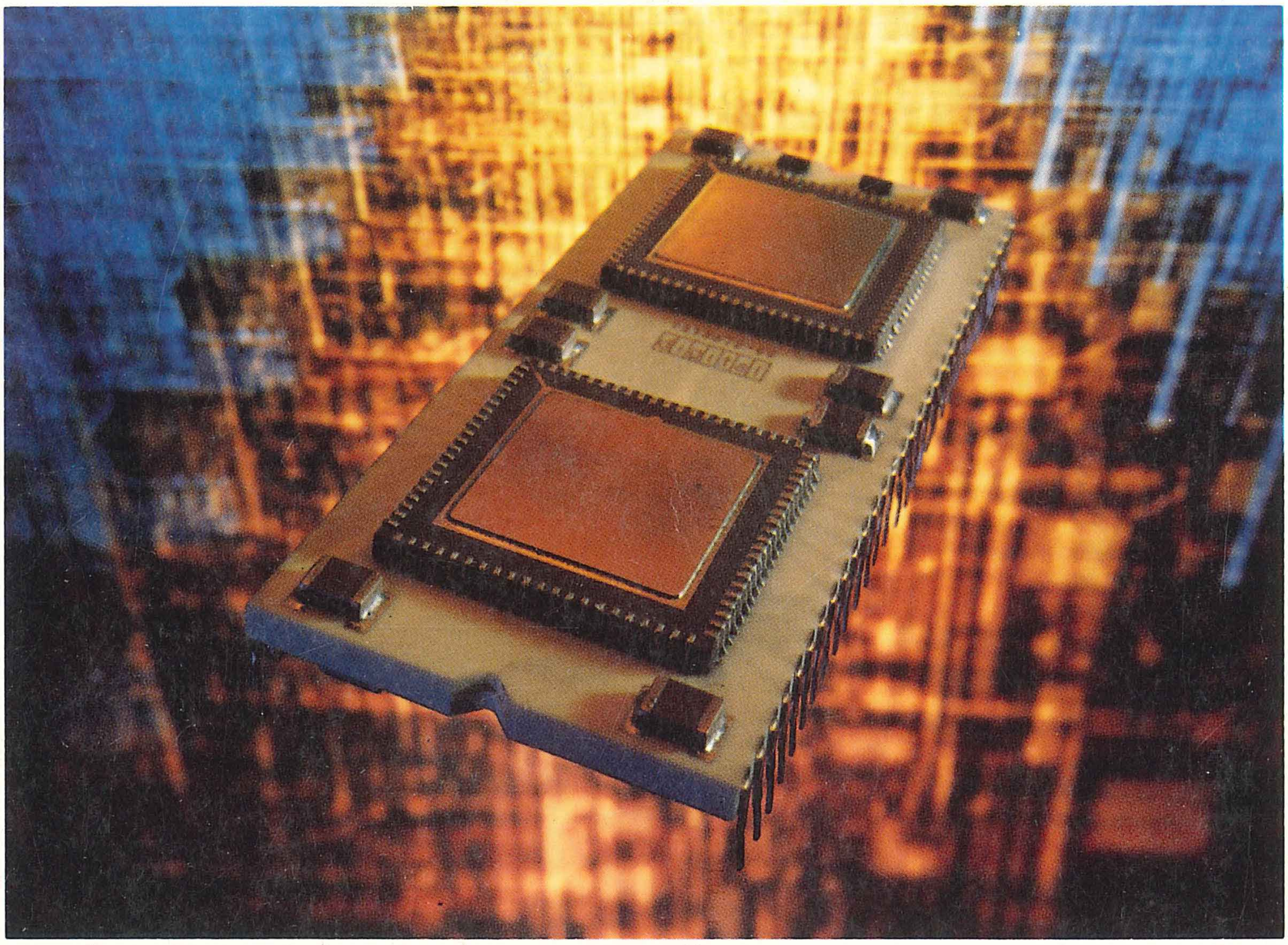
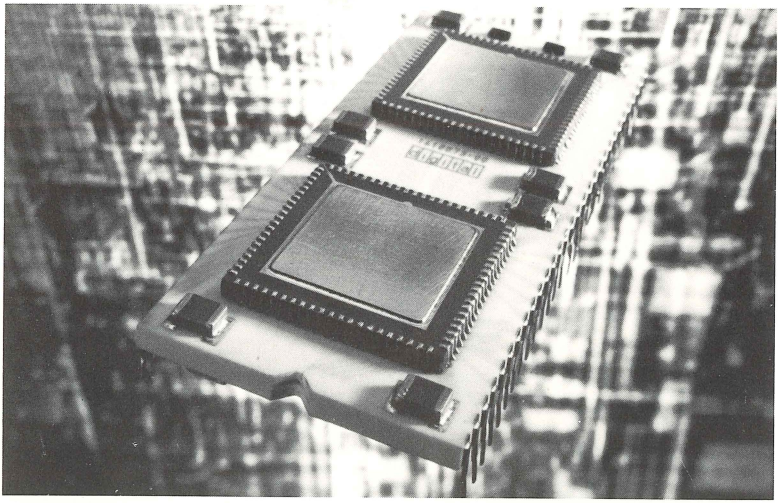
232



PDP-11

Architecture Handbook

digital



The J-11 chipset.

The PDP-11/70 system.

The tradition continues with the J-11, DIGITAL's newest high-performance microprocessor. It offers the architecture, power, and functions of the PDP-11/70 (the PDP-11 family performance leader) in a single 60-pin package. The J-11 will form the basis of a new line of DIGITAL products. These powerful systems will carry the PDP-11 architecture years into the future.

PDP-11

Architecture Handbook

digital

Digital Equipment Corporation makes no representation that the interconnection of its products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting of license to make, use, or sell equipment constructed in accordance with this description.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this manual.

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

TABLE OF CONTENTS

CHAPTER 1 ARCHITECTURE AND THE PDP-11 FAMILY

CHAPTER 2 KEY ELEMENTS OF PDP-11 ARCHITECTURE

CHAPTER 3 PDP-11 DATA REPRESENTATION

CHAPTER 4 ADDRESSING MODES

CHAPTER 5 INSTRUCTION SET

CHAPTER 6 PDP-11 FLOATING-POINT

CHAPTER 7 COMMERCIAL INSTRUCTION SET

CHAPTER 8 TRAPS AND INTERRUPS

CHAPTER 9 MAPPING TO MEMORY AND BUSSES

PREFACE

DIGITAL's PDP-11 computer family was launched in 1970. Since that time, they have enjoyed unparalleled success—DIGITAL has sold more minicomputers and more 16-bit microcomputers than any other company. DIGITAL's leadership of the 16-bit marketplace is due to a number of factors:

- product range
- cost effectiveness
- compatibility

DIGITAL offers you a full range of products to meet your needs: from chipsets to boards to systems. With this broad product range, we can meet the needs of your application.

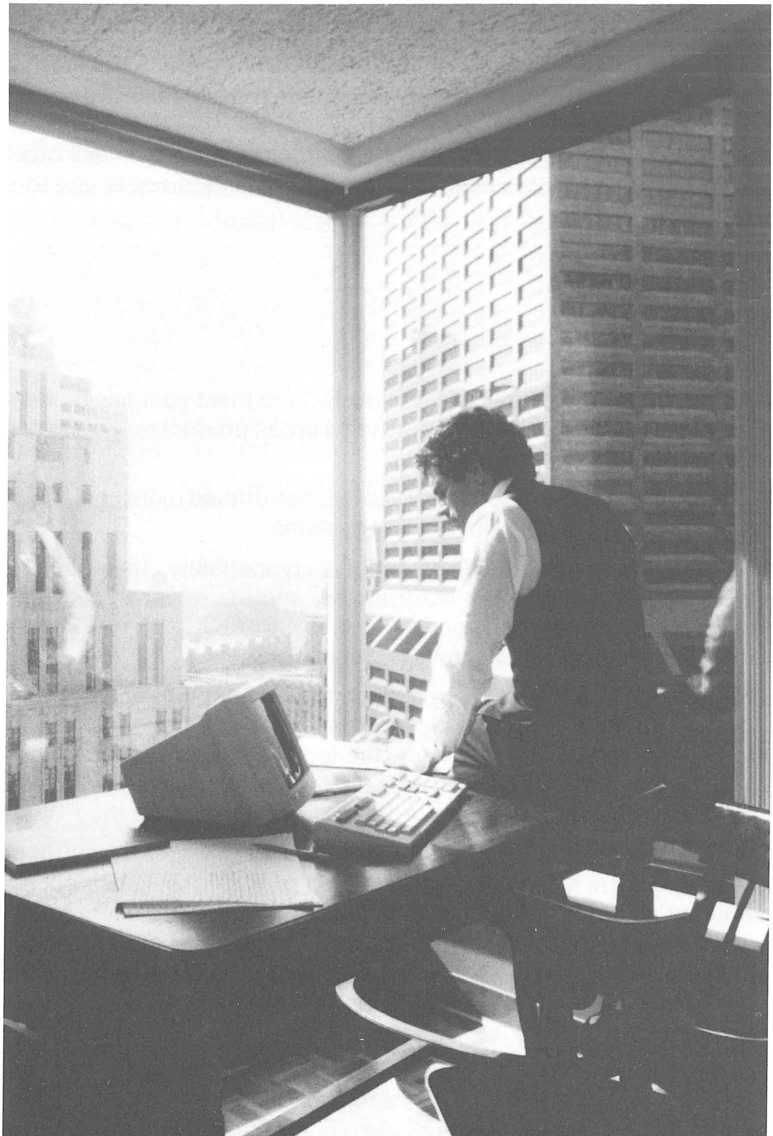
As technologies have improved, DIGITAL has offered more computing power in smaller and less expensive systems.

The key concept of the PDP-11 family is compatibility. The software, the operating systems, the I/O systems, and the peripherals are all largely compatible. The broadly-based compatibility of the PDP-11 family is a function of its common architecture.

The purpose of this book it to explain the architecture of DIGITAL's PDP-11 computers. This book is aimed at two distinct groups of readers: first, those who need to know the technical details of PDP-11 architecture; second, those who may be new to computers or inexperienced with PDP-11s and need a tutorial introduction to the PDP-11 family and its architecture.

Those readers in the more technical group may want to read about the evolution of the PDP-11 and LSI-11 computers, and the PDP-11 Milestones section of Chapter 1. Then, they should refer directly to the chapter(s) where they have specific questions. Those readers who frequently refer to Table 5-1—Instruction Set— will appreciate its black page tabs, which make it easy to find.

Readers who require tutorial information should read all of the first two chapters, using Chapter 2 as a reference guide. From the brief introductions in Chapter 2, the reader should be able to select those chapters and appendices of particular interest.



CHAPTER 1
ARCHITECTURE AND THE PDP-11 FAMILY

INTRODUCTION

What is computer architecture?

Before we define computer architecture, let us review some basic computer terms. A program is a series of instructions that tell a computer how to operate on data. Most programmers today work in a high-level language such as BASIC, COBOL, or FORTRAN. Programs coded in these languages must be translated into instructions that the computer can understand. One high-level language statement is translated (or compiled) into many machine instructions. To have more control over the computer, and to gain a better appreciation of its operation, a programmer must use a language that is closer to the machine instructions used in the computer. Such a language is called an assembly language.

Each assembly language statement corresponds to one machine instruction (an elementary computer operation). The entire group of instructions that can be used in assembly language statements is called the **instruction set**.

Programs access individual data items, manipulate them, and move them into different areas of the computer. To provide fast, temporary storage and facilitate operations on these data, the assembly language programmer can use a number of locations called **registers**.

To find data or program instructions, the computer must remove information from its main storage area, called main memory. Each memory location has an address—a number used to find that information location. The methods used to arrive at this address are called **addressing techniques**.

A computer can find out about external events by its **interrupt structure**. This mechanism is also used for making an orderly transition between programs.

We have now introduced the main elements of computer architecture:

- Instruction sets
- Addressing techniques
- Registers
- Interrupt structures

These are the tools that the assembly language programmer manipulates to write programs. *We can define computer architecture as the behavior of a computer seen by an assembly language program.*

The Importance of a Consistent Architecture

Why is computer architecture so important? When we hear about advances in the computer industry we usually hear about new technologies for building faster, smaller and less expensive computers. Technologies seem to overshadow computer architecture. But the most advanced computer technologies won't help a programmer if the computer's architecture—the programmer's tool kit—is awkward.

Let's make an analogy to music to illustrate the importance of a consistent architecture. There is a standard architecture for the modern piano. If you know how to play the piano, you can play any piano, because the architecture—the keyboard—is standard. Middle C is always in the same position; the number of keys, their placement, and their pitch are all specified and standard.

There is great latitude in the construction of pianos using this architecture, however. Different pianos may produce the same pitch when a key is struck, but the tone is a function of the construction. A spinet does not produce the same tone quality as a concert grand piano. This reflects the woods used in constructing the piano, as well as the size and shape of the sound board, and other factors. Also, the touch and action of pianos will vary with the quality of materials and construction.

What if a manufacturer decided to build an inexpensive electronic piano? To save production costs, he wants to use buttons instead of the traditional piano keys. Who will buy his new pianos? By changing the piano's architecture, he risks losing the market for the millions of people trained to play the piano.

Just as the manufacturers of pianos are committed to their architecture, so too is DIGITAL committed to its PDP-11 computer architecture.

Thousands of programmers trained on PDP-11s want to see its proven architecture maintained as computer technologies advance. Their programs will run, producing the same results on PDP-11 computers of different generations, whether built with transistors, or very large-scale integrated circuits, because the architecture has been maintained. When an architecture is maintained, existing programs will not require recoding; they will run on new machines that are smaller, faster, and cheaper. From a corporate viewpoint, this means major savings—a company can protect its investment in software while taking advantage of the latest computer technologies.

THE PDP-11 FAMILY CONCEPT

The PDP-11 family of computers shares a common architecture. They are all based on a 16-bit word length, a common instruction set, and the same addressing techniques. They also share the same data management utilities, the same input/output (I/O) systems, and the same programming languages. If you have learned to program one computer in the PDP-11 family, you can easily program another member of the PDP-11 family. The software written for one member of the PDP-11 family will run on other family members. The peripherals and I/O systems for the PDP-11 family are also largely compatible. Because the PDP-11 family has been around since 1970, its hardware and software are proven, time-tested, and thoroughly debugged.

DIGITAL has shipped over 300,000 PDP-11 computers since 1970. The success of the PDP-11 family of computers is a function of their common architecture that provides compatibility across all models: from small, single-user, chip-based microcomputers to large minicomputers that support timesharing services. *By providing its customers with a clear growth-path to expand their computer systems as their requirements expand, DIGITAL has built more minicomputers and more 16-bit microcomputers than any other manufacturer.* Within the PDP-11 family of computers, you can upgrade or extend any DIGITAL system by adding memory, peripherals, or processors without worrying about major incompatibilities in your computing system. And your personnel who are already familiar with the PDP-11 environment will lose no time recoding or learning about another architecture.

Compatibility

DIGITAL's PDP-11 architecture gives you compatibility . It is your assurance that no matter what system you choose or how you decide to mix and match software, peripherals, or CPUs, you will be using proven products that were designed from the beginning to be part of a family. Your application or your business can grow while continuing to use existing software. You can start with a small system and migrate upward to more powerful PDP-11s or eventually choose to build a system with one of DIGITAL's super-minicomputers in the VAX-11 series. Your PDP-11s will function superbly as front ends or as distributed processing nodes in a VAX-11 environment.

Because of this common architecture, you can match a PDP-11 system to your job. A solution can be tailored to your needs, today. As your workload grows, you can expand your computing capability since nearly all of the peripherals and software will work with any PDP-11 processor.

Peripherals

Your selection of PDP-11 peripherals is impressive. 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 low-cost cassette-tape devices through high-capacity Winchester disks, and from intelligent, rugged DECwriters for hard copy, to human-engineered video display terminals. You can also choose from a variety of peripheral products developed and supported by third parties. Either way, there is a complete spectrum of peripheral devices available to complement the software, and provide the complete answer to customer needs in all market areas: business, education, industry, laboratory, and engineering.

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

Networking

DIGITAL's networking capabilities, both hardware and software, are unsurpassed in the industry. You can have a mix of PDP-11 systems doing different jobs, communicating among themselves or with other DIGITAL systems or with equipment from other manufacturers.

Software and Operating Systems

The large installed base of PDP-11 computers means that software for your application will be easy to find. You can choose from proven software developed and supported by DIGITAL or you can choose from software supported by third parties. An important source of software is DECUS—The Digital Equipment Computer Users Society. DECUS is one of the largest and most active user groups in the computer industry, with over 60,000 members world-wide. Membership is free to owners of DIGITAL computers. The DECUS program library contains over

1700 software packages written and submitted by members and DIGITAL employees, and available for the cost of media and copying only.

The PDP-11 family of processors supports a complete range of compatible operating-systems:

- realtime multitasking
- multiuser, interactive timesharing
- small and midrange commercial
- multiuser data management
- development and runtime for microcomputers

For more information about PDP-11 software and operating systems, see the PDP-11 Software Handbook. It includes a general description of each operating system, the file structures, and data handling facilities, the user interfaces, the system utilities, and the language processors supported.

Today's PDP-11s provide a full product range—from chips to systems to networks. The differences among the various PDP-11 processors are primarily internal communications (bus) structure, size, and processing power. PDP-11 processors support multiple operating systems so that the right hardware, operating system, and application software can be combined to meet your exact requirements. The same software that runs on our smaller, low-priced PDP-11 systems will run on our larger models.

PDP-11 ARCHITECTURE AND SYSTEM PERFORMANCE

Both the technology and the architecture of a computer system are important to its performance. The architecture, and particularly the Instruction Set Processor (ISP) of the PDP-11 family of computers have been designed with performance in mind. The PDP-11 architecture makes it possible to perform more functions with fewer instructions. This means that comparing the instruction execution speeds of computers from different vendors may be misleading. Even though a processor may execute individual instructions more slowly, it may execute an application more quickly because its instructions are more powerful. More powerful instructions give PDP-11 computers a significant advantage.

Several factors affecting performance were incorporated into the PDP-11 architecture:

- Bit efficiency

- Simple, yet powerful instruction set
- Addressing capability

A bit-efficient architecture allows the computer to execute an algorithm with fewer instructions bits. Bit efficiency is a function of the number of bits in the instruction word and the number of operations performed for each instruction. A computer with a large instruction word may be more bit efficient than a computer with a small instruction word if the computer can do an equal number of operations with far fewer instructions.

The benefits of bit efficiency are small program size and high execution speed. Programs can be smaller because fewer bits are needed to perform a given operation. Compact programs are more likely to fit into high-speed, on-board memory. Also, fewer memory references are required to fetch program instructions.

The PDP-11 instruction set is powerful, yet simple to use and learn. It lets the programmer address different data types the same way. To save memory space and simplify control and communications, PDP-11 instructions allow byte and word addressing. Another mechanism for saving memory space and program code is the ability of single instructions (with double operands) to perform several operations. A full set of conditional branch instructions foster structured programming by helping avoid the use of jump instructions.

The PDP-11 architecture has an elegant addressing capability that is flexible and simple. It uses the same instruction to address a processor register, main memory, or an I/O device. No distinction is made between data and address locations, even in the processor registers. This can be helpful when manipulating arrays, for example. In a system with dedicated data and address registers, an array subscript must often be created in data registers before it can be copied to address registers to access the operand. This transfer from data to address requires additional program code which can reduce system performance. DIGITAL's PDP-11 addressing modes avoid this problem.

EVOLUTION OF THE PDP-11

For the past 25 years, DIGITAL has refined the PDP-11 computer family. We have reduced the size and cost of new family members that provide the performance and functions usually found only in much more expensive computers. The PDP-11 family grew out of experience with the PDP-8—DIGITAL's first mass-produced minicomputer. The PDP-8—a 12-bit, single-address computer—was originally designed for process-control and laboratory applications. It was also used for message switching and other realtime applications. The PDP-8 pioneered the idea of using a minicomputer for small, general-purpose time-sharing.

The first PDP-8 system was shipped in April 1965. Within 15 years, over 50,000 PDP-8 family computers were produced, and the design was improved ten times to use the latest technologies.

Out of the PDP-8 experience, DIGITAL engineers planned the next generation of minicomputers—the PDP-11 family—around these features:

- growth path within the family
- ease of programming
- faster interrupt handling
- more registers
- byte and string handling
- more physical memory
- flexible addressing modes
- support for applications based in read-only memory (ROM)
- better I/O processing

Growth Path—The PDP-11 family succeeded in this area beyond the goals of the original design group. Counting the VAX/VMS with PDP-11 compatibility mode, there are now twenty members of the PDP-11 family. The PDP-11 family offers a range of performance and memory—with compatibility—unprecedented in the industry.

Ease of Programming—The compatibility of PDP-11 family processors makes it very easy to switch from one processor to another.

Faster Interrupt Handling—This problem was solved by the UNIBUS interrupt vector design. This fast mechanism requires only four memory cycles from the time an interrupt request is issued until the first instruction of the interrupt routine begins execution. This fast context switching gives 16-bit PDP-11 computers better realtime performance than some 32-bit computers.

More Registers—Other minicomputers had skimped on registers; the PDP-11 architecture called for eight 16-bit registers. Later, six 64-bit registers were added as accumulators for floating-point arithmetic.

Byte and String Handling—The PDP-11 architecture provided for direct byte addressing from the beginning. In 1977, string handling was added with the Commercial Instruction Set (CIS).

More Physical Memory—As the PDP-11 family outgrew the original 16-bit address space, memory management was added, allowing 22-bit addressing (up to four Mbytes).

Flexible Addressing Modes—The PDP-11 architecture uses the autoincrement/autodecrement addressing mechanism in lieu of a hardware stack. This successful PDP-11 solution has been widely copied in the industry.

Provision for ROM—PDP-11s make extensive use of read-only memories for bootstrap loaders, program debuggers, and simple functions. Most code written for PDP-11s is reentrant without special effort by the programmer.

