippany, Princeton, Somerset NEW MEXICO, Albuquerque, Los Alamos NEW YORK, Albany, Buffalo, Long Island, New York City, Rochester, Syracuse, Westchester NORTH CAROLINA, Chapel Hill, Charlotte OHIO, Cincinnati, Cleveland, Columbus, Dayton OKLAHOMA, Tulsa OREGON, Portland PENNSYLVANIA, Harrisburg, Philadelphia, Pittsburgh RHODE ISLAND, Providence SOUTH CAROLINA, Columbia, Greenville TENNESSEE, Knoxville, Nashville TEXAS, Austin, Dallas, El Paso, Houston, San Antonio UTAH, Salt Lake City VERMONT, Burlington VIRGINIA, Fairfax, Richmond WASHINGTON, Seattle, Spokane WASHINGTON D.C. WEST VIRGINIA, Charleston WISCONSIN, Milwaukee INTERNATIONAL — EUROPEAN AREA HEADQUARTERS: Geneva, Tel: [41] (22)-93-33-11 INTERNATIONAL AREA HEADQUARTERS: Acton, MA 01754, U.S.A., Tel: (617) 263-6000 AUSTRALIA, Adelaide, Brisbane, Canberra, Hobart, Melbourne, Perth, Sydney, Townsville AUSTRIA, Vienna BELGIUM, Brussels BRAZIL, Rio de Janeiro, Sao Paulo CANADA, Calgary, Edmonton, Halifax, Kingston, London, Montreal, Ottawa, Quebec City, Regina, Toronto, Vancouver, Victoria, Winnipeg DENMARK, Copenhagen ENGLAND, Basingstoke, Birmingham, Bristol, Ealing, Epsom, Leeds, Leicester, London, Manchester, Reading, Welwyn FINLAND, Helsinki FRANCE, Bordeaux, Lyon, Paris, Puteaux, Strasbourg HOLLAND, Amstelveen, Delft, Utrecht HONG KONG IRELAND, Dublin ISRAEL, Tel Aviv ITALY, Milan, Rome, Turin JAPAN, Osaka, Tokyo MEXICO, Mexico City, Monterrey NEW ZEALAND, Auckland, Christchurch, Wellington NORTHERN IRELAND, Belfast NORWAY, Oslo, PUERTO RICO, San Juan SCOTLAND, Edinburgh REPUBLIC OF SINGAPORE SPAIN, Barcelona, Madrid SWEDEN, Gothenburg, Stockholm SWITZERLAND, Geneva, Zurich, WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nurnberg, Stuttgart