Better I/O Processing—The PDP-11's improved interrupt structure greatly enhances its I/O capabilities. The LSI-11 Bus includes block mode data transfer to reduce CPU overhead during I/O. All PDP-11 family computers provide Direct Memory Access (DMA) for high-priority communications with memory.

The design goals of the PDP-11 engineers were not all realized with their first production model. The PDP-11 architecture provided a solid foundation for family growth. As you can see from the list that follows, the PDP-11 architecture was extended by new instruction sets for floating point and commercial applications. PDP-11 processors were reengineered to have better performance, smaller packaging, and more attractive prices.

PDP-11 MILESTONES

1970

- UNIBUS
  - Byte (8-bit) or word (16-bit) addressing
  - Consistent addressing
  - Interrupt capabilities
- Extended Arithmetic Element (EAE)--hardware multiply and divide
- Eight General Purpose Registers (GPRs)

1972

- Floating-point processor
  - 6 registers
  - 46 instructions
- Fastbus (PDP-11/45)

- Memory Management (KT11C)
- Fully protected multiprogramming with three access modes:
  - Kernel
  - Supervisor
  - User
- Second set of GPRs for a total of 16 (PDP-11/45)
- Programmed interrupt request

1973

- Extended Instruction Set (EIS) for multiply and divide
- Floating Instruction Set (FIS)

1975

- LSI-11 "computer-on-a-board"--first 16-bit microcomputer
- 22-bit addressing for processor, peripherals (PDP-11/70)
- 32-bit wide DMA bus (PDP-11/70)

1976

- Fast, all bipolar memory (PDP-11/55)

1977

- LSI-11/2 offers LSI-11 performance in half the space (a double-height board 5.2 x 8.5 inches or 13 x 22 cm)
- Commercial Instruction Set (CIS):
  - Character sets and strings
  - Packed and zoned decimal strings
  - Variable length strings
- Writable Control Store (WCS) extends function code to invoke user-written microcode (PDP-11/60)
- Remote diagnosis (PDP-11/70)

1978

- Virtual Address eXtensions (VAX)—a 32-bit super-minicomputer including PDP-11 compatibility mode

1979

- LSI-11/23 offers 2.5 times the operating speed of the LSI-11/2 in the same board area

1980

- PDP-11/44 offers new levels of performance in its price range:
    - Winchester disk support
    - Up to four Mbytes of main memory

1981

- FALCON SBC-11/21--designed for dedicated, ROM-based, real-time applications--is the smallest 16-bit single-board microcomputer in the industry
- LSI-11/23 offers memory management--22-bit addressing of up to four Mbytes of main memory--and RSX-11M support for realtime applications
- PDP-11/23 PLUS supports up to one Mbyte of parity MOS memory
    - Supports RSX-11M-PLUS
    - CIS option for COBOL-81
- PDP-11/24--the newest and smallest UNIBUS processor
    - Supports up to four Mbytes memory
    - Winchester disk support

1982

- Professional 300 series personal computers
    - Based on PDP-11 architecture
    - Multitasking operating system
    - Software development on PDP-11s or VAXs

- MICRO/T-11--DIGITAL's first 16-bit microprocessor on a single 40-pin chip
    - PDP-11 instruction set

- MICRO/J-11 offers the performance of the 11/70 in a 60-pin package
    - On-chip memory management addresses up to 4 Mbytes
    - 46 floating-point instructions standard

- MICRO/PDP-11--for customers who need a low-cost, Winchester-based, PDP-11 system
  - — 10 Mbyte Winchester system disk
  - — All PDP-11 software available

EVOLUTION OF THE LSI-11

Introduction

In recent years, minicomputers have been adapted to a wide variety of applications. They have displaced larger computer systems in many traditional markets. At the same time, they have opened up many new markets, primarily because of their low cost, small size, and ease of use. Still, in spite of this remarkable success, minicomputers are not without competition.

In cost-sensitive areas, the minicomputer is being eased out of its dominant position by a new generation of VLSI (Very-Large-Scale Integration) microcomputers. These new "processors on a chip" have found a warm reception from designers seeking inexpensive computing power. That warm reception sometimes cools, however, when the user finds himself with a collection of components, instead of a complete computing system. The discovery that he is largely on his own when it comes to software and debugging support has a similarly chilling effect. The entry into the world of programming PROMs, using FORTRAN cross-assemblers and simulators, and writing even simple software routines from scratch can be a traumatic experience indeed. Still, the advantages of LSI microcomputers are very real, and many users have found the difficulties worthwhile. However, some users wonder why they cannot have the best of both worlds: the low cost and small size of the microcomputer, and the ease of software development and performance of the minicomputer systems with which they are familiar.

Therefore, the appearance of new LSI microcomputer systems that are fully compatible with a line of 16-bit minicomputers was a significant event. The first of these new microcomputers was the DIGITAL LSI-11, a complete 4K PDP-11 on a 21.6 X 26.7 cm (8.5 X 10.5 inch) board. Priced to compete with other LSI microcomputers, it offered true minicomputer performance with the highest levels of support. While not intended to be yet another low-end minicomputer, the LSI-11 brought many minicomputer strengths to its new microcomputer applications.

To provide minicomputer performance at a microcomputer price, the LSI-11 was designed to optimize system costs, rather than component costs. A one-chip central processor, then, was not necessarily superi-

or to a four-chip one--the choice was made on the basis of total system cost and performance. On this basis, a microprogrammed processor was selected, permitting the inclusion of features like a "zero cost" realtime clock and automatic dynamic memory refresh. The built-in ASCII programmer's console was also made feasible by the microprogramming feature.

Awareness of system costs and performance were the primary motivations in designing the LSI-11. System issues included:

- Cost of ownership
- Ease of interconnection
- Preservation of customer's training and software investment
- Availability of proven peripherals and software

All these issues dictated PDP-11 compatibility. The LSI-11 microcomputers use the PDP-11 architecture, including the PDP-11 instruction set and addressing modes. They use a bus structure based on the PDP-11 UNIBUS, but smaller and less expensive–the LSI-11 Bus.

DIGITAL's next advance in LSI technology was the LSI-11/2, which offered the performance of the LSI-11 in one half the board size. The LSI-11/23 maintained the LSI-11/2's board size but more than doubled its operating speed. Next came the FALCON Single Board Computer (SBC-11/21), which was the industry's smallest 16-bit computer on a board.

Recent Trends in PDP-11/LSI-11 Development

The most recent announcements in the LSI-11/PDP-11 family have focused in three areas: personal computers, chipsets, and microcomputer systems. DIGITAL's range of personal computers extends from the-powerful, PDP-11 based Professional 300 series to word processing and accounting with the DECmate II and to systems running industry-standard software with the Rainbow 100. (DECmate II and Rainbow 100 are not PDP-11 - based products.)

DIGITAL's 16-bit chipset offerings provide levels of price and performance to meet any application. From the low-cost MICRO/T-11 microprocessor in a 40-pin chip, to the ultimate in 16-bit, single-package microprocessor performance--the MICRO/J-11.

DIGITAL is also presenting a family of systems between the personal computers and the LSI-11/PDP-11 computers--the MICRO/PDP-11. An aggressively-priced member of our proven PDP-11 family, the multiuser MICRO/PDP-11 features compact microcomputer packaging for the office environment. The single-user MICRO/PDP-11 is optimized for the technical environment.

Future Directions

The evolution of PDP-11 systems offers a striking demonstration of the impact of technology and architecture on a computer family. While maintaining a consistent architecture, PDP-11 computers have incorporated increasingly sophisticated technology to provide better performance at lower cost. DIGITAL is continuing to develop its 16-bit products. We will continue to lead this market by lowering the cost and extending the range of PDP-11 computing. Simplicity and reliability of design will continue to lower our cost of ownership.

THE PDP-11 FAMILY ALBUM

DIGITAL is the only major vendor to sell products with compatible hardware and software at the chip, board, box, and system levels. The latest generation of PDP-11 family members are described in this section. Products are divided into these categories:

- Microprocessor chipsets
- Microcomputer boards
- Personal computers
- Microcomputer systems
- Minicomputer (UNIBUS) systems

Chipsets

The **MICRO/T-11** is DIGITAL's first single-chip microprocessor. This chip complements DIGITAL's board, box, and system products by offering customers any level of integration. The MICRO/T-11 is a 16-bit microprocessor in a 40-pin chip. Through the use of a programmable mode register, MICRO/T-11 can be adapted to a wide variety of applications. By selecting either static or dynamic memory and either 8-bit or 16-bit mode, the designer determines the functions of mode-dependent pins. OEMs will find DIGITAL's MICRO/T-11 chip products to be a solution that meets size requirements while utilizing the PDP-11 base-level instruction set and powerful interrupts. The ability to migrate from PDP-11 products down to the chip level is an advantage for designers familiar with the PDP-11 instruction set and development tools. Key features include:

- 16-bit microprocessor in a single 40-pin chip
- Selectable 8-bit or 16-bit data bus

- Dynamic RAM refresh capability
- PDP-11 instruction set and addressing modes

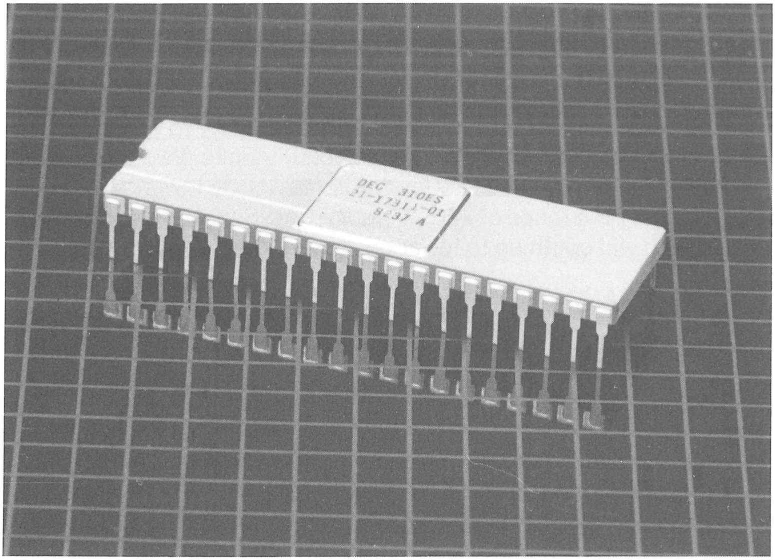


Figure 1-1 The MICRO/T-11 Chip

The **MICRO/J-11** offers the performance and architecture of the PDP-11/70 in a single 60-pin package. Based on CMOS technology, the MICRO/J-11 has 16-bit I/O, a 32-bit internal data path, and can address up to 4 Mbytes of memory with on-chip memory management. The MICRO/J-11 implements the full PDP-11 instruction set including hardware multiply/divide (EIS), FP11 floating-point (46 instructions), and MICRO Online Debugging Task (ODT). Key features include:

- 16-bit I/O
- 32-bit internal data path
- On-chip memory management to address up to 4 Mbytes memory
- Full PDP-11 instruction set
- 46 floating-point instructions standard
- Extended instruction set standard

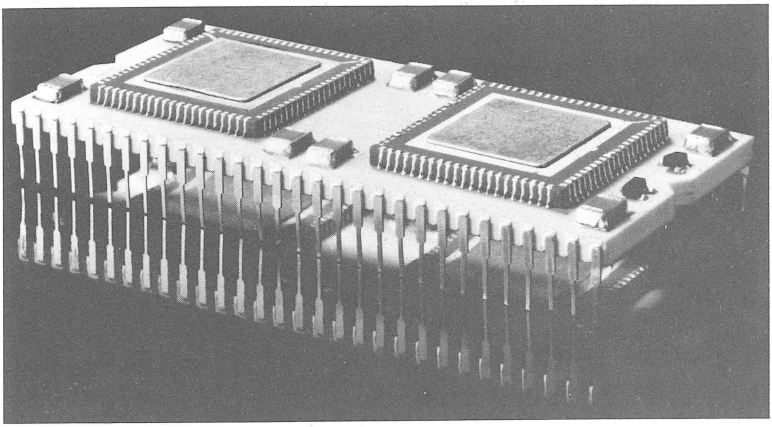


Figure 1-2 The MICRO/J-11 Chipset

Boards

The **LSI-11/23** offers 2.5 times the operating speed of the LSI-11/2 in the same board area. The LSI-11/23 approaches the performance of mid-range minicomputers in a single board. Its 22-bit addressing capability lets the LSI-11/23 address four Mbytes of main memory. Its comprehensive memory management feature provides memory relocation, segmentation, and protection for this extended address range. Key features include:

- Extended LSI-11 Bus for 22-bit addressing
- RSX-11M-PLUS support for realtime applications
- Full memory management

The **FALCON SBC-11/21**--the smallest 16-bit single-board microcomputer in the industry--was designed for dedicated, ROM-based, realtime applications. The FALCON offers more on-board RAM and ROM memory than any other DIGITAL microcomputer. It features two asynchronous serial I/O ports with eight programmable baud rates, 24 parallel I/O lines, and a crystal-controlled realtime clock. The FALCON packs all this computing power onto a 44 square inch board. Key features are:

- Most on-board RAM and ROM of any DEC microcomputer
- Two asynchronous serial I/O ports (selectable baud rates)
- 24 parallel I/O lines
- Crystal-controlled realtime clock

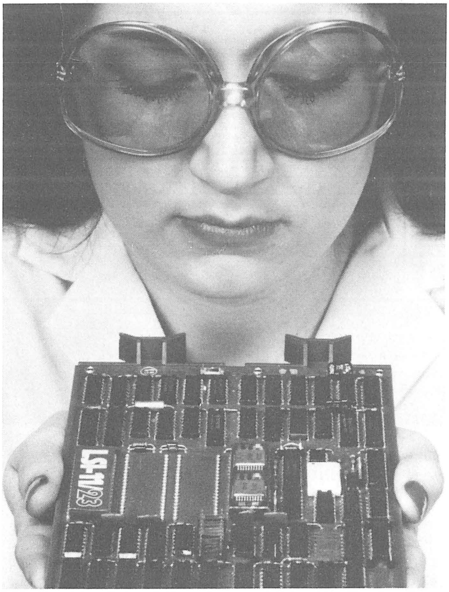


Figure 1-3 The LSI-11/23 Board

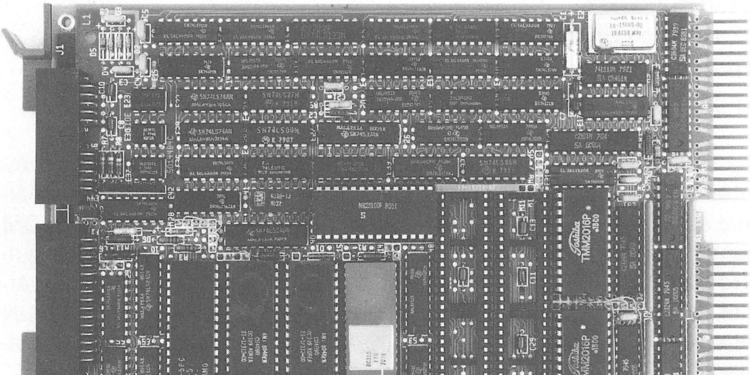


Figure 1-4 The FALCON SBC-11/21 Single Board Computer

Personal computers

Recently, DIGITAL introduced a complete range of personal computers:

- Professional 300 series with multitasking operating system and software development on PDP-11s or VAXs
- DECmate II for word processing and general accounting
- Rainbow 100 for CP/M™ applications

CP/M is a trademark of Digital Research, Inc.

The **Professional 300** series are full-fledged members of the PDP-11 family. The microprocessor chip used in the Professional 325 and 350 is the F-11, the same chip used in the LSI-11/23 and the PDP-11/23 PLUS. This gives the user true minicomputer performance in a desktop personal computer. As a PDP-11 family member, the Professional 300 personal computers incorporate the RSX-11M-PLUS operating system into their operating system. The Professional 300 personal computers can transfer files to any DIGITAL computers running RSX-11M, RSX-11M-PLUS, or VAX/VMS (using optional communications software). Also, applications for the Professional may be developed on a PDP-11 or VAX host system for debugging on the Professional 350. The Professional computers are very easy to use, thanks to their menus, help service, file and disk services, and editor. The optional Telephone Management System for the Professional 350 allows automatic personal and computer-to-computer dialing.

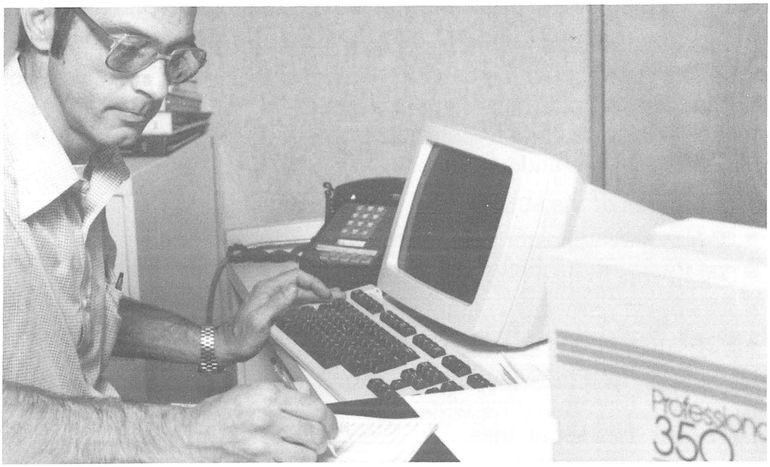


Figure 1-5 The Professional 350 Personal Computer

Microcomputer systems

The **MICRO/PDP-11** is a system for technical and commercial customers who need more performance than a personal computer, but lower cost than an LSI-11 or PDP-11 system. Unlike typical desktop microcomputer systems, the MICRO/PDP-11 provides ample performance to handle small business, departmental, or technical applications. It shares much of the hardware and software found in DIGITAL's larger minicomputer systems. A full-fledged PDP-11 processor, the MICRO/PDP-11 is based on the PDP-11/23 PLUS processor. It includes a 10 MByte, 5¼ inch (13.1 cm) Winchester disk and 800 Kbyte, dual 5¼ inch dual diskette subsystem for backup and media interchange.

Users can choose from a wide range of PDP-11 operating systems to match their needs–RSTS/E, RSX-11M-PLUS, CTS-300, RT-11, Micro-Power/Pascal, UNIX™ , and DSM-11.

UNIX is a trademark of Bell Laboratories.

The powerful PDP-11/23 PLUS CPU supports full 22-bit, 4 Mbyte addressing, which means that large applications will run on the MICRO/PDP-11. For extra computing power in specialized applications, two optional microcoded chips are available: one for floating-point data, and one for the Commercial Instruction Set (CIS). For even faster floating-point performance,a separate floating-point processor is available.

Attractive packaging in three configurations—table top, floor stand, and rack mount—allow wide flexibility of installation. Its modular construction permits easy assembly and disassembly with simple tools.

Unlike most microcomputer systems in its price range, the MICRO/PDP-11's communication capabilities are extensive and can easily be enhanced to match the growth of a business. The MICRO/PDP-11 can be fully integrated into a distributed processing environment utilizing DECnet hardware and software.

Key features of the MICRO/PDP-11 include:

- 10 Mbyte Winchester system disk
- 800 Kbytes in dual floppy diskettes
- All PDP-11 software available
- PDP-11/23 PLUS CPU quad module with:
    - full memory management
    - line frequency clock
    - two serial lines
    - user-friendly boot/diagnostics
- Floating-point and CIS options

Figure 1-6 The MICRO/PDP-11 System

The **PDP-11/23 PLUS** gives minicomputer performance at a microcomputer price. Its 22-bit addressing supports up to one Mbyte of parity MOS memory (although it can address four Mbytes). A Commercial Instruction Set option, designed to work with COBOL 81, is standard on commercial systems. A microcoded floating-point option increases computational speed, and if your application demands more, a hardware floating-point option provides even greater performance. The system distribution panel makes installation or relocation easy. The PDP-11/23 PLUS is available with a wide assortment of peripherals, interconnect options, video and hardcopy terminals, and a broad choice of system software to fit your applications.

The compatibility of DIGITAL software from the most powerful PDP-11 system downward to the least expensive means that PDP-11/23 PLUS users can run the RSX-11M-PLUS, RSX-11M, and RSTS/E operating systems used on the most powerful PDP-11s, as well as RT-11. And users can work with the same command language, the same query and report writer, and the same forms manager used on larger PDP-11s.

DECnet Phase III features—adaptive routings, multidrop terminal support, and network command terminals—are available on the PDP-11/23 PLUS.

These features make the PDP-11/23 PLUS an ideal candidate for distributed processing applications in which it can serve as a departmental computer running four to six local terminals. The PDP-11/23 PLUS can communicate with a corporate, divisional, or plant host system through DECnet or through DIGITAL's Internet software that links it with non-DIGITAL computer systems.

Although DIGITAL will install it for you, you can save money by installing the PDP-11/23 PLUS yourself.

Key features of the PDP-11/23 PLUS include:

- Extended LSI-11 Bus for 22-bit addressing
- RSX-11M-PLUS support for real-time applications
- Full memory management
- CIS option for COBOL-81

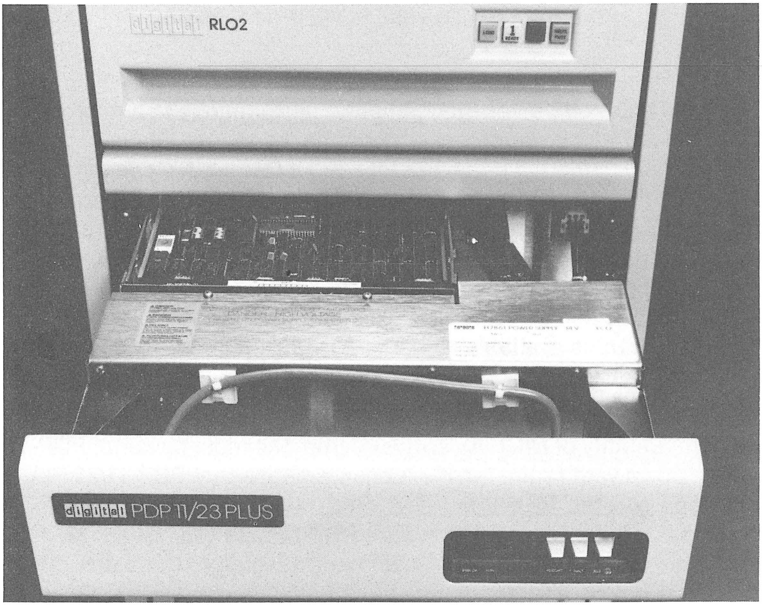


Figure 1-7 The PDP-11/23 PLUS System

Minicomputer (UNIBUS) systems

The **PDP-11/24**--the newest and smallest UNIBUS processor offers midrange capacity at a small-system price. The PDP-11/24 uses LSI technology to provide better performance and memory management capabilities previously available only on larger PDP-11 systems. The 22-bit addressing allows users to address up to 4 Mbytes of main memory permitting more resident tasks, more users, and faster response.

Designed for compactness and reliability, the entire CPU fits on a single board. And the PDP-11/24 is the smallest system that supports UDA50/RA80 Winchester disk technology.

With Winchester technology, very reliable, high-density, nonremovable RA80 disks provide 121-Mbyte storage capacity. The UDA50 UNIBUS disk controller optimizes disk requests so that the hardware automatically schedules requests to multiple RA80s, and handles complete error recovery and buffering between the device and the system.

In addition to its impressive disk throughput performance, the RA80's average seek time is exceptional for a product in its price and capacity range.

A microcoded floating-point option for the PDP-11/24 increases computational speed, and if your application demands more, a hardware floating-point option provides even greater performance. A Commercial Instruction Set option, designed to speed the compilation and execution of COBOL 81, is standard on commercial systems. Key features of the PDP-11/24 include:

- Supports up to four Mbytes memory
- Single-board CPU
- Winchester disk support
- CIS and floating-point options

The **PDP-11/44** offers new levels of performance in its price range. Its outstanding features include a high-speed CPU that can access up to four Mbytes of main memory. A large, eight-Kbyte cache memory with a 275-nanosecond cycle time accelerates program execution and increases system throughput. In addition, the cache helps isolate main memory from CPU fetches, making more I/O bandwidth available to direct memory access (DMA) devices.

The PDP-11/44 meets rigorous reliability and maintainability standards. Its Error Checking Code (ECC) memory detects and corrects errors. A built-in microprocessor controls the ASCII console, provides extensive system diagnostics, and can control a dual TU58 cartridge

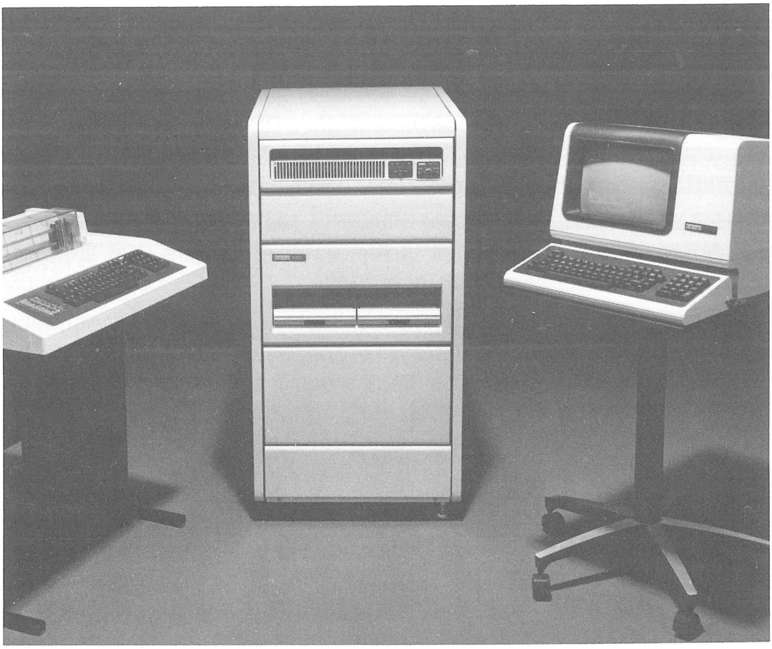


Figure 1-8 The PDP-11/24 System

tape subsystem for loading diagnostic programs should the standard load medium be unavailable. System cabling and mounting are designed for easy access. Remote diagnosis allows problems to be pinpointed quickly, and the correct replacement parts to be dispatched.

Because disk performance can play a key role in a computer's overall performance, particularly in those applications when I/O is large in relation to the amount of computation, the PDP-11/44 supports the new DIGITAL Storage Architecture (DSA). The DSA describes new disks, an intelligent controller, connections, and software protocols for attaching to DIGITAL systems. The disk subsystems of the DSA feature:

- Low cost of ownership per megabyte
- More data storage per square foot of floor space
- Choice of Winchester or removable disks
- Optimized I/O throughput
- Industry's most comprehensive data integrity features
- High availability

Figure 1-9 The PDP-11/44 System

The high performance of the DSA disk products is due both to the technology of the disks themselves and to their intelligent controller, the UDA50. It interfaces DSA disk subsystems to the UNIBUS, and supports up to four disk drives, connected radially. The UDA50 contains a high-speed, 16-bit processor that can handle data rates up to 3 Mbytes per second. The UDA50 permits high-density recording by providing powerful error correcting. It unburdens the host system of the overhead associated with error handling and I/O throughput optimization. Its seek ordering algorithm minimizes seek distances, reduces seek latency, and provides substantial throughput improvement over first-in-first-out (FIFO) servicing. The UDA50 permits overlapped seeks, initiating simultaneous seek operations to all disks with I/O requests to reduce effective seek time in multi-drive subsystems. It allows one disk to transfer data concurrently with ongoing operations on other disks.

The DSA disk drives that are supported by the UDA50 include:

- RA80—121 Mbyte fixed media
- RA60—205 Mbyte removable media
- RA81—456 Mbyte fixed media

The fixed-media of the RA80 and RA81 disk drives incorporate Winchester technology. The high density surfaces of the Winchester disk drives are accessed by twin heads which improve access time and permit the transfer of more data per seek operation. The sealed head disk assembly results in a virtually contaminant-free environment with significantly greater reliability.

The PDP-11/44 systems support the optional Commercial Instruction Set, and an optional floating-point processor. A 64-Kbyte chip memory is included on larger configurations, and is optionally available on others.

PDP-11/44 systems are available with a full complement of mass storage and interconnect options, and a wide choice of system software.

Key features of the PDP-11/44 include:

- Winchester disk support
- Up to four Mbytes of main memory
- Reliability and maintainability features
- Eight-Kbyte cache memory with 275-nanosecond cycle time

CHAPTER 2

KEY ELEMENTS OF PDP-11 ARCHITECTURE

INTRODUCTION

This chapter is a brief introduction to the main elements of PDP-11 architecture. As we introduce each topic in this chapter, we will refer you to a specific chapter for details. Key elements of PDP-11 architecture include:

- Data representation
- Addressing and registers
- The PDP-11 instruction sets
- Traps and interrupts
- Mapping of memory and busses
- PDP-11 bus structures

DATA REPRESENTATION

The PDP-11 architecture accommodates a variety of data types, which may be separated into categories according to the groups of instructions that manipulate them. They are:

- Integer data
- Floating point data
- String data

Integer data types are manipulated by the basic PDP-11 instruction set. The string data types are manipulated by the Commercial Instruction Set, which is offered as an option on some PDP-11 processors. Floating-point data types are manipulated by the Floating-Point Instruction Set (FP-11) which runs on a Floating-Point Processor (FPP). An FPP may be either a separate processor or a microcode option.

Data representation is treated in detail in Chapter 3.

Integer data types include 8-bit bytes and 16-bit words. Integer data types are stored in memory in binary form, which is represented entirely in ones and zeroes. (Computers use binary representation because it is simple: a one can be represented by the presence of a charge or a switch set on, while a zero can be the absence of charge or a switch set off. Thus, a large number could be represented by a series of switches set on or off to represent binary digits.) In an integer data word or byte, the leftmost, or most significant bit (MSB) can be used as a sign bit. The MSB is always zero for positive values and one for negative values.

Floating point data types are the computer's way of handling very large or small numbers. They represent approximations to quantities using a scientific notation consisting of a sign, the exponent of a power of two, and a fraction between .5 (inclusive) and 1.0 (exclusive). The FP11 instruction set provides two types of floating point data, one 32-bits long and the other 64-bits long. The 32-bit data are called single-precision floating, or just floating; the 64-bit data are called double-precision floating or just double.

The instructions that manipulate floating point data are explained in Chapter 6.

String data types may be divided into two categories:

- Character string data
- Decimal string data

Character string data have their own data type in the Commercial Instruction Set (hereafter called CIS). A character string consists of a contiguous sequence of bytes in memory specified by beginning address and length. This data type is useful when representing names, data records, or text. The manipulations done on character strings include copying, searching, concatenating, and translating. A character string that contains ASCII codes for decimal digits is called a numeric string.

The CIS is treated in detail in Chapter 7.

Decimal string data have two data types: numeric strings and packed strings. Both have similar arithmetic and operational properties; they differ primarily in their representation of signs and the placement of digits in memory. Decimal strings are used to represent numbers in decimal form (which may not be used for computation), as opposed to binary integer form.

ADDRESSING AND REGISTERS

Within the processor there are locations called **general purpose registers** (GPRs) that can be used for temporary data storage, addressing, and as accumulators during computations. Eight 16-bit general purpose registers are available for use with the PDP-11 instruction set, but some of these registers have special uses. For example, one register is designated the Program Counter (PC); another is the Stack Pointer (SP).

Any operation performed by the computer can be specified by an instruction. Each instruction specifies:

- Function to be performed (operation code)
- General purpose register to be used in locating the data (operand)

- Addressing mode to specify how the registers are used

The datum being manipulated by an instruction is called the instruction operand. An instruction operand can be located in main memory, in a general register, or in the instruction itself. The method for specifying an operand's location is called the operand addressing mode. These addressing modes use the registers in a variety of ways to locate the operand or its address. Addressing and registers are explained further in Chapter 4.

INSTRUCTION SETS

There are three instruction sets available on PDP-11 processors:

- PDP-11
- Floating-point
- Commercial

The PDP-11 instruction set is standard on all PDP-11 family processors; the Commercial Instruction Set and the Floating-Point Instruction Set are optional on certain processors.

PDP-11 Instruction Set

The PDP-11 instruction set offers a wide selection of operations and addressing modes. There are seven categories of PDP-11 instructions:

- Single-operand
- Double-operand
- Branch
- Jump and Subroutine
- Trap
- Miscellaneous
- Condition code

To save memory space and simplify control and communications, PDP-11 instructions allow byte and word addressing in both single-operand and double-operand formats. Double-operand instructions let you perform several operations with a single instruction. Branch, jump, and subroutine instructions each provide a means for diverting program flow to a specified location. Trap instructions specify another form of change in program flow, but to a predetermined location. Condition code instructions set or clear the condition codes (four bits in the Processor Status Word [PSW] indicating the results of previous instructions).

See Chapter 5 for more information on the PDP-11 instruction set.

Floating-Point Instruction Set

Floating point data types are manipulated by the Floating-Point Instruction Set (FP-11), which runs on an optional floating-point processor, which may be either a separate processor or microcode. (A microcoded floating point processor is standard on the J-11 chipset.)

The Floating-Point Instruction Set is described in Chapter 6.

Commercial Instruction Set

COBOL processing makes extensive use of string data types, which are manipulated by the Commercial Instruction Set (CIS). The CIS is offered as an option on some PDP-11 processors.

The CIS is discussed in Chapter 7.

TRAPS AND INTERRUPTS

Processor Traps

PDP-11 processor traps are triggered by power failures and certain hardware and software errors. Processor traps protect the programmer and the processor. They save the current PC and Processor Status Word (PSW) and pass control to a trap-handling routine. This saves the programmer work. They also protect the processor and the operating system, if the programmer inadvertantly codes an illegal instruction, or an instruction which might violate the integrity of the operating system. A trap causes the processor to execute instructions pointed to by a certain permanently assigned address. **Trap instructions** are used to make an orderly transition to the trap routine and save the context of the CPU.

Interrupts

Interrupts are used by certain system devices to reduce their wait for CPU service. PDP-11 processors offer the programmer fast interrupt handling. Only four memory cycles are required from the time an interrupt request is issued until the first instruction of the interrupt routine begins execution. By using interrupts, the processor is relieved of doing routine control functions for peripheral devices. Instead, the processor can ignore the peripheral, which may be reading a tape or doing some time-consuming operation, until the peripheral is finished and has data ready for the CPU. Then the device will use an interrupt to get the CPU's attention before it can execute the next instruction.

Traps and interrupts are examined in Chapter 8.

MAPPING TO MEMORY AND BUSSES

Memory management matches the virtual addresses generated by the

CPU with physical addresses in memory and with physical I/O bus addresses. It also protects operating system software and shared routines from modification and allocates protected memory space for each user. The UNIBUS map is a hardware device separate from the memory management unit. The UNIBUS map converts 18-bit UNIBUS addresses to 22-bit memory addresses. There is no map on the extended LSI-11 Bus. Processors and peripherals can generate and present 22-bit addresses directly to the extended LSI-11 Bus.

Memory management and bus mapping are described in Chapter 9.

PDP-11 BUS STRUCTURES

The two PDP-11 physical I/O busses—the UNIBUS and the LSI-11 Bus—are both covered in Chapter 10. The brief, tutorial overview of the UNIBUS and LSI-11 Bus found in that chapter is augmented by appendices that contain timing diagrams and technical specifications.

UNIBUS

The UNIBUS, DIGITAL's unique data bus, was the first data bus in the history of the minicomputer industry to enable devices to send, receive, or exchange data without processor intervention or intermediate buffering in memory. The UNIBUS forms the hardware and software backbone of the PDP-11/24 and PDP-11/44 processors. Memory elements on the UNIBUS have ascending addresses starting at zero, while registers storing I/O data or the status of individual peripheral devices have addresses in the highest 8 Kbytes of addressing space. Peripheral devices may have one or more addresses.

LSI-11 Bus

The LSI-11 Bus is the low-end member of DIGITAL's bus family. Most DIGITAL microcomputers use the LSI-11 Bus or the extended LSI-11 Bus. The LSI-11 Bus operates very much like the UNIBUS, but to make it more cost-effective for microcomputer applications, it has fewer signal lines. Both the LSI-11 Bus and the UNIBUS are treated in Chapter 10.

OTHER TOPICS (APPENDICES)

Other topics related to PDP-11 architecture are included in appendices. The topic of each appendix is listed and briefly discussed below.

Assignment of Bus Addresses and Vectors

Appendix A covers both the LSI-11 Bus and the UNIBUS. Topics covered include:

- I/O Page Device Addresses

- Interrupt and Trap Vectors
- Priority Ranking for Floating Vectors
- Floating CSR Address Devices
- Device Addresses

PDP-11 Family Differences

Appendix B contains a family differences table that shows in detail the issues involved in software migration between PDP-11 family members. Any program developed using PDP-11 operating systems with higher level languages will migrate with very little difficulty. Certain assembly language applications may require slight modifications for a smooth migration.

The Floating Instruction Set

The Floating Instruction Set (FIS) is a software option for the LSI-11/2 processor. The FIS consists of four special floating instructions that accelerate floating point calculations. The FIS is covered in Appendix C.

UNIBUS Timing Diagrams

UNIBUS timing diagrams and other technical details are given in Appendix D.

LSI-11 Bus Technical Specifications

Topics covered in Appendix E include LSI-11 Bus timing diagrams, and bus pin-out descriptions.

Programming Techniques

PDP-11 processors offer the programmer a combination of flexibility and power. The instruction set, addressing modes, and programming techniques play together to help you develop new software or use existing software. Programming techniques that pertain to architecture are included in this handbook. These include:

- Stacks
- Subroutine linkage
- Reentrancy

Stacks are a basic element of the PDP-11 architecture. They are areas of memory set aside by the programmer or the operating system for temporary storage and linkage. A stack is handled on a last-in/first-out (LIFO) basis: items are retrieved in the reverse of their storage order. A PDP-11 stack starts at the highest location reserved for it and expands downward to lower addresses as items are added.

Often, one of the general purpose registers must be used in a subroutine or interrupt service routine and then returned to its original value.

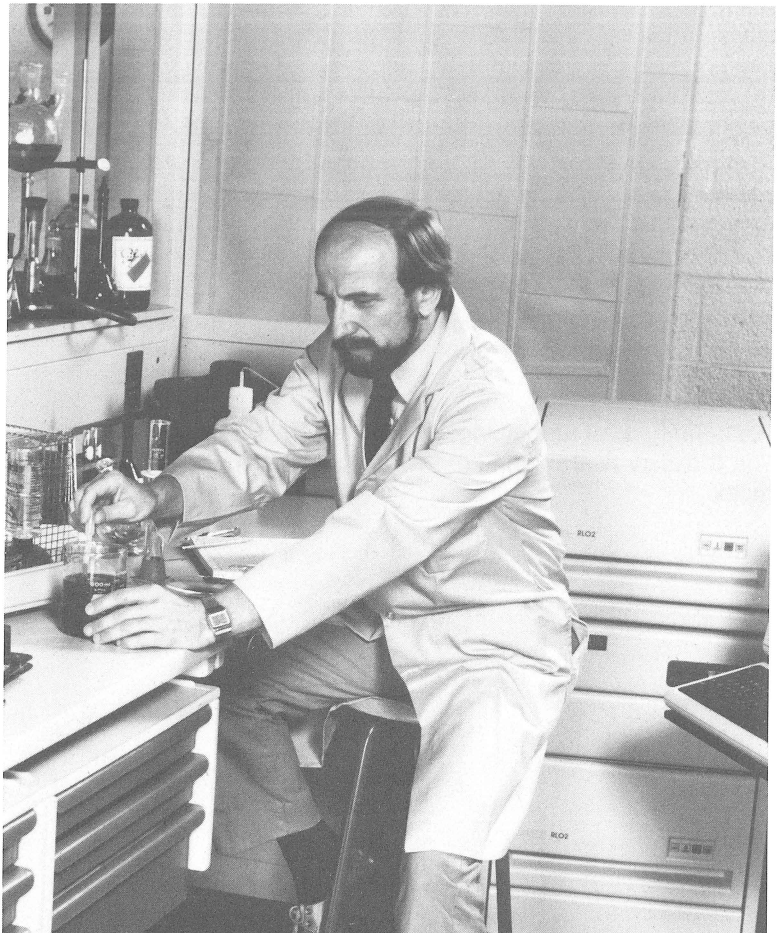
A stack can be used to store the contents of the registers involved. A stack is also useful to store the **linkage** information between a **subroutine** and its calling program. In many cases, operations performed by the subroutine can be applied directly to data located on or referenced by the stack without actually moving the data into the subroutine.

Reentrancy is the ability to share a single copy of a program among different users or different tasks. This makes more efficient use of memory. Reentrant 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.

PDP-11 programming techniques and examples are covered in Appendix F.

Glossary

For definitions of terminology used in this book, refer to the Glossary. The Glossary is at the end of the book, between Appendix F and the Index.


RLO2
RLO2

CHAPTER 3

DATA REPRESENTATION

Data representation is an important aspect of computer architecture. To deal efficiently with different kinds of information, a computer architecture must allow for a range of data types. The programmer's choice of data type should be a function of the application rather than the computer. However, some computers must use nonstandard addressing techniques with certain data types. These computers require more memory and will execute applications more slowly when using these "problem" data types. PDP-11 architecture avoids these compromises. You can use the data type that best suits your application without worrying about nonstandard addressing techniques.

Another feature of the PDP-11 family's data types is upward compatibility. The PDP-11 data types are a subset of the VAX-11 data types. This can be very convenient if you want to transfer your PDP-11 application to an environment with 32-bit addressing.

The PDP-11 data types may be separated into categories according to the groups of instructions that operate on them. They are:

- Integer data
- Character string data
- Decimal string data
- Floating point data

Integer data types are supported by the basic PDP-11 instruction set. The string data types are used by the Commercial Instruction Set, which is offered as an option on some PDP-11 processors. Floating point data types are manipulated by the Floating-Point Instruction Set (FP-11) which runs on a Floating-Point Processor (FPP) which may be either a separate processor or microcode.

The Commercial Instruction Set (CIS-11) is treated in detail in Chapter Seven. The floating point instructions are described in Chapter 6 (The Floating Point Processor—FP-11) and in Appendix C (The Floating Instruction Set—FIS).

INTEGER DATA TYPES

Integer data types include 8-bit bytes, and 16-bit words. Integer data types are stored in memory in binary form, which is represented entirely in ones and zeroes. As unsigned quantities, integers extend upward from 0. As signed quantities, the integers are represented in two's complement form. This means that a negative number is one greater than the bit-by-bit complement of its positive counterpart. Thus, positive numbers have a 0 most significant bit (MSB). The MSB or sign bit is always 1 for negative values.

Byte

A byte is eight contiguous bits starting on an addressable byte boundary or located in a register, Rn <7:0>. The bits are numbered from the right 0 through 7. The byte is specified by its address A. When interpreted as a signed quantity, a byte is a two's complement integer with bits increasing in significance from 0 through 6, and with bit 7 designating the sign. The value of the integer is in the range – 128 through 127.

For the purposes of addition, subtraction, and comparison, PDP-11 instructions also provide direct support for the interpretation of a byte as an unsigned integer with a value in the range 0 through 255.

Word

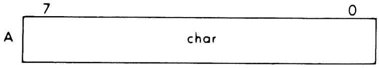
A word, two contiguous bytes, starts on an arbitrary word boundary or is located in a register Rn<15:0>.

Words are specified by their address A, the address of the byte containing bit 0. When interpreted as a signed quantity, a word is a two's complement integer with bits increasing in significance from 0 through 14, and with bit 15 designating the sign. The value of the integer is in the range –32768 through 32767. For the purposes of addition, subtraction, and comparison, PDP-11 instructions also provide direct support for the interpretation of a word as an unsigned integer with a value in the range 0 through 65535.

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."

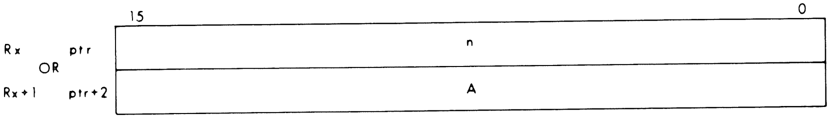
The character is an 8-bit byte:



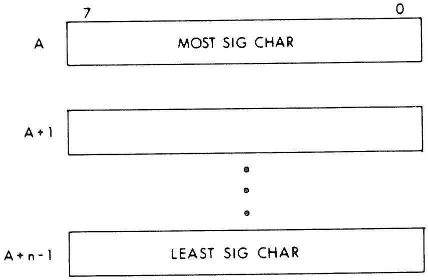
The character is used as an operand by CIS-11 instructions. When it appears in a general register, the character is in the low-order half; the high-order half of the register must be zero. When it appears in the instruction stream, the character is in the low-order half of a word; the high-order half 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 CIS-11 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 zeroth 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 CIS-11 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.

		15 8	7 0
Rx OR	ptr	0	mask
Rx+1	ptr+2	TABLE ADDRESS	

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:

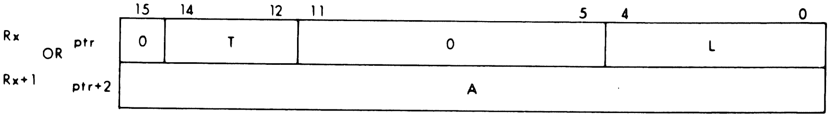
First Word

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.

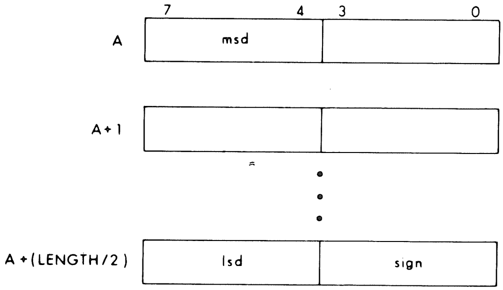
Unsigned Packed Strings — The unsigned sign designator is 1111_2.

PACKED SIGN NIBBLE (in binary):

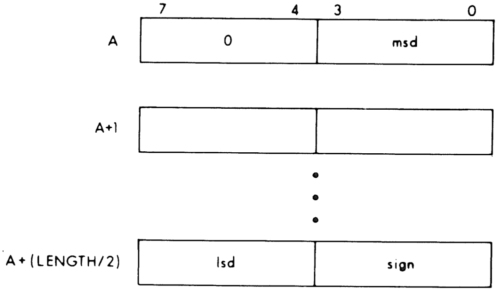
sign nibble	**preferred designator**	**alternate designators**
positive	1100	1010 , 1110 , 1111
negative	1101	1011
unsigned	1111	

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 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).

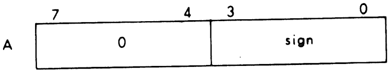
The following is a packed string with an odd number of digits:



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 for destination strings. Bits <3:0> must be a valid sign for source strings, and are used to store the sign of the result for destination strings. When used as a source, zero-length strings represent operands with zero magnitude. When used as a destination, they can only reflect a result of zero magnitude without indicating overflow. The following is a zero-length packed string:



A valid packed string is characterized by:

1. A length from 0 to 31_{10} digits.
2. Every digit nibble is in the range 0000_2 to 1001_2.
3. For even length sources, bits <7:4> of the lowest addressed byte are 0000_2.
4. Signed packed strings—sign nibble is either 1010_2, 1011_2, 1100_2, 1101_2, 1110_2 or 1111_2.
5. Unsigned packed strings—sign nibble is 1111_2.

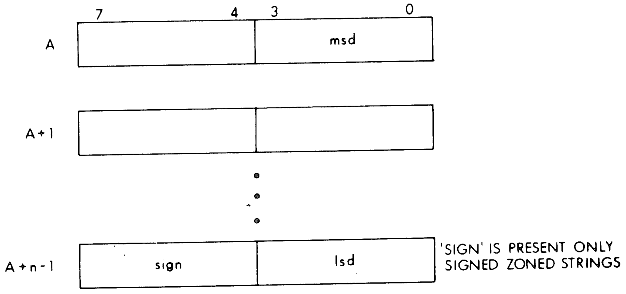
Zoned Strings

Zoned strings represent one decimal digit in each byte. Each byte is divided into two portions—the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low-order nibble contains the value of the corresponding decimal digit.

Signed Zoned Strings — When used as a source string, the high-order nibble of the least significant byte contains the sign of the number; the high-order nibbles of all other bytes are ignored. Destination strings are stored with the sign in the high-order nibble of the least significant byte, and 0011_2 in the high-order nibble of all other bytes. 0011_2 in the high-order nibble corresponds to the ASCII encoding for numeric digits. The positive sign designator is 0011_2; the negative sign designator is 0111_2.

Unsigned Zoned Strings — When used as a source string, the high-order nibbles of all bytes are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all bytes.

The number of bytes needed to contain a zoned string is identical to the length of the decimal number.



A zero-length, zoned string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a 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.

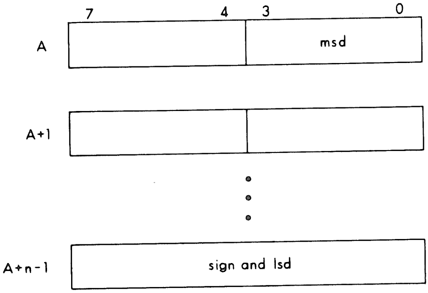
The following table shows the sign of the decimal string and the value of the digit which is encoded in the sign byte. Source strings will properly accept both the preferred and alternate designators; destination strings will store the preferred designator. The preferred designators correspond to the ASCII graphics "A" to "R," "{," and "." The alternate designators correspond to the ASCII graphics "0" to "9," "[," "?," "]," "!" and ":".

OVERPUNCH SIGN/DIGIT BYTE (in binary):

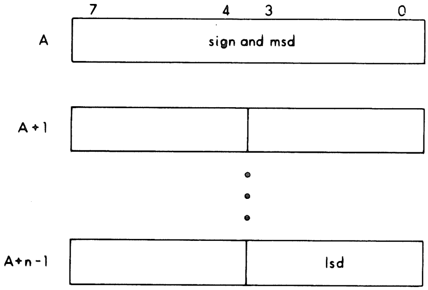
overpunch sign/digit	**preferred designator**	**alternate designators**
+0	01111011	00110000 , 01011011 , 00111111
+1	01000001	00110001
+2	01000010	00110010
+3	01000011	00110011
+4	01000100	00110100
+5	01000101	00110101
+6	01000110	00110110
+7	01000111	00110111
+8	01001000	00111000
+9	01001001	00111001
−0	01111101	01011101 , 00100001 , 00111010
−1	01001010	
−2	01001011	
−3	01001100	
−4	01001101	
−5	01001110	
−6	01001111	
−7	01010000	
−8	01010001	
−9	01010010	

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.

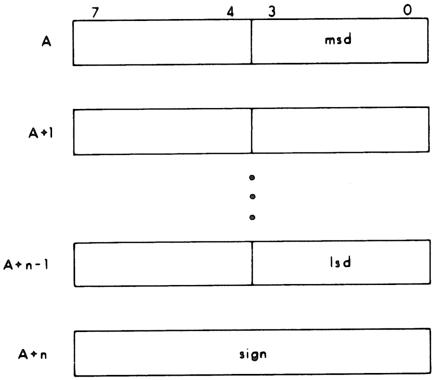
When used as a source string, the high-order nibbles of all digit bytes are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all digit bytes. 0011_2 in the high-order nibble corresponds to the ASCII encoding for numeric digits. The preferred positive sign designator is 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:

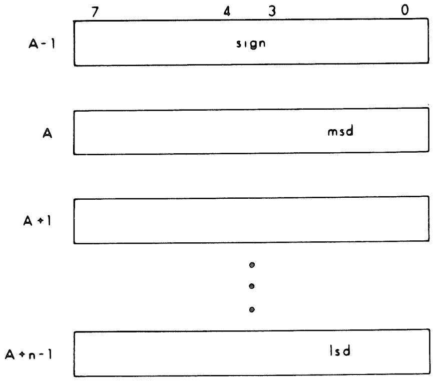
sign byte	**preferred designator**	**alternate designator**
positive	00101011_2	00100000_2
negative	00101101_2	

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:

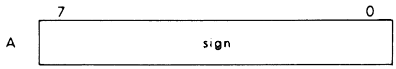


The following is a leading separate string:

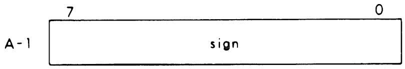


A zero-length separate string occupies a single byte of memory which contains the sign. When used as a source, zero-length strings provide operands with zero magnitude; when used as a destination, they can only reflect a result of zero magnitude without indicating overflow; the sign of the result is stored.

The following is a zero-length trailing separate string:



The following is a zero-length leading separate string:



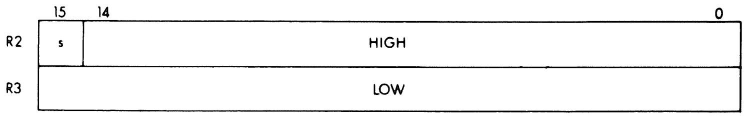
A valid separate string is characterized by:

1. A length from 0 to 31_{10} digits.
2. The low-order nibble of each digit byte is in the range 0000_2 to 1001_2.
3. The sign byte is either 00100000_2, 00101011_2 or 00101101_2.

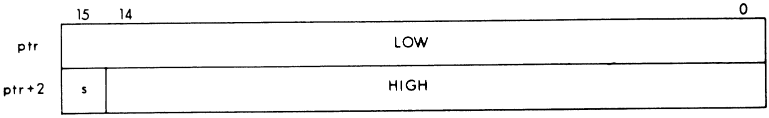
Long Integer

Long integers are 32-bit binary two'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.

FLOATING POINT DATA FORMATS

Floating point data are used only by processors which include a floating point option (standard on the MICRO/J-11). The floating point instruction set (FP11) is covered in Chapter 6 of this book.

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. The value of a floating datum is in the approximate range .29*10** −38 through 1.7*10**38. 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 177_8 is regarded as producing a floating overflow; if the true exponent is less than -177_8, the operation is regarded as producing a floating underflow. A biased exponent of 0 can thus arise from arithmetic operations as a special case of 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:

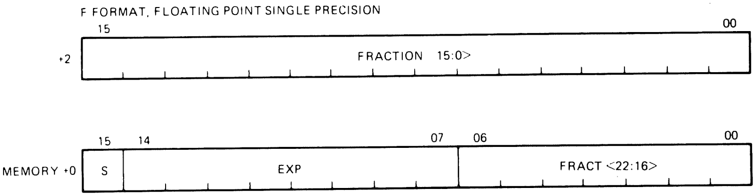
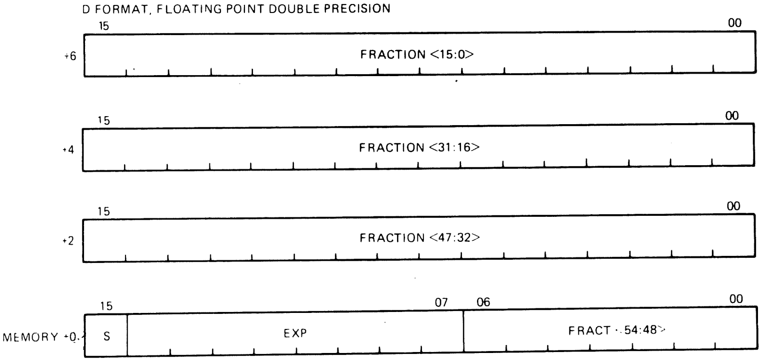


Figure 3-1 Single-Precision Format



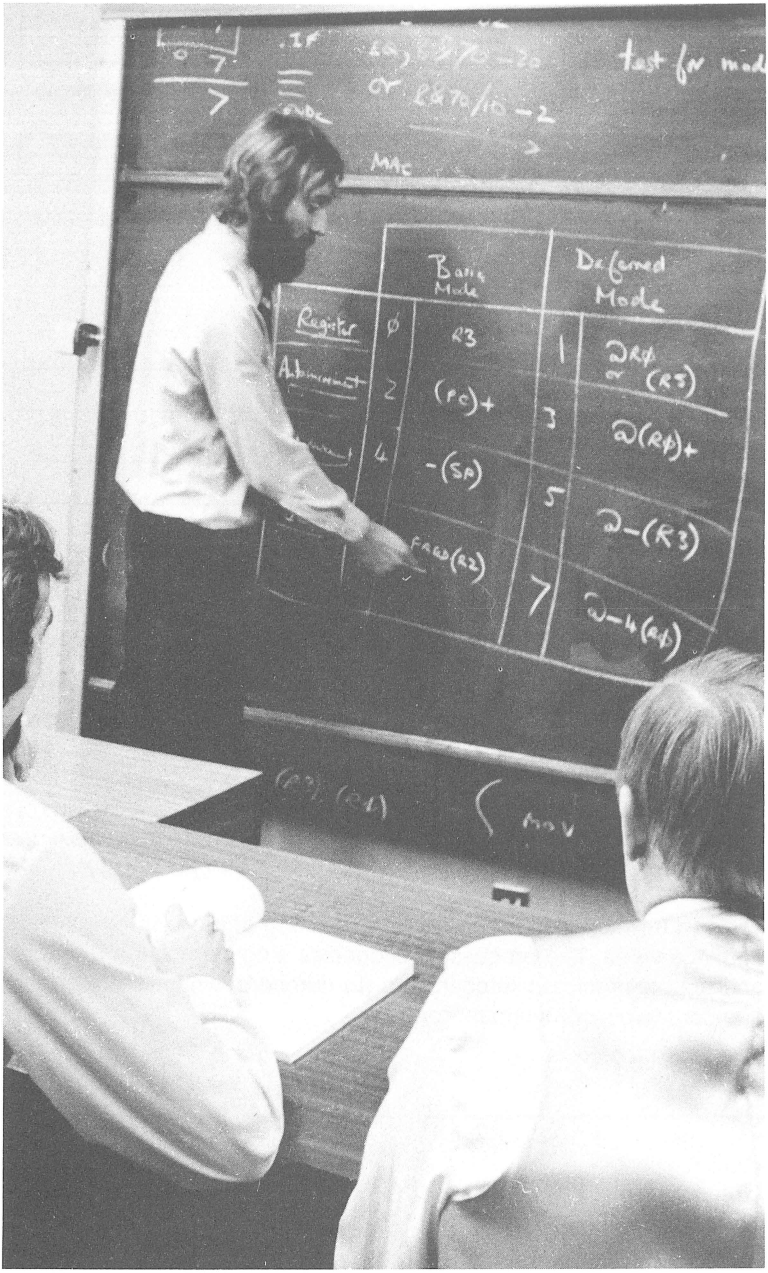
S SIGN OF FRACTION

EXP EXPONENT IN EXCESS 200 NOTATION, RESTRICTED TO 1 TO 377 OCTAL FOR NON VANISHING NUMBERS.

FRACTION 23 BITS IN F FORMAT, 55 BITS IN D FORMAT + ONE HIDDEN BIT (NORMALIZATION). THE BINARY RADIX POINT IS TO THE LEFT.

Figure 3-2 Double-Precision Format

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 two's complement form.


test for mode
MAC
Basic Mode
Deferred Mode
Register
Addressing
ϕ
R3
1
@Rϕ or (R3)
2
(PC)+
3
@(Rϕ)+
4
-(SP)
5
@-(R3)
7
@-4(Rϕ)
(R3), (R4)
MOV

CHAPTER 4
ADDRESSING MODES

In the PDP-11 family, all operand addressing is accomplished through the 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 (where required)
- 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 and destination, 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 Appendix F. 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 Chapter 5 for a description of the full instruction set and its formats.

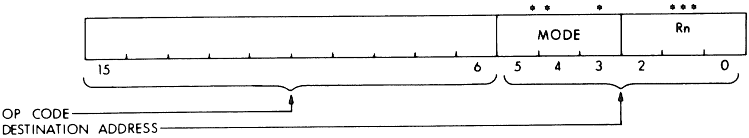
To illustrate the use of the various addressing modes clearly, the following instructions are used in this chapter:

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

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

Single- and double-operand instructions use the following formats:

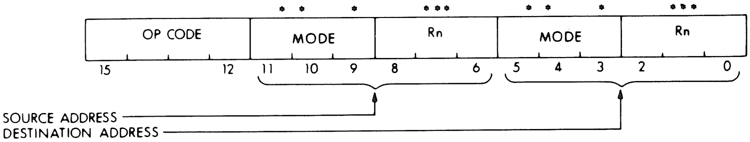
The instruction format for the first word of all single-operand instructions (such as clear, increment, test) is:



* SPECIFIES DIRECT OR INDIRECT ADDRESS
** SPECIFIES HOW REGISTER WILL BE USED
*** SPECIFIES ONE OF 8 GENERAL PURPOSE REGISTERS

Single-Operand Instruction Format

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



* DIRECT DEFERRED BIT FOR SOURCE AND DESTINATION ADDRESS
** SPECIFIES HOW SELECTED REGISTERS ARE TO BE USED
*** SPECIFIES A GENERAL REGISTER

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 a simple accumulator. The operand is contained in the selected register (low-order byte for byte operations). Some assemblers require that a general register be defined as follows:

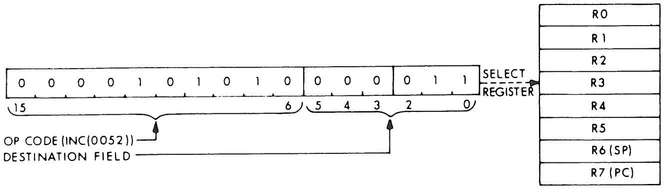
```
R0 = %0
R1 = %1
R2 = %2
```

% indicates register definition (as originally known to the assembler).

Register Mode Example

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

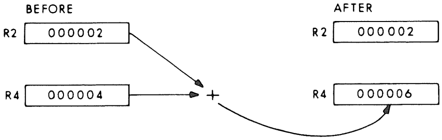
Represented as:



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 or (Rn)

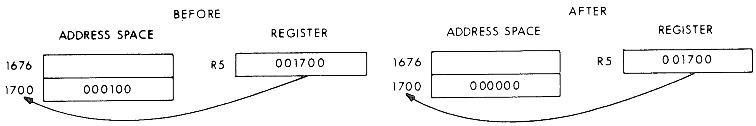
In register deferred mode, the address of the operand is stored in a general purpose register. The address contained in the general purpose register directs the CPU to the operand. The operand is located outside the CPU's general purpose registers, either in memory or in an I/O register.

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)+

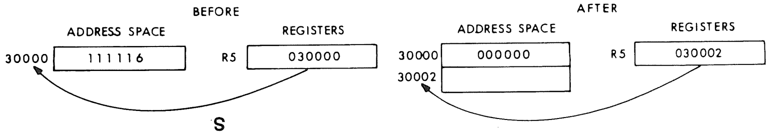
In autoincrement mode, the register contains the address of the operand; the address is automatically incremented after the operand is retrieved. The address then references the next sequential operand. This mode allows automatic stepping through a list or series of operands stored in consecutive locations. 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 one if you are using byte instructions, by two if you are using word instructions. However, R6 and R7 are always incremented by two.

To make it easy to remember that the register is incremented after use, the + sign follows the register name.

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 two.

Represented as:



S

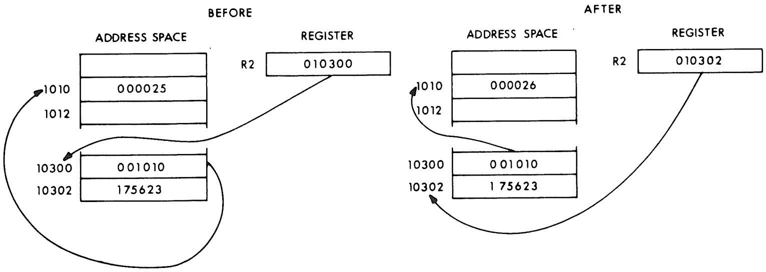
AUTOINCREMENT DEFERRED MODE MODE 3 @(Rn)+

In autoincrement deferred mode, the register contains a pointer to the address of the operand. The + indicates that the pointer in Rn is incremented by two (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, and mode 3 is used to step through a table of addresses that point to data.

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 one, contents of R2 are incremented by two.

Represented as:



AUTODECREMENT MODE MODE 4 −(Rn)

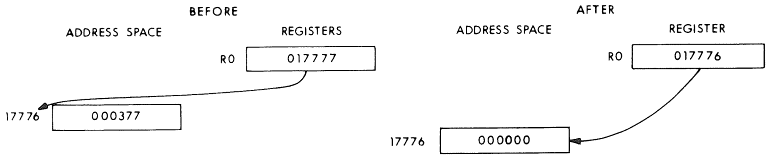
In autodecrement mode, the register contains an address that is automatically decremented; the decremented address is used to locate an operand. This mode is similar to autoincrement mode, but allows stepping through a list of words or bytes in reverse order. The address is decremented by one for bytes,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

To remind you that the register is decremented prior to use, the – sign precedes the register name.

Autodecrement Mode Example

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

Represented as:



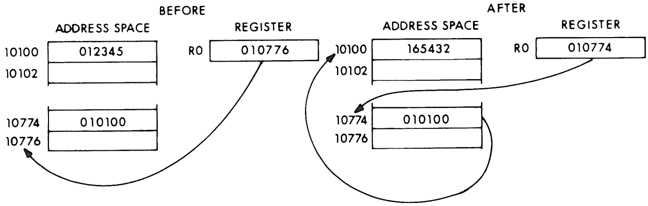
AUTODECREMENT DEFERRED MODE MODE 5 @ –(Rn)

In autodecrement deferred mode, the register contains a pointer to the address of the operand. The pointer is first decremented by two (for both word and byte operations), then the new pointer is used to retrieve an address stored outside the CPU's general purpose registers. 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 two and then used as the address of the address of the operand. The operand is one's complemented.

Represented as:



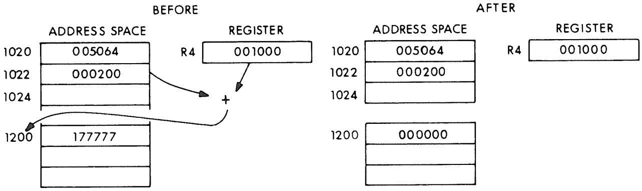
INDEX MODE MODE 6 X(Rn)

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 resulting location is then cleared.

Represented as:



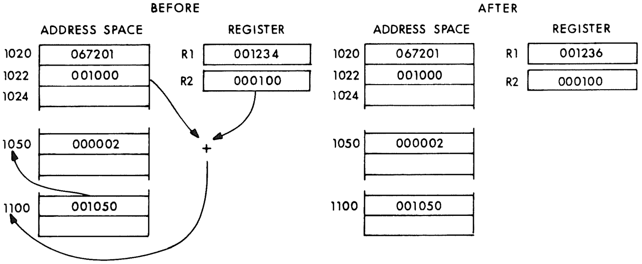
INDEX DEFERRED MODE MODE 7 @X(Rn)

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 to be 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 of the PDP-11 addressing 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.

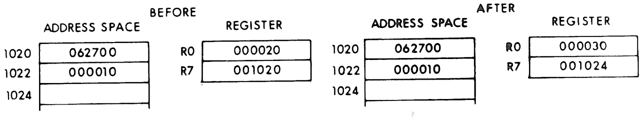
PC IMMEDIATE MODE MODE 2 #n

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:



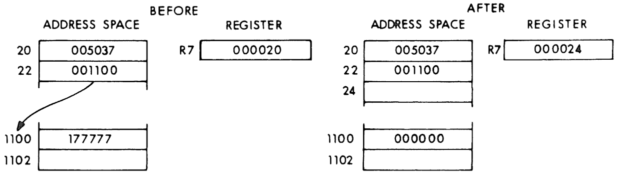
PC ABSOLUTE MODE MODE 3 @#A

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 X(PC) or 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.

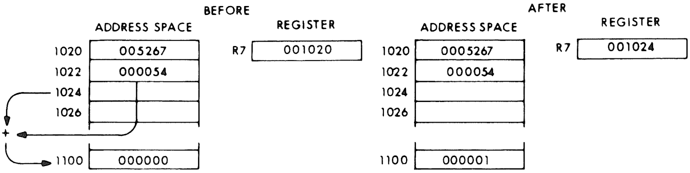
With the relative addressing mode, the address of the operand is always determined with respect to the updated PC. Therefore, if the entire program is relocated, the operand remains the same relative distance away and may be accessed with changing the instruction.

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. It is the default mode generated by the MACRO assembler.

PC Relative Mode Example

Symbolic	Instruction Octal Code	Description
INC A	005267 000054	To increment A, the contents of the memory location in the second word of the instruction are added to the updated PC to produce the address of A (1100). The contents of A are increased by one.

Represented as:



PC RELATIVE DEFERRED MODE MODE 7 @X(PC) or @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.

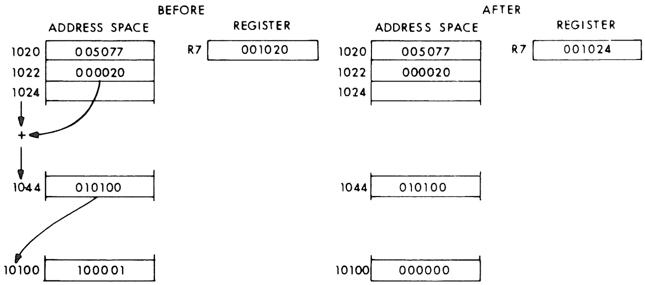
This mode is similar to the relative mode, except that it involves one additional level of addressing to obtain the operand. The sum of the offset and updated PC (PC + 4) serves as a pointer to an address. When the address is retrieved, it can be used to locate the operand.

PC Relative Deferred Mode Example

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

Symbolic	Instruction Octal Code	Description
		tion to the updated PC to produce A—location 1044—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 one for byte and two for word instruction. R6-R7 are always incremented by two.
100	4	Autodecrement	–(Rn)	Register is decremented and then used as a pointer to sequential data.

Basic Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
				R0-R5 are decremented by one for byte and by two for word instructions. R6-R7 are always decremented by two.
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 two, even for byte instructions).
101	5	Autodecrement Deferred	@ – (Rn)	Register is decremented (always by two, even for byte instructions) and then used as a pointer to a word containing the address of the operand.

Indirect Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
111	7	Index Deferred	@X(Rn)	Value X (the index is always found in the next memory location and increments the PC by two) 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, four of these modes are renamed, as you can see in the table below.

PC Register Addressing Modes

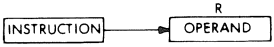
Binary Code	Mode	Name	Symbolic	Function
010	2	Immediate	#n	Operand is contained in the instruction.
011	3	Absolute	@#A	Absolute address is contained in the instruction.
110	6	Relative	A	Address of A, relative to the instruction, is contained in the instruction.
111	7	Relative Deferred	@A	Address of A,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

GRAPHIC SUMMARY OF PDP-11 ADDRESSING MODES

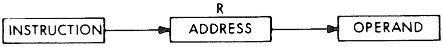
General Register Addressing Modes

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

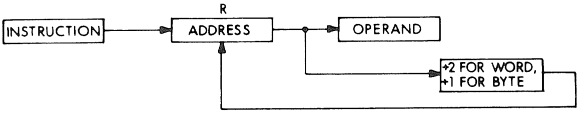
Mode 0 **Register** OPR R R contains operand.



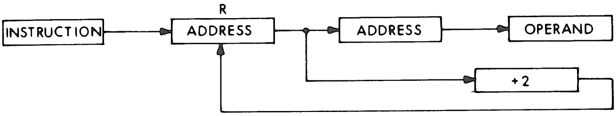
Mode 1 **Register deferred** OPR (R) R contains address.



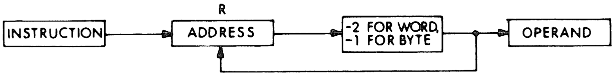
Mode 2 **Autoincrement** OPR (R) + Contents of R are used as address, then increment R . Note that R6 and R7 are always incremented by two.



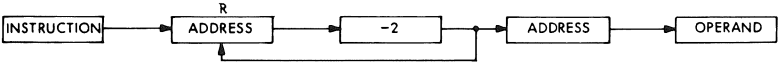
Mode 3 **Autoincrement deferred** OPR @(R) + R contains address of address, then increment R by two.



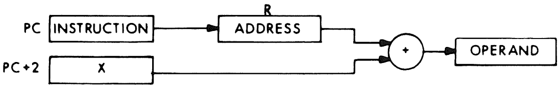
Mode 4	**Autodecrement**	OPR −(R)	Decrement R, then R contains address. Note that R6 and R7 are always decremented by two.



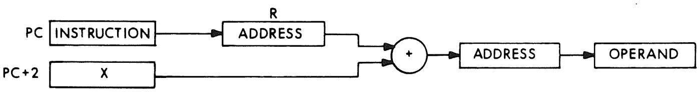
Mode 5	**Autodecrement deferred**	OPR @−(R)	Decrement R by two, then R contains address of address.



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



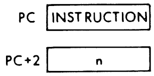
Mode 7	**Index deferred**	OPR @X(R)	R + X is address of address. X is contained in the word following the instruction.



Program Counter Addressing Modes
Register = 7

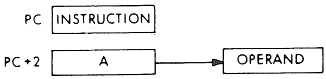
Mode 2 **Immediate** OPR #n

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



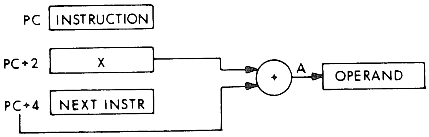
Mode 3 **Absolute** OPR @#A

Address A is contained in the word following the instruction.



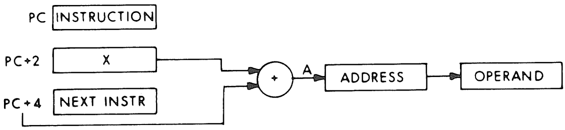
Mode 6 **Relative** OPR A

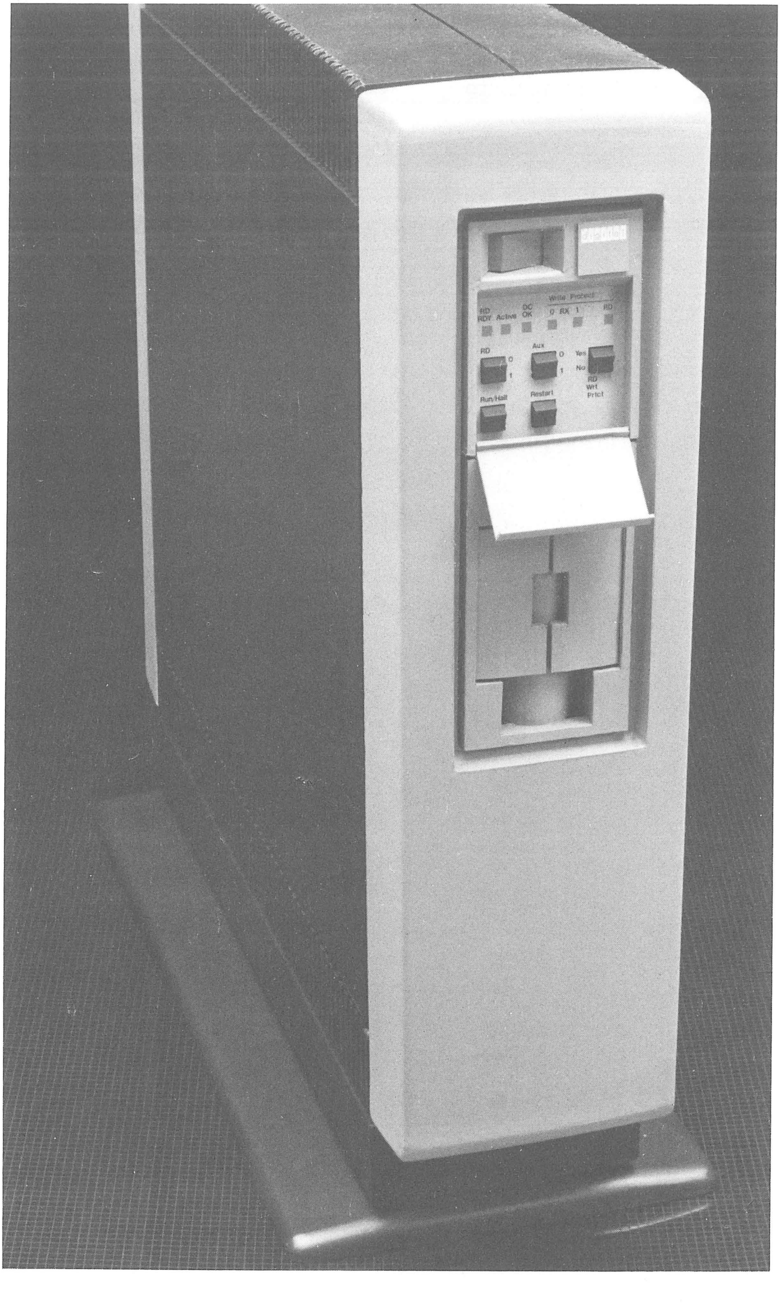
PC + 4 + X is address. PC + 4 is updated PC.



Mode 7 **Relative deferred** OPR @A

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




Write Protect
RD RDY
Active
DC OK
0
RX
1
RD
RD
0
1
Aux
0
1
Yes
No
RD Wrt Prtct
Run/Halt
Restart

CHAPTER 5
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 that 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 destination
	COM(B)	1's complement dst

INC(B)	increment dst
DEC(B)	decrement dst
NEG(B)	2's complement negate dst
NOP	no operation
TST(B)	test dst
TSTSET	test dst, set low bit (MICRO/J-11 only)
WRTLCK	read/lock dst, write/unlock R0 into dst (MICRO/J-11 only)

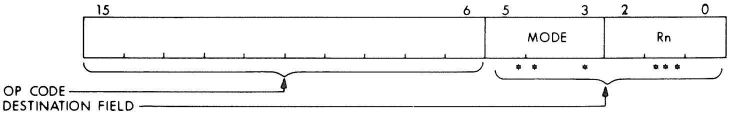
Shift & Rotate

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



* SPECIFIES DIRECT OR INDIRECT ADDRESS
** SPECIFIES HOW REGISTER WILL BE USED
*** SPECIFIES ONE OF 8 GENERAL PURPOSE REGISTERS

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

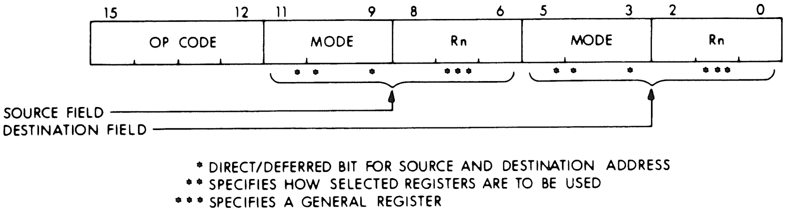

15
12
11
9
8
6
5
3
2
0
OP CODE
MODE
Rn
MODE
Rn
* *
*
* * *
* *
*
* * *
SOURCE FIELD
DESTINATION FIELD
* DIRECT/DEFERRED BIT FOR SOURCE AND DESTINATION ADDRESS
* * SPECIFIES HOW SELECTED REGISTERS ARE TO BE USED
* * * SPECIFIES A GENERAL REGISTER

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

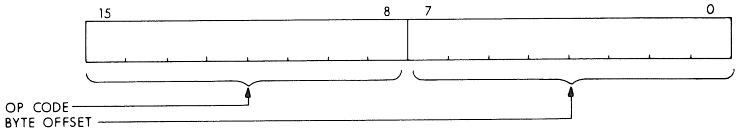


Figure 4-3 Branch Instruction Format

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

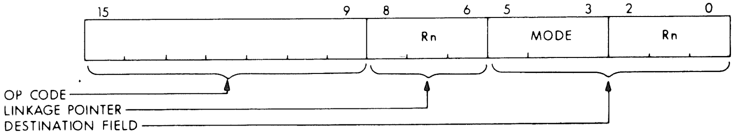


Figure 4-4 JSR Instruction Format

- 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

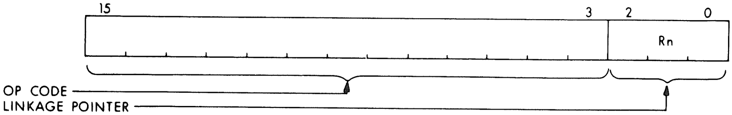


Figure 4-5 RTS Instruction Format

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.

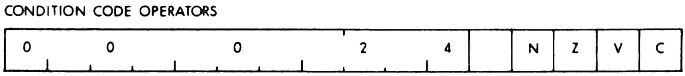


Figure 4-6 Condition Code Operators' Format

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
ADCB
ADD
ASL
ASLB
ASR
ASRB
BCC
BCS
BEQ
BGE
BGT
BHI
BHIS
BIC
BICB
BIS
BISB

BIT
BITB
BLE
BLO
BLOS
BLT
BMI
BNE
BPL
BPT
BR
BVC
BVS
CLR
CLRB
CCC, CLN, CLZ, CLV, CLC
CMP
CMPB

COM
COMB
DEC
DECB
EMT
HALT
INC
INCB
IOT
JMP
JSR
MARK
MOV
MOVB
NEG
NEGBB
NOP
RESET

ROL
ROLB
ROR
RORB
RTI
RTS
RTT
SBC
SBCB
SCC, SEN, SEZ, SEV, SEC
SOB
SUB
SXT
SWAB
TRAP
TST
TSTB
XOR
WAIT

The basic PDP-11 instructions are standard on:

- MICRO/T-11
- MICRO/J-11
- LSI-11/2
- FALCON SBC-11/21 (except for MARK instruction)
- MICRO/PDP-11
- PDP-11/23 PLUS
- PDP-11/24
- PDP-11/44

The PDP-11 compatibility mode on VAX-11 implements all basic PDP-11 instructions except: MARK, RESET, TRAP, WAIT, BPT, EMT, IOT, and HALT.

CSM
Available on MICRO/J-11 and PDP-11/44 only.

Extended Integer Instructions (EIS)
ASH
ASHC
DIV
MUL

EIS is standard on:

- MICRO/PDP-11
- PDP-11/23 PLUS
- PDP-11/24
- PDP-11/44
- VAX-11 compatibility mode

EIS is also available as an option on the LSI-11/2.

MFPD, MFPI, MTPD, MTPI
Available on the MICRO/J-11, LSI-11/23, MICRO/PDP-11, PDP-11/23-PLUS, PDP-11/24, PDP-11/44, and VAX-11 compatibility mode.

MFPS, MTPS
Available on the MICRO/T-11, MICRO/J-11, LSI-11/2, FALCON SBC-11/21, LSI-11/23, MICRO/PDP-11, PDP-11/23-PLUS, and PDP-11/24.

MFPT
Available on the MICRO/T-11, MICRO/J-11, FALCON SBC-11/21, LSI-11/23, MICRO/PDP-11, PDP-11/23-PLUS, PDP-11/24, and PDP-11/44.

SPL
Available on MICRO/J-11 and PDP-11/44 only.

TSTSET, WRTLCK
Available on MICRO/J-11 only.

Table 5-1 PDP-11 Instruction Set

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

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				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

			16> Rv1 ← tmp<15:0> The double word R,Rv1 is shifted NN places to the right or left, where NN = (src) <5:0>	shift. C: loaded with high-order bit when left shift; loaded with low-order bit when right shift (loaded with the last bit shifted out of the 32-bit operand).	specified by the count field (bits <5:0> of the source operand). The registers are replaced by the result. First, bits <31:16> of the result are stored in register R. Then, bits <15:0> of the result are stored in register Rv1. The count ranges from −32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift, but condition codes are affected. Condition codes are always set on the 32-bit result. **Note:** 1) The sign bit of the register R is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C bit stores the last bit shifted out. 2) Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign of the register.
ASL ASLB Arithmetic	SO SO	0063DD 1063DD	(dst) ← (dst) shifted one place to the left	N: set if high-order bit of the result is set (result < 0)	Shifts all bits of the destination left one place. The low-order bit is loaded with a 0. The C bit of the

Table 5-1 PDP-11 Instruction Set, continued

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.

BCC Branch if Carry Clear	PC	103000 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is clear.
BCS Branch if Carry Set	PC	103400 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is set. Used to test for a carry in the result of a previous operation.
BEQ Branch if Equal (to zero)	PC	001400 PLUS 8-bit offset	PC ← PC + (2 × offset) if Z = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the Z bit and causes a branch if Z is set. For example, it is used to test equality following a CMP operation, and to test that no bits set in the destination were also set in the source following a BIT operation, and, generally, to test that the result of the previous operation was 0.
BGE Branch if Greater than or Equal	PC	002000 PLUS 8-bit offset	PC ← PC + (2 × offset) if $N \veebar V = 0$	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if N and V are either both clear or both set. BGE is the complementary operation to BLT. Thus, BGE always causes a branch when it follows an operation that caused addition of two positive numbers. BGE also causes a branch in a 0 result.

Table 5-1 PDP-11 Instruction Set, continued

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

					long as the source has a higher unsigned value than the destination.
BHIS Branch if Higher than or Same	PC	103000 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is cleared.
BIC BICB Bit Clear	DO	04SSDD 14SSDD	(dst) ← ~ (src) Λ (dst)	N: set if high-order bit of result set Z: set if result = 0 V: cleared C: unaffected	Clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are unaffected.
BIS BISB Bit Set	DO	05SSDD 15SSDD	(dst) ← (src)v(dst)	N: set if high-order bit of result set Z: set if result = 0 V: cleared C: unaffected	Performs inclusive OR operation between the source and destination operands and leaves the result at the destination address, i.e., corresponding bits set in the source are set in the destination. The contents of the destination are lost.
BIT BITB Bit Test	DO	03SSDD 13SSDD	(dst) Λ (src)	N: set if high-order bit of result set Z: set if result = 0 V: cleared	Performs logical AND comparison of the source and destination operands and modifies Condition Codes accordingly. Neither the

Table 5-1 PDP-11 Instruction Set, continued

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

					branch if the result of the previous operation was 0.
BLO Branch if Lower	PC	103400 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is set. Used to test for a carry in the result of a previous operation.
BLOS Branch if Lower or Same	PC	101400 PLUS 8-bit offset	PC ← PC + (2 × offset) if CvZ = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the previous operation caused either a carry or a zero result. BLOS is the complementary operation to BHI. The branch occurs in comparison operations as long as the source is equal to or has a lower unsigned value than the destination.
BLT Branch if Less Than	PC	002400 PLUS 8-bit offset	PC ← PC + (2 × offset) if $N \not\vee V = 1$	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the exclusive OR of the N and V bits is 1. Thus, BLT always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BLT always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Fur-

Table 5-1 PDP-11 Instruction Set, continued

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

					the result of the previous operation was not 0.
BPL Branch if Plus	PC	100000 PLUS 8-bit offset	PC ← PC + (2 × offset) if N = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the N bit and causes a branch if N is clear. BPL is the complementary operation of BMI.
BPT Breakpoint Trap	PC	000003	−(SP) ← PS −(SP) ← PC PC ← (14) PS ← (16)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	Performs a trap sequence with a trap vector address of 14. Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids. No information is transmitted in the low byte.
BR Branch (Unconditional)	PC	000400 PLUS 8-bit offset	PC ← PC + (2 × offset)	N: unaffected Z: unaffected V: unaffected C: unaffected	Provides a way of transferring program control within a range of −128 to +127 words with a one-word instruction. An unconditional branch.
BVC Branch if V bit Clear	PC	102000 PLUS 8-bit offset	PC ← PC + (2 × offset) if V = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the V bit and causes a branch if the V bit is clear. BVC is the complementary operation to BVS.

Table 5-1 PDP-11 Instruction Set, continued

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		

Code Bits					
CLC Clear C	CC	000241	C ← 0		
CLN Clear N	CC	000250	N ← 0		
CLV Clear V	CC	000242	V ← 0		
CLZ Clear Z	CC	000244	Z ← 0		
CMP CMPB Compare	DO	02SSDD 12SSDD	(src) – (dst) [in detail (src) + ~ (dst) + 1]	N: set if result < 0 Z: set if result = 0 V: set if there is arithmetic overflow; i.e., operands of opposite signs and the sign of the destination is the same as the sign of the result. C: set if there is a bor-	Compares the source and destination operands and sets the condition codes, which may then be used for arithmetic and logical conditional branches. Both operands are unaffected. The only action is to set the condition codes. The comparison is customarily followed by a conditional branch instruction. Note that unlike the subtract instruction, the order of

Table 5-1 PDP-11 Instruction Set, continued

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
	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 (SP) to the Supervisor Mode 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-

			temp <3:0> ← 0;
			PSW <13:12> ← PSW <15:14>;
			PSW <15:14> ← 01;
			PSW <4> ← 0;
			−(SP) ← temp;
			−(SP) ← PC;
			−(SP) ← (dst);
			PC ← (10);
			end;
			else trap to 10 in Kernel mode;
					erand), and sets the PC to the contents of location 10 (in Supervisor space). The called program in Supervisor space may return to the calling program by popping the argument word from the stack and executing RTI. On return, the Condition Codes are determined by the PSW word on the stack. Hence, the called program in Supervisor space may control the Condition Code values following return.

DEC DECB Decrement	SO	0053DD 1053DD	(dst) ← (dst) − 1	N: set if result < 0 Z: set if result = 0 V: set if (dst) was M.N.I. C: unaffected	Subtracts 1 from the contents of the destination.
DIV Divide	DO	071RSS	R,Rv1 ← R,Rv1/(src)	N: set if quotient < 0 (unspecified if V = 1) Z: set if quotient = 0 (unspecified if V = 1)	

Table 5-1 PDP-11 Instruction Set, continued

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.

					Caution: EMT is used frequently by DIGITAL system software and is therefore not recommended for general use.
HALT	MS	000000		N: unaffected Z: unaffected V: unaffected C: unaffected	Causes the processor operation to cease. The console is given control of the processor. The console data lights display the contents of the PC (which is the address of the HALT instruction plus 2). Transfers on the UNIBUS are terminated immediately. Pressing the continue key on the console causes processor operation to resume.
INC INCB Increment	SO	0052DD 1052DD	(dst) ← (dst) + 1	N: set if result < 0 Z: set if result = 0 V: set if (dst) was M.P.I. C: unaffected	Adds 1 to the contents of the destination.
IOT I/O Trap	PC	000004	−(SP) ← PS --(SP) ← PC PC ← (20) PS ← (22)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector	Performs a trap sequence with a trap vector address of 20. Used to call the I/O executive routine IOX in the paper tape software system and for error reporting in the disk operating system. No information

Table 5-1 PDP-11 Instruction Set, continued

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

					deferred mode is legal and will cause program control to be transferred to the address held in the specified register. **Note that instructions are word data and therefore must be fetched from an even numbered address. A boundary error trap condition will result when the processor attempts to fetch an instruction from an odd address.**
JSR Jump to Subroutine	PC	004RDD	(tmp) ← (dst) (tmp is an internal processor register) ↓(SP) ← reg (push reg contents onto processor stack) reg ← PC (PC holds the location following JSR; this address now put in reg) PC ← tmp (PC now points to	N: unaffected Z: unaffected V: unaffected C: unaffected	In execution of the JSR, the old contents of the specified register (the linkage pointer) are automatically pushed onto the R6 stack and new linkage information is placed in the register. Thus, subroutines nested within subroutines to any depth may all be called with the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include instructions in each routine to save and restore the linkage pointer. Further, since all linkages are

Table 5-1 PDP-11 Instruction Set, continued

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,

					the modified register contact is pushed on SP. For example, JSR R5,@(R5)+ will cause the modified value of R5 to be pushed to SP.
MARK	PC	0064NN	SP ← PC + 2 × NN PC ← R5 R5 ← (SP)+ NN = number of parameters	N: unaffected Z: unaffected V: unaffected C: unaffected	Used as part of the standard PDP-11 subroutine return convention. MARK facilitates the stack clean-up procedures involved in subroutine exit. Assembler format is: MARK N
MFPD Move From Previous Data space MFPI Move From Previous Instruction space	MS	1065SS 0065SS	tmp ← (src) −(SP) ← tmp	N: set if the source < 0 Z: set if the source = 0 V: cleared C: unaffected	Pushes a word onto the current R6 stack from an address in previous space determined by PS<13:12>. The source address is computed using the current registers and memory map. When MFPI is executed and both previous mode and current mode are User, the instruction functions as though it were MFPD.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
PDP-11/03, and PDP-11/04)					
MFPS Move Byte from PSW	MS	1067DD	(dst) ← PS<7:0> dst lower 8 bits	N: set if PS bit 7 = 1 Z: set if PS <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	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

			processor subcode		PDP-11/24). The high byte of R0 is loaded with a processor-specific subcode, (currently 0 on the PDP-11/24 and PDP-11/44). The condition codes are not affected. The previous contents of R0 are lost. **Note:** On processors where this instruction is not implemented, a reserved instruction trap through vector 10_8 is taken.
MOV MOVB Move	DO	01SSDD 11SSDD	(dst) ← (src)	N: set if (src) < 0 Z: set if (src) = 0 V: cleared C: unaffected	Moves the source operand to the destination location. The previous contents of the destination are lost. The source operand is not affected. Byte: Same as MOV. The MOVB to a register (unique among byte instructions) extends the most significant bit of the low-order byte (sign extension) into the high byte of the selected register. Otherwise, MOVB operates on bytes exactly as MOV operates on words.
MTPD Move To	MS	1066DD 0066DD	tmp ← SP+ (dst) ← tmp	N: set if the source < 0 Z: set if the source = 0	This instruction pops a word off the current R6 stack determined by PS

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Previous Data space				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.
MTPI Move To Previous Instruction space					
MTPS Move Byte To PSW	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

				C: same as above	set with this instruction. The src operand remains unchanged.
MUL Multipy	DO	070RSS	R,Rv1 ← R × (src)	N: set if product < 0 Z: set if product = 0 V: cleared C: set if the result is less than -2^{15} or greater than or equal to 2^{15}. Condition codes set on 32-bit result even if R is odd.	The contents of the destination register and source taken as 2's complement integers are multiplied and stored in the destination register and the succeeding register (if R is even). If R is odd, only the low-order product is stored. Assembler syntax is: MUL S,R. (Note that the actual destination is R, Rv1, which reduces to just R when R is odd.)
NEG NEGB Negate	SO	0054DD 1054DD	(dst) ← −(dst)	N: set if result < 0 Z: set if result = 0 V: set if result = M.N.I. C: cleared if result = 0; set otherwise	Replaces the contents of the destination address by its 2's complement. Note that 100000 is replaced by itself.
NOP No Operation	CC	000240 000260	None	N: unaffected Z: unaffected V: unaffected	No operation is performed.

Table 5-1 PDP-11 Instruction Set, continued

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.

ROR RORB Rotate Right	SO	0060DD 1060DD	(dst) ← (dst) rotate right one place	N: set if high-order bit of the result is set Z: set if all bits of result are 0 V: loaded with the exclusive OR of the N bit and the C bit as set by ROR. C: set if the low-order bit of the destination was set prior to instruction execution.	Rotates all bits of the destination right one place. The low-order bit is loaded into the C bit and the previous contents of the C bit are loaded into the high-order bit of the destination.
RTI Return from Interrupt	MS	000002	PC ← (SP)+ PS ← (SP)+	N: loaded from current R6 stack Z: loaded from current R6 stack V: loaded from current R6 stack C: loaded from current R6 stack	Used to exit from an interrupt or trap service routine. The PC and PS are restored (popped) from the R6 stack. If the RTI sets the T bit in the PS, a trace trap will occur prior to executing the next instruction. When executed in Supervisor Mode, the current and previous

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					mode bits in the restored PS cannot be Kernel. When executed in User mode, the current and previous mode bits in the restored PS can only be User. RTI cannot clear PS <11> if it was already set. When executed in user or supervisor mode, PS <7:5> are unaffected.
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 trace trap, while RTI permits a trace trap. If a trace trap is pending, the first instruction after the RTT will be executed prior to the next "T" trap. In the case of the RTI instruction, the "T" trap will occur immediately after the RTI. When executed in Supervisor Mode, the current and previous mode bits in the restored PS cannot be Kernel. When executed in User Mode, the current and previous mode bits in the restored PS can only be User.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					RTT cannot clear PS<11> if it was already set. When executed in user or supervisor mode, PS <7:5> are unaffected.
SBC SBCB Subtract Carry	SO	0056DD 1056DD	(dst) ← (dst) − C	N: set if result < 0 Z: set if result = 0 V: set if (dst) =M.N.I. C: set if (dst) was 0 and C was 1 prior to instruction execution.	Subtracts the contents of the C bit from the destination.
S Set Selected Condition Codes	CC	000260 PLUS 4-bit mask	Set condition code bits. Selectable combinations of these bits may be set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified; sets the bit specified by bit 0, 1, 2, or 3. Bit 4 is a 1. **Operation:** PSW <3:0> ← PSW <3:0> v mask <3:0>		
SCC	CC	000277	N, Z, V, C ← 1		

Set all Condition Codes					
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 5-1 PDP-11 Instruction Set, continued

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

SUB Subtract	DO	16SSDD	(dst) ← (dst) − (src) {in detail (dst) ← (dst)+ ~ (src)+1	N: set if result < 0 Z: set if result = 0 V: set if there is arithmetic overflow as a result of the operation, i.e., if the operands were of opposite signs and the sign of the source is the same as the sign of the result. C: set if there is a borrow into the most significant bit of the result, i.e., if (dst) + ~ (src)+1 was less than 2^{16}.	Subtracts the source operand from the destination operand and leaves the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. In double precision arithmetic, the C bit, when set, indicates a borrow.

Table 5-1 PDP-11 Instruction Set, continued

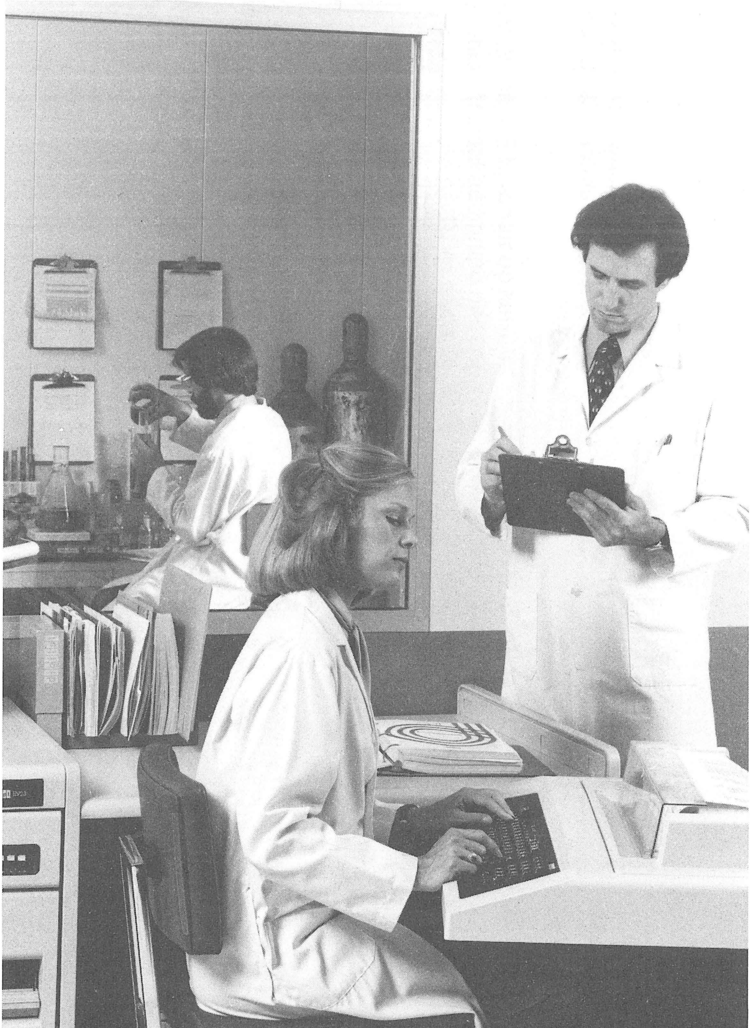
Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
SWAB Swap Bytes	SO	0003DD	tmp ← (dst) <7:0> (dst) <7:0> ← (dst) <15:8> (dst) <15:8> ← tmp	N: set if high-order bit of low-order byte (bit 7) of result is set Z: set if low-order byte of result = 0 V: cleared C: cleared	Exchanges high-order byte and low-order byte of the destination word (destination must be a word address).
SXT Sign Extend	SO	0067DD	(dst) ← 0 if N bit is clear (dst) ← −1 if N bit is set	N: unaffected Z: set if N bit clear V: cleared C: unaffected	If the condition code bit N is set, then a −1 is placed in the destination operand; if the N bit is clear, then a 0 is placed in the destination operand. This instruction is particularly useful in multiple precision arithmetic because it permits the sign to be extended through multiple words.

TRAP	PC	104400 to 104777	−(SP) ← PS −(SP) ← PC PC ← (34) PS ← (36)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34. **Note:** Since DIGITAL software makes frequent use of EMT, the TRAP instruction is recommended for general use.
TST TSTB Test	SO	0057DD 1057DD	tmp ← (dst)	N: set if result < 0 Z: set if result = 0 V: cleared C: cleared	Sets the condition codes N and Z according to the contents of the destination address.
TSTSET Test Destination and Set Low Bit.	SO	0072DD	(R0)←(dst)	N:set if R0 < 0 Z:set if R0 = 0 V:clear C:gets contents of bit 0	Reads/Locks destination word and stores it in R0. Writes/Unlocks (R0)v1 into destination. If mode is 0, traps to 10.

Table 5-1 PDP-11 Instruction Set, continued

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

					an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT.
WRTLCK Read/Lock Destination. Write/Un-Lock R0 into destination.	SO	0073DD	(dst)←(R0)	N:set if R0 < 0 Z:set if R0 = 0 V:clear C:unchanged	Writes contents of R0 into destination using bus lock. If mode is 0, traps to 10.
XOR Exclusive OR	DO	074RDD	(dst) ← R∀ (dst)	N: set if the result < 0 Z: set if result = 0 V: cleared C: unaffected	The exclusive OR of the register and destination operand is stored in the destination address. Contents of register are unaffected. Assembler format is XOR R,D.



CHAPTER 6

FLOATING-POINT INSTRUCTION SET FP-11

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-11/2. Please refer to Appendix C for a description of FIS.
2. The FP11 instruction set supports both single-precision and double-precision floating-point arithmetic. It is available as a microcode option, KEF11-AA, for the LSI-11/23, MICRO/PDP-11, PDP-11/23-PLUS and PDP-11/24. It is also available as a faster, hardware option on the MICRO/PDP-11, PDP-11/23-PLUS, PDP-11/24 (FPF-11), and PDP-11/44 (FP11-F). The microcoded FP11 instruction set is standard on the J-11 chipset. 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-precision 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 6-1.)

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.

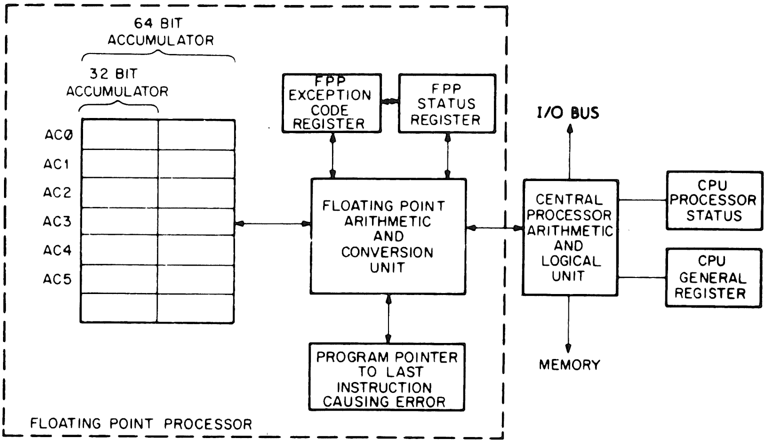

64 BIT ACCUMULATOR
32 BIT ACCUMULATOR
ACØ
AC1
AC2
AC3
AC4
AC5
FPP EXCEPTION CODE REGISTER
FPP STATUS REGISTER
FLOATING POINT ARITHMETIC AND CONVERSION UNIT
PROGRAM POINTER TO LAST INSTRUCTION CAUSING ERROR
FLOATING POINT PROCESSOR
I/O BUS
CENTRAL PROCESSOR ARITHMETIC AND LOGICAL UNIT
MEMORY
CPU PROCESSOR STATUS
CPU GENERAL REGISTER

Figure 6-1 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.

NOTE

> For implementation details on the various FP11 options, see the Microcomputers and Memories Handbook or the PDP-11 Systems Handbook.

Floating-Point Data Formats

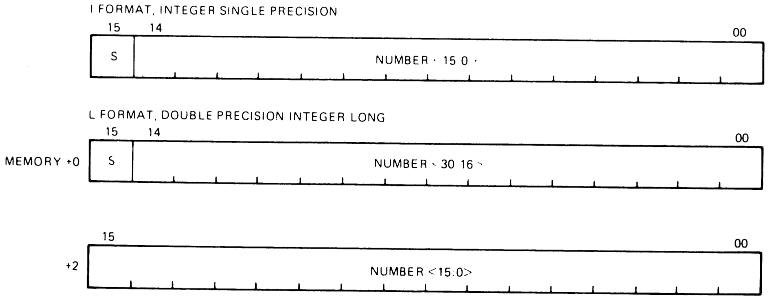
Please refer to Chapter 3—Data Representation—for information on floating-point data formats.

FLOATING-POINT STATUS REGISTER (FPS)

This register provides mode and interrupt control information for the floating-point unit and indicates conditions resulting from the execution of the previous instruction. It may be loaded via the LDFPS instruction; it may be read via the STFPS instruction (both of these instructions are included in the Floating-Point Instructions section of this chapter). The floating-point status register is illustrated in Figure 6-3.

For the purposes of discussion a set bit = 1 and a clear bit = 0. Four bits of the FPS register control the modes of operation:

- Single/Double: floating-point numbers can be either single-precision 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 and FP11-E.


I FORMAT, INTEGER SINGLE PRECISION
15
14
00
S
NUMBER <15:0>
L FORMAT, DOUBLE PRECISION INTEGER LONG
15
14
00
MEMORY +0
S
NUMBER <30:16>
15
00
+2
NUMBER <15:0>

WHERE S = SIGN OF NUMBER

NUMBER = 15 BITS IN I FORMAT, 31 BITS IN L FORMAT

Figure 6-2 Two's Complement Format

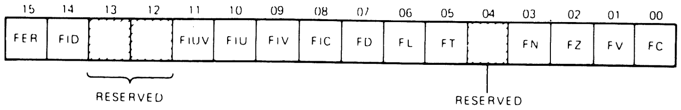


Figure 6-3 Floating-Point Status Register

The FPS register contains an error flag and four condition codes: carry, overflow, zero, and negative, which are similar 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, FP11-E 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 that 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 and FP11-E, FMM bit) can 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 occurs and the corresponding interrupt is enabled

This action is independent of the FID bit status.

Also note that the FPP never clears 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 **Name:**
Function: Reserved for future DIGITAL use.

Bit: 12 **Name:**
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 on the floating-point option, except on NEG, ABS, and TST1, for which it occurs after execution. When FIUV is clear, " – 0" can be loaded and used in any floating-point option operation. Note that the interrupt is not activated by the presence of " – 0" in a floating-point accumulator; 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 zero, which is correct. An exception is discussed later in the detailed description of the LDEXP instruction.

If the FIU bit is clear 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 clear 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 clear, 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 used; when clear, single-precision is used.

Bit: 6 **Name:** Floating Long Integer Mode (FL)
Function: The FL bit is used in conversion between integer and floating-point format. When set, the integer format assumed is double-precision two's complement (i.e., 32 bits). When clear, the integer format is assumed to be single-precision two'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 clear, the result is rounded.

Bit: 4 **Name:** Floating Maintenance Mode (FMM)
Function: FP11-C and FP11-E 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 clear.

Bit: 2 **Name:** Floating Zero (FZ)
Function: FZ is set if the result of the last floating-point operation was zero—including a zero stored as the result of underflow or overflow—otherwise it is clear.

Bit: 1 **Name:** Floating Overflow (FV)
Function: FV is set if the last floating-point operation resulted in an exponent overflow; otherwise it is clear.

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-to-integer 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-to-integer or double-to-integer conversion error
8	Floating overflow
10	Floating underflow
12	Floating undefined variable
14	Maintenance trap (FP11-C and FP11-E 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, the FEC and FEA registers are read-only; no instructions exist to write into these 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 10_8 for D format.

In mode 0, the user can make use of all six floating-point accumulators (AC0-AC5) as source or destination. Specifying floating-point option 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 floating-point 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 contains 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 zero as an exact zero (unless FIUV is enabled and the operand is " – 0", in which case an interrupt occurs). For all arithmetic operations except DIV, a zero operand implies that the instruction is exact. The same holds for DIV if the zero operand is the dividend. But if the divisor is zero, 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. Some processors have a slightly larger error bound; see Appendix B for details.

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, the rounded value equals the chopped value which equals the 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 zero is a reasonable resolution of the problem for many applications. This is done by the floating-point option if the underflow interrupt is disabled. The error incurred by this action is absolute rather than relative; 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 examination of 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:

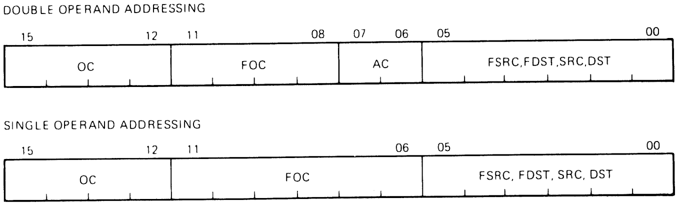
- On overflow, it is too small by 400_8.
- On underflow, if the biased exponent is zero, it is correct. If it is not zero, it is too large by 400_8.

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-precision 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. In our notation, FSRC and FDST use floating-point addressing modes; SRC and DST use CPU addressing modes. Figure 6-3 illustrates single-floating-point and double-floating-point operand addressing.

In the descriptions of the floating-point instructions, all implementations operate identicallly, except where explicitly stated otherwise. Table 5-1 describes the floating-point conventions used in the PDP-11 instruction set.



OC OPCODE 17
FOC FLOATING OPCODE
AC FLOATING POINT ACCUMULATOR (AC0 AC3)
FSRC AND FDST USE FPP ADDRESSING MODES
SPC AND DST USE CPU ADDRESSING MODES

Figure 6-4 Single-Operand and Double-Operand Addressing

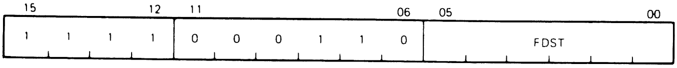
Table 6-1 Floating-Point Conventions

Symbolic	**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

Symbolic	**Description**
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

Floating-Point Instructions

ABSF
ABSD
Take Absolute Value 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.

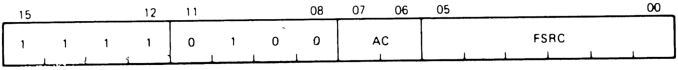
Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: If a " – 0" is present in memory and the FIUV bit is enabled, then an exact zero is stored in memory. The condition codes reflect an exact zero (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-precision 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:

- Overflow with interrupt disabled
- Underflow with interrupt disabled.

For these exceptional cases, an exact zero 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:

- One LSB in truncated mode with either single-precision or double-precision.
- From ½ LSB to ¾ LSB in rounding mode, depending on floating-point option. See Appendix B—PDP-11 Family Differences—for details.

Special Comment: The undefined variable " – 0" can occur only in conjunction with overflow or underflow. It will be stored in AC, only if the corresporflow, 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:

- One LSB in truncated mode with either single-precision or double-precision.
- From ½ LSB to ¾ LSB in rounding mode depending on floating-point option. See Appendix B—PDP-11 Family Differences—for details.

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

15			12	11											00
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Format: CFCC

Operation:
C◄— FC
V◄— FV
Z◄— FZ
N◄— FN

Description: Copy the floating-point condition codes into the CPU's condition codes.

CLRF
CLRD
Clear Floating/Double 1704 FDST

15			12	11					06	05					00
1	1	1	1	0	0	0	1	0	0	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.

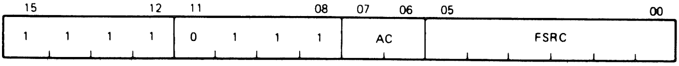
Interrupts: No interrupts will occur.

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.

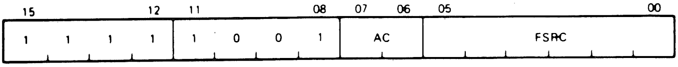
Interrupts: If FIUV is enabled, trap on " – 0" in (fsrc) occurs before execution.

Accuracy: These instructions are exact.

Special Comment: An operand which has a biased exponent of 0 is treated as if it were an exact zero. In this case, where both operands are zero, the FPP will store an exact zero 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 zero, it is treated as an exact zero. For fsrc this would imply division by zero; in this case the instruction is aborted, the FEC register is set to four 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:

- Overflow with interrupt disabled
- Underflow with interrupt disabled

For these exceptional cases, an exact zero 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 one 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

15			12	11			08	07	06	05					00
1	1	1	1	1	1	1	1	AC		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 one LSB in chopping mode and by LSB in rounding mode.

LDCIF LDCLF
LDCID LDCLD

Load and Convert Integer or Long Integer
to Floating or Double-Precision 177(AC)SRC

15			12	11			08	07	06	05					00
1	1	1	1	1	1	1	0	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 are 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 one LSB in chopping mode and by ½ LSB in rounding mode.

LDEXP
Load Exponent

176(AC + 4)SRC

15			12	11			08	07 06	05 00
1	1	1	1	1	1	0	1	AC	SRC

Format: LDEXP SRC,AR

Operation: If $-200_8 < \text{(src)} < 200_8$, EXP[AC] ← SRC $+200_8$ and the rest of AC is unchanged.

If (src) $> 177_8$ and FIV is enabled, EXP[AC] ← $[\text{(src)} + 200_8]<7:0>$ on the FP11-A, -F and KEF11-AA. See Appendix B—PDP-11 Family Differences—for the FP11-C.

If (src) $> 177_8$ and FIV is disabled, AC ← exact 0.

If (src) $< -177_8$ and FIU is enabled, EXP[AC] ← $[\text{(src)} + 200_8] <7:0>$ on the FP11-A, -F and KEF11-AA. See Appendix B—PDP-11 Family Differences—for 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 equals (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)] $\leq 177_8$.

If (src) $> 177_8$, the result is treated as overflow. If (src) $< -177_8$, the result is treated as underflow. See Appendix B—PDP-11 Family Differences—for treatment of abnormal conditions by the FP-11C and FP-11B.

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) $\neq -200$, AC changes from a floating-point number treated as zero 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

15			12	11			08	07 06	05 00
1	1	1	1	0	1	0	1	AC	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-precision 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

15			12	11					06	05 00
1	1	1	1	0	0	0	0	0	1	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

15			12	11			08	07	06	05					00
1	1	1	1	0	0	1	1	AC		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
$\frac{1}{2} \leq f < 1$ 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 $0 \leq g < 1$

Both N and g have the same sign as PROD. They are returned as follows:

If AC is an even-numbered accumulator (0 or 2), N is stored in AC + 1 (1 or 3), and g is stored in AC.

If AC is an odd-numbered accumulator, N is not stored, and g is stored in AC.

The two statements above can be combined as follows:

N is returned to ACv1 and g is returned to AC, where v means OR.

Five special cases occur, as indicated in the following formal description with L = 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 one 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 one 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.

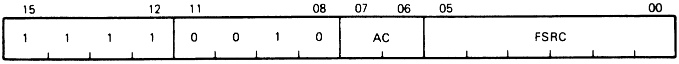
ARG * 2/PI = N + g. The low two bits of N identify the quadrant, and g is the argument reduced to the first quadrant. The accuracy of N + g is limited to L bits because of the factor 2/PI. The accuracy of the reduced argument thus depends on the size of N.

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 two, 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 two). 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 zero, (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 zero 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 zero, which is correct.

Accuracy:

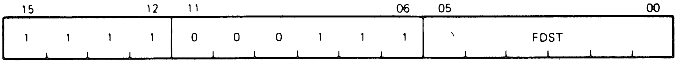
Errors due to overflow and underflow are described above. If neither occurs, the error incurred is bounded by one LSB in chopping mode and ½ LSB in rounding mode.

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-precision 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 floating-point processor stores an exact zero in memory. If a negative number is present, then the floating-point processor stores the actual negative result in memory. The condition codes reflect an exact zero (FZ ← 1).

SETF
Set Floating Mode 170001

15			12	11											00
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1

Format: SETF

Operation: FD ← 0

Description: Set the floating-point option in single-precision mode.

SETD
Set Floating Double Mode 170011

15			12	11											00
1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	1

Format: SETD

Operation: FD ◄— 1

Description: Set the floating-point option in double-precision mode.

SETI
Set Integer Mode 177002

15			12	11											00
1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0

Format: SETI

Operation: FL ◄— 0

Description: Set the floating-point option for short-integer data.

SETL
Set Long-Integer Mode 177012

15			12	11											00
1	1	1	1	0	0	0	0	0	0	0	0	1	0	1	0

Format: SETL

Operation: FL ◄— 1

Description: Set the floating-point option for long-integer data.

STCFD
STCDF
Store and Convert from Floating to Double and from Double to Floating 176(AC)FDST

15			12	11			08	07	06	05					00
1	1	1	1	1	1	0	0	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 then 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

15			12	11			08	07	06	05					00
1	1	1	1	1	0	0	0	AC		FDST					

Format: STF AC,FDST

Operation: (fdst) ← AC

Condition Codes:
FC ← FC
FV ← FV
FZ ← FZ
FN ← FN

Description: Store single-precision 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.

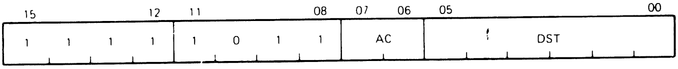
Overflow and underflow cannot occur.

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 floating-point processor. 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
$2^{32}-1$ 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.

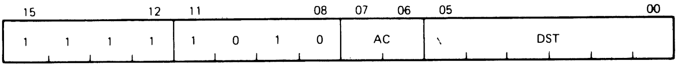
Numbers to be converted are always chopped (rather than rounded) before conversion. This is true even when the chop mode bit FT is cleared in the FPS register.

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 always exact.

STFPS
Store Floating-point Processor's Program Status 1702 DST

15			12	11					06	05					00
1	1	1	1	0	0	0	0	1	0	DST					

Format: STFPS DST

Operation: (dst) ← FPS

Description: Store floating-point status register in dst.

Special Comment: Bits 13, 12, and 4 (if maintenance mode is not implemented) are stored as 0. All other bits are the corresponding bits in the FPS.

STST
Store Floating-point Processor's Status 1703 DST

15			12	11					06	05					00
1	1	1	1	0	0	0	0	1	1	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

15			12	11			08	07	06	05	00
1	1	1	1	0	1	1	0	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-precision 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 zero 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 zero, 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 zero or one, 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:

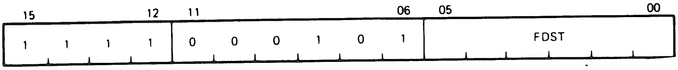
1. 1 LSB in truncated mode with either single-precision or double-precision
2. From ½ LSB to ¾ LSB in rounding mode depending on floating-point processor. See Appendix B—PDP-11 Family Differences—for details.

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 7
COMMERCIAL INSTRUCTION SET

The PDP-11 Commercial Instruction Set (CIS11) option is supported by the following processors:

- MICRO/PDP-11
- PDP-11/23 PLUS
- PDP-11/24
- PDP-11/44

The CIS11 consists of the following extended instruction groups:

07602X	Commercial Load 2 Descriptors
07603X	Character String Move
07604X	Character String Search
07605X	Numeric String
07606X	Commercial Load 3 Descriptors
07607X	Packed String
07613X	Character String Move (in-line)
07614X	Character String Search (in-line)
07615X	Numeric String (in-line)
07617X	Packed String (in-line)

These include instructions which operate on character strings and on decimal numbers. Each generic type of instruction is provided in two forms. The essential difference between the two forms is the manner in which operands are delivered to the instruction. The first form is the "register" form, where operands are implicitly obtained from the general registers. The second form is the "in-line" form, where operands or word address pointers to operands follow the opcode word in the instruction stream. The mnemonic for the in-line form is the mnemonic for the register form suffixed with the letter "I." The condition codes are set identically for both forms. The in-line forms minimize register modification.

Instructions are also provided which efficiently load operands into the general registers.

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

For a discussion of character data types—characters, character strings, and character sets—refer to Chapter 3.

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 corresponding pair of unequal characters. For the other search instructions, they show whether or not the operand strings were completely examined.

The condition codes for some character string search instructions may be interpreted according to the notion of success or failure. 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

For a discussion of decimal string data types—numeric strings and packed strings—refer to Chapter 3.

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

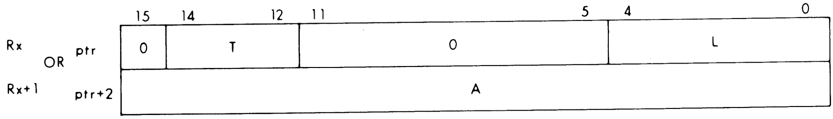
length <4:0> Number of digits specified as an unsigned binary integer

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

Decimal String Instructions

The decimal string instruction groups aid manipulation of decimal data. Several numeric (byte) and packed decimal data types are supported. Instructions are provided for basic arithmetic operations, as well as for compare, shift, and convert functions.

Instructions

Each arithmetic, shift and compare instruction operates on a single class of data type. Both numeric and packed string instructions are provided for most operations. Convert instructions have a source operand of one data type and a destination operand of another data type. Decimal string instructions specify to which class each of their decimal string operands belong. The data type supplied as part of each operand's descriptor may be any valid data type of the class. This permits a general mixing of data types within numeric and packed classes.

The data types on which an instruction operates are designated by the last letter(s) of the opcode mnemonic. "N" denotes numeric strings, "P" denotes packed strings, and "L" denotes long binary integers.

The arithmetic instructions are ADDN(I), ADDP(I), SUBN(I), SUBP(I), MULP(I) and DIVP(I). ASHN(I) and ASHP(I) shift a decimal string by a specified number of digit positions (either direction) with optional rounding, and store the result in the destination string. Thus, they effectively multiply or divide by a power of ten. If the shift count is zero, these shift instructions can be used simply to move decimal strings (destinations are stored with preferred representation). Move negated may be accomplished by using SUBN(I) or SUBP(I). Arithmetic comparison instructions, CMPN(I) and CMPP(I), are provided to examine the relative difference between two decimal strings.

CVTNL(I) and CVTPL(I) convert a decimal string to a long (32-bit) 2's complement integer. CVTLN(I) and CVTLP(I) convert a long integer to a decimal string. CVTNP(I) and CVTPN(I) convert between numeric and packed decimal strings.

The instructions are:

Numeric String Instructions

ADDN(I)	Add numeric
SUBN(I)	Subtract numeric
ASHN(I)	Arithmetic shift numeric
CMPN(I)	Compare numeric

Packed String Instructions

ADDP(I)	Add packed
SUBP(I)	Subtract packed
MULP(I)	Multiply packed
DIVP(I)	Divide packed
ASHP(I)	Arithmetic shift packed
CMPP(I)	Compare packed

Convert Instructions

CVTNL	Convert numeric to long
CVTLN	Convert long to numeric
CVTPL	Convert packed to long
CVTLP	Convert long to packed
CVTNP	Convert numeric to packed
CVTPN	Convert packed to numeric

Condition Codes

For instructions which store a value in a destination string, the N and Z bits reflect the value stored. The N bit indicates a negative destination; the Z bit indicates a destination having zero magnitude. A destination string with zero magnitude is considered to be positive (even if a negative zero was stored as a consequence of decimal overflow). Thus, the setting of N and Z are mutually exclusive.

The V bit will indicate whether the destination string accurately represents the result of the instruction. It is also set if division by zero was attempted. If the V bit is set, the destination string will represent the least significant portion of the result (truncated). If the V bit is cleared, the destination represents the true result.

For DIVP(I), C indicates division by zero. Otherwise, C is always cleared.

For comparisions using the CMPN(I) and CMPP(I) instructions, the N and Z bits reflect the signed relationship between the source strings. The signed branch instructions can test the result. V and C are cleared.

For instructions which return a long integer value, N reflects the sign of the 2's complement integer, and Z indicates whether it was zero. V indicates whether the long integer could not contain all significant digits and sign of the result. CVTNL(I) and CVTPL(I) also use C to represent a borrow from a more significant portion of the long binary result. Otherwise, C is cleared.

Operand Delivery

The "register form" of decimal string instructions implicitly 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 ≤ r ≤ 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 suspended, PS<8> will be set, R7 will be backed up to address the opcode word, and the suspension-event will be serviced. When the instruction is resumed, PS<8> indicates that execution of the instruction has previously begun.

In order to make these instructions suspendable on all processors, the instruction state is part of the user state which is saved by interrupt handling routines. This includes the general registers, condition codes and memory. This state is processor dependent when suspended. Software should not attempt to interpret or modify this state; it must only be saved and restored. Up to 64_{10} words of internal instruction state may also have been pushed onto the stack. This state must not be modified by software. The instruction will remove this state from the stack when it is resumed.

If PS<8> is set prior to executing a potentially suspendable instruction, the effect of the instruction is UNPREDICTABLE.

At the normal completion of an potentially suspendable instruction, PS<8> will be cleared.

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

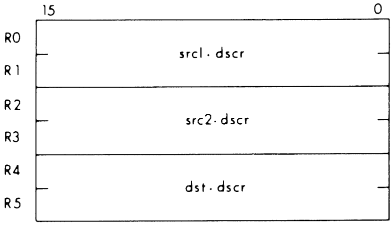
C: cleared

Opcodes:

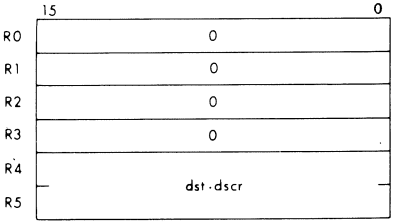
ADDN	076050
ADDP	076070
ADDNI	076150
ADDPI	076170

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.

15 — 12	11 — 8	7 — 0
0	rnd.dgt	shift.cnt

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.

	15 — 0
R0 R1	src.dscr
R2 R3	dst.dscr
R4	shift.dscr

When the instruction is completed, the source descriptor registers and shift descriptor register are cleared.

	15 — 0
R0	0
R1	0
R2 R3	dst.dscr
R4	0

In-line Form—ASHNI and ASHPI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word decimal string source descriptor, a word address pointer to a two-word decimal string destination descriptor, and a shift descriptor word. R0-R6 are unchanged when the instruction is completed.

Notes:

1. If bits <15:12> of the shift descriptor word are not zero, the effect of the instruction is unpredictable.
2. If bits <11:8> of the shift descriptor are not a valid decimal digit, the results of the instruction are unpredictable.
3. Any overlap of the source and destination strings will produce unpredictable results.

CMPC/CMPCI

Purpose: Compare Character

Operation: Src1 is compared with src2 (src1-src2)

Condition Codes: The condition codes are based on the arithmetic comparison of the most significant pair of unequal src1 and src2 characters (src1.byte-src2.byte)

N: set if result < 0; cleared otherwise

Z: set if result = 0: cleared otherwise

V: set if there was arithmetic overflow, that is, src1.byte<7> and src2.byte<7> were different, and src2.byte<7> was the same as bit <7> of (src1.byte-src2.byte); cleared otherwise

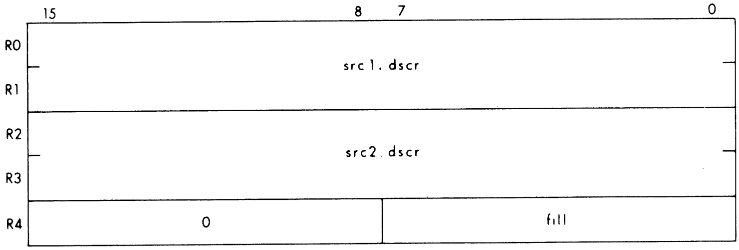
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Opcodes:		
	CMPC	076044
	CMPCI	076144

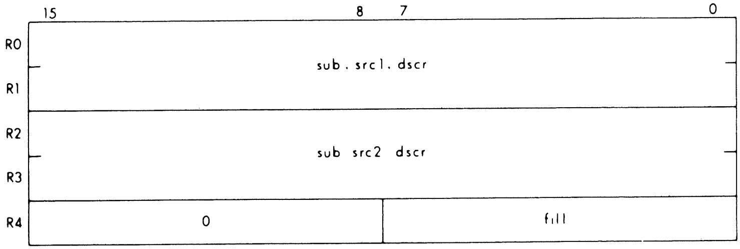
Description: Each character of src1 is compared with the corresponding character of src2 by examining the character strings from most significant to least significant characters. If the character strings are of unequal length, the shorter character string is conceptually extended to the length of the longer character string with fill characters beyond its least significant character. The instruction terminates when the first corresponding unequal characters are found or when both character strings are exhausted. The condition codes reflect the last comparison, permitting the unsigned branch instructions to test the result.

Register Form—CMPC

When the instruction starts, the operands must have been placed in the general registers. The first source character string descriptor is placed in R0-R1, the second source character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero.



The instruction terminates with 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.

	15 … 0
R0	scr1 . dscr
R1	
R2	src2 . dscr
R3	

When the instruction is completed, the source descriptor registers are cleared.

	15 … 0
R0	0
R1	0
R2	0
R3	0

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

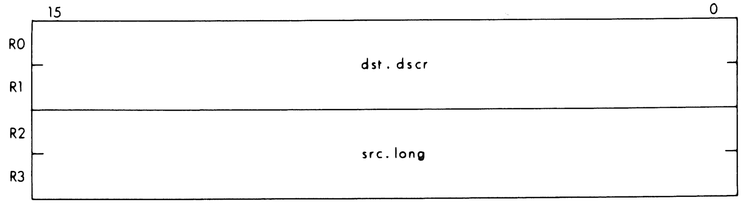
C: cleared

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.

	15 ... 0
R0 R1	dst.dscr
R2	0
R3	0

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.

	15	0
R0	scr.dscr	
R1		

When the instruction is completed, the source decimal string descriptors are cleared, and the destination long integer is returned in R2-R3.

	15	0
R0	0	
R1	0	
R2	dst.long	
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

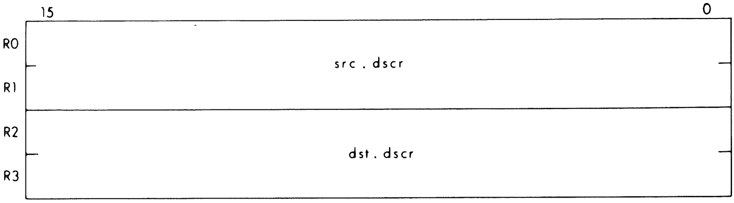
Opcodes:

CVTNP	076055
CVTPN	076054
CVTNPI	076155
CVTPNI	076154

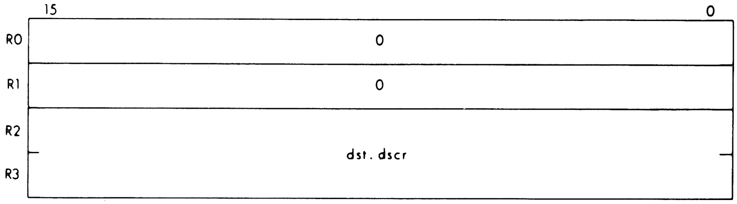
Description: These instructions convert between numeric and packed decimal strings. The source decimal string is converted and moved to the destination string. The condition codes reflect the result of the operation, and whether all significant digits were stored.

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.

	15	0
R0	src1. dscr	
R1		
R2	src2. dscr	
R3		
R4	dst. dscr	
R5		

When the instruction is completed, the source descriptor registers are cleared.

	15	0
R0	0	
R1	0	
R2	0	
R3	0	
R4	dst. dscr	
R5		

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

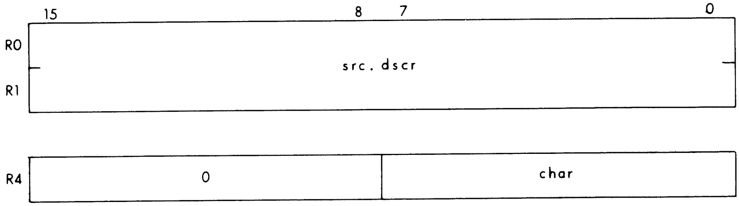
V: cleared

C: cleared

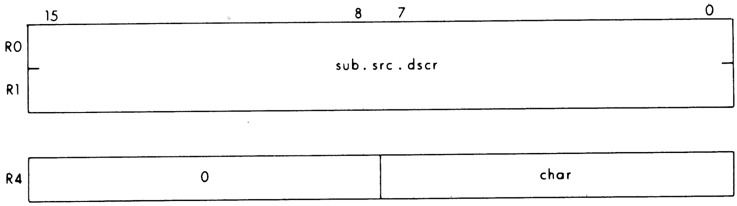
Opcodes:		
	LOCC	076040
	LOCCI	076140

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.

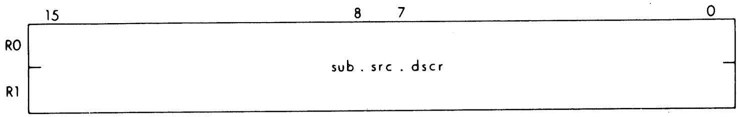


When the instruction is completed, R0-R1 contain a character set descriptor which represents the substring of the source character string beginning with the located character.



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

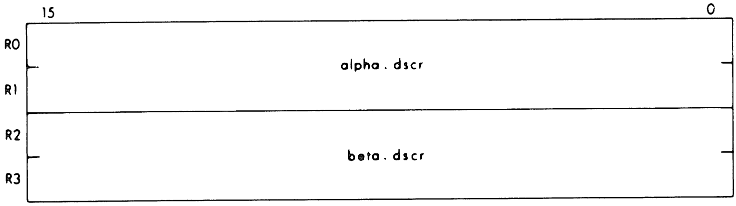
Condition Codes:
N: not affected
Z: not affected
V: not affected
C: not affected

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

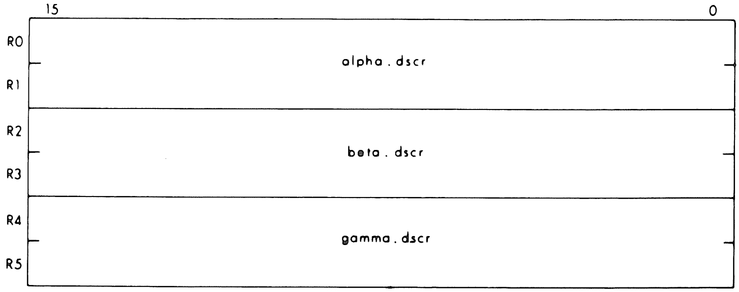
Condition Codes:
N: not affected
Z: not affected
V: not affected
C: not affected

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

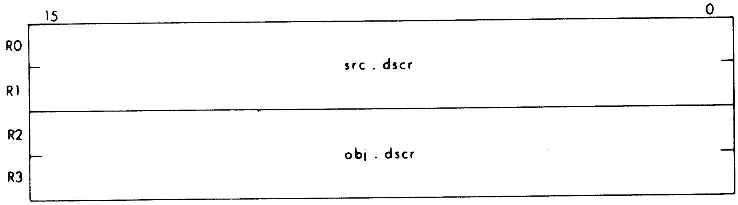
Opcodes:

MATC	076045
MATCI	076145

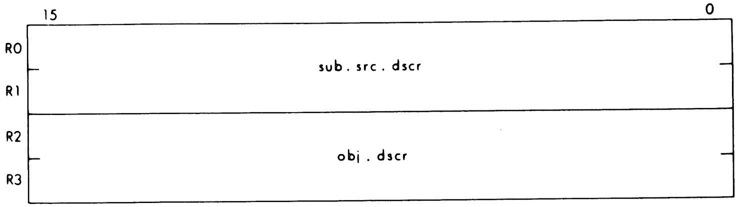
Description: The source character string is searched from most significant to least significant character for the first occurrence of the entire object character string. A character string descriptor is returned in R0-R1 which represents the portion of the original source character string from the most 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.

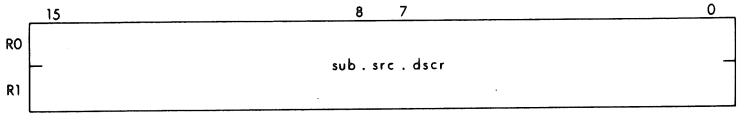


The instruction terminates with a character substring descriptor returned in R0-R1 which represents the portion of the original source character string beginning with the most significant character to competely match the object character string.



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.

	15 8	7 0
R0 R1	src dscr	
R2 R3	dst dscr	
R4	0	fill

When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared.

	15 8	7 0
R0	max(0,src len-dst len)	
R1	0	
R2	0	
R3	0	
R4	0	fill

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.

	15 8	7 0
R0	src dscr	
R1		
R2	dst. dscr	
R3		
R4	0	fill

When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared.

	15 8	7 0
R0	max(0, src. len - dst. len)	
R1	0	
R2	0	
R3	0	
R4	0	fill

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

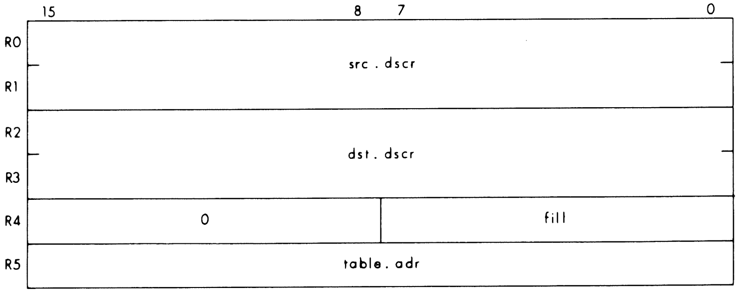
C: cleared if there was a carry from the most significant bit of the result; set otherwise

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

	15 8	7 0
R0	max(0, src. len - dst. len)	
R1	0	
R2	0	
R3	0	
R4	0	fill
R5	table. adr	

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

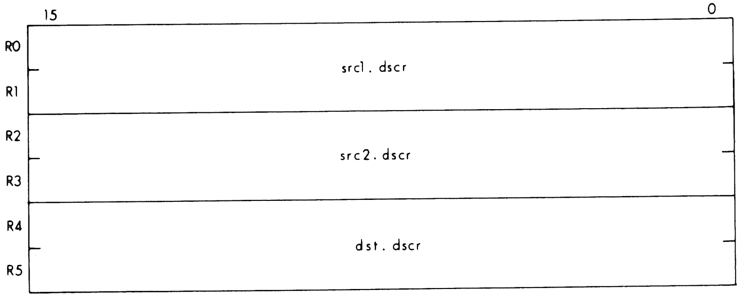
Opcodes:

MULP	076074
MULPI	076174

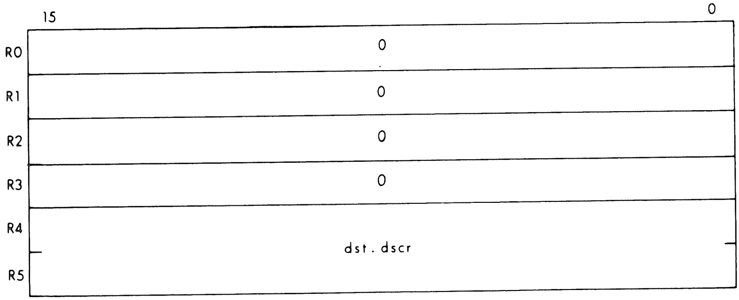
Description: Src1 and src2 are multiplied, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

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

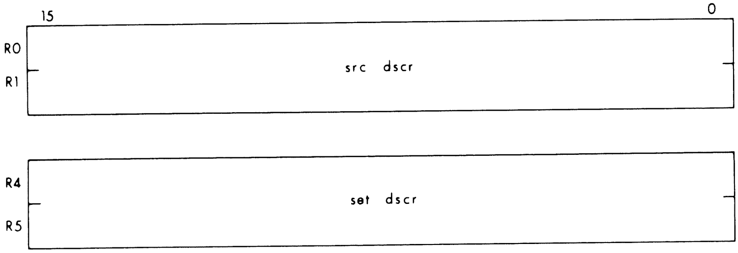
Opcodes:

SCANC	076042
SCANCI	076142

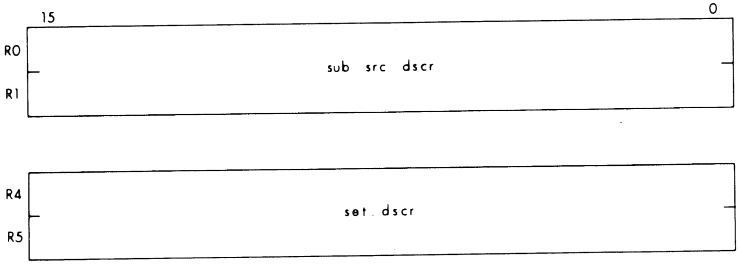
Description: The source character string is searched from most significant to least significant character until the first occurrence of a character which is a member of the character set. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning 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.

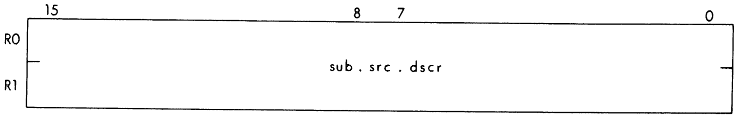


When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is a member of the character set.



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

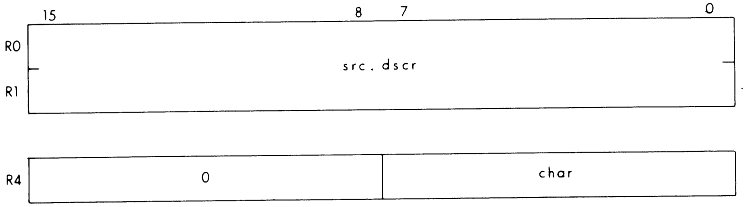
V: cleared

C: cleared

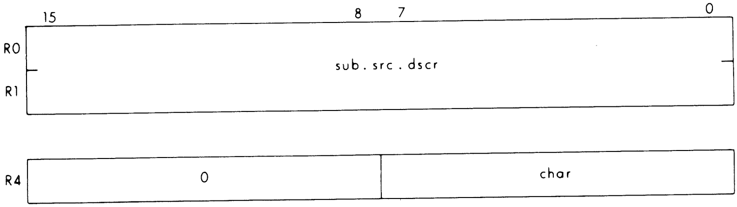
Opcodes:	SKPC	076041
	SKPCI	076141

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.



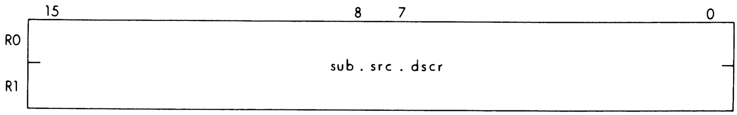
When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which was not equal to the search character.



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

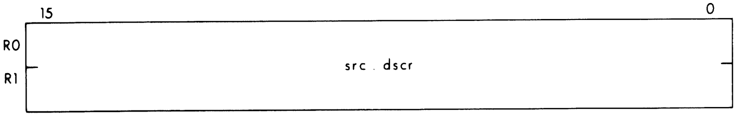
Opcodes:

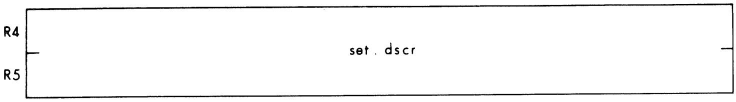
SPANC	076043
SPANCI	076143

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.

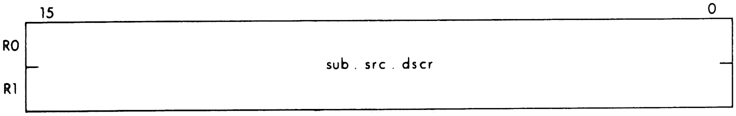
Register Form—SPANC

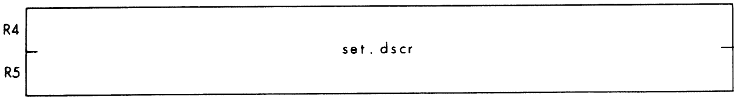
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the character set descriptor is placed in R4-R5.





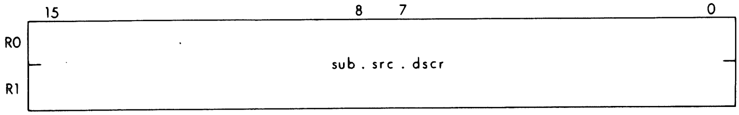
When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is not a member of the character set.





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.

	15 0
R0	srcl . dscr
R1	
R2	src2. dscr
R3	
R4	dst. dscr
R5	

When the instruction is completed, the source descriptor registers are cleared.

	15 0
R0	0
R1	0
R2	0
R3	0
R4	dst. dscr
R5	

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.

Supreme
Storwal

CHAPTER 8
TRAPS AND INTERRUPTS

An interrupt is a signal that breaks the normal flow of control of the routine being executed, transfering control to a specific location in memory. An interrupt is normally caused by an external event such as a done condition in a peripheral. The fast interrupt handling of PDP-11 processors reduces the time that system devices must wait for CPU service. Interrupts also relieve the processor from doing routine control functions for the peripherals. An interrupt is distinguished from a trap, which is caused by the execution of a processor instruction.

PDP-11 processor traps are triggered by power failures and certain hardware and software errors. A trap causes the processor to execute instructions pointed to by a certain permanently assigned address. Traps protect both the programmer and the processor. If a power failure occurs during program execution, the processor traps to the power-fail program, which saves the contents of registers. Traps also abort illegal instructions, such as attempts to address non-existent memory.

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, BPT, 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 63 Hz, as measured by DC voltage, the power-fail sequence is initiated. The central processor automatically traps through location 24 and the power-fail program has 2 msec to save all volatile information (data in registers), and to condition peripherals for power-fail.

When power is restored, the processor can be strapped to trap through location 24 and execute the power-up routine, which restores the machine to its state prior to power failure. (This feature is used only in systems with PROM or RAM with battery-backup. All other processors would power up the hardware bootstrap.)

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. *This error is not detected in the MICRO/T-11, SBC-11/21, LSI-11/2, LSI-11/23, Professional 300 series, MICRO/PDP-11, PDP-11/23 PLUS, or PDP-11/24.*

Time-out Errors

These errors occur when the processor attempts to access memory, or a device on the system bus, and there is no reply from the memory or device. This error usually occurs in attempts to address nonexistent memory or peripherals.

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

Reserved Instructions

There is a set of illegal and reserved instructions which cause the processor to trap through location 10. This may indicate that the desired instruction set (e.g. FPP or CIS) has not been installed in this particular processor. It may then be useful to emulate the instruction in software.

Vector Address and Trap Errors

000	(reserved)
004	CPU errors, stack overflow
010	Illegal and reserved instructions
014	BPT, breakpoint trap
020	IOT, input/output trap
024	Power-fail
030	EMT, emulator trap
034	TRAP instruction
114	Memory system errors
240	PIRQ, program interrupt request
244	Floating-point error
250	Memory management

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 (emulator trap) 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 registers, 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 needs to be passed between them, and, finally, cause a 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, including I/O, and file manipulation. 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
```

This finally expands to

```
          MOV #2.*400, R0
          BISB CHAN, R0
          EMT 374
```

Notice the EMT low byte is 374. This is interpreted by the EMT handler to indicate a group of routines. Then the contents of the high byte of R0 are tested by the handler to identify exactly which routine within the group is being requested, in this case routine number 2. The low byte of R0 contains the optional channel number.

Summary of PDP-11 Processor Trap Vectors:

VECTOR ADDRESS	FUNCTION SERVED
4	Illegal instructions (JSR, JMP for mode 0) Bus errors (odd address error, time-out) Stack limit (Red or Yellow Zones) Illegal internal address Microbreak

10	Reserved instruction XFC with UCS disabled SPL, MTPS, MFPS, FADD, FSUB, FMUL, FDIV if no FIS. HALT in user mode (jumper option)
14	Trace (T bit)
20	IOT instruction
24	Power-fail
30	EMT instruction
34	TRAP instruction
114	Cache parity error Bus memory parity error User Control Store parity error
240	PIRQ (program interrupt request)
244	Floating point exception
250	Memory management abort

INTERRUPTS

Interrupt-driven techniques are used to reduce CPU and peripheral 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 other low-priority programs 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 time. The CPU completes the instruction being executed and then is interrupted and vectors to an interrupt service routine. (For instructions such as those of the Commercial Instruction Set (CIS), which are potentially very long, the instruction is stopped in the middle and will be resumed when the interrupt is dismissed. This saves a long delay or latency waiting for the instruction to complete.) 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. Some of 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 would be the same 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 new PC), 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 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.

Note that the interrupt service routine must explicitly save the contents of any general registers that it requires before overwriting them. For example:

```
MOV R5, -(SP)                    ;Save R5

MOV R4, -(SP)                    ;and R4
```

Now, the service routine may freely use R4 and R5. At the end of the interrupt, the service routine must restore the original contents of any "borrowed" registers, noting the reversed order! It may then execute an RTI (Return from Interrupt) instruction to restore the original PC and PS. For example:

```
MOV (SP)+, R4                    ;Restore R4

MOV (SP)+, R5                    ;and R5

RTI                              ;Now back to the
                                 ;interrupted program
```

Caution When Clearing Device Interrrupt Enable Bits

Clearing device Interrupt Enable bits while the device is still active can lead to a bus time-out error when the processor attempts to receive the interrupt vector from that device. This can cause degradation of the computer system's throughput. Consider the example:

```
    PSW = 0
    CLR @ #177564
```

As a result, the DLV-11 Serial Line Unit Interrupt Enable bit is being cleared. Now, assume that the transmitter is still active and sending characters, and further assume that the Done bit in the status register becomes set shortly after the CLR instruction is fetched, but before the Interrupt Enable bit can be cleared. The device will now post an interrupt request because the Done bit has been set and the Interrupt Enable bit is still set. The CLR instruction will complete execution and the processor will recognize the interrupt request since there was not enough time for the device to disable the interrupt request. The processor will then attempt to obtain a vector from the interrupting device. However, a bus time-out error will occur because the device now has had enough time to remove the interrupt request and will not respond. The LSI-11/2 processor treats this time-out as a fatal condition and halts; other processors time out and them resume execution. If multiple interrupt requests were pending at this time, a time-out would not occur since the next device needing service would respond with its interrupt vector.

One method of avoiding this problem is to disable interrupts immediately before the Interrupt Enable bit is cleared. For example:

```
MTPS #200
CLR @#177564
MTPS #0
```

In this situation, enough time has been allowed for the interrupt request to be removed bv the device.

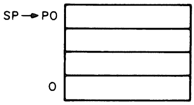
Nesting

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. Nested interrupt service routines and subroutines are illustrated in Figure 8-5.

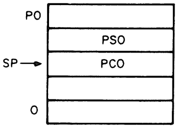
Nesting

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

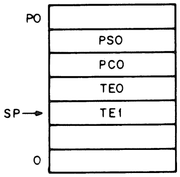
1. Process 0 is running; SP is pointing to location P0.



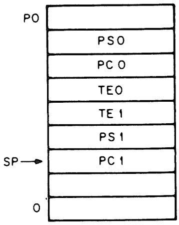
2. Interrupt stops process 0 with PC = PC0, and status = PS0; starts process 1.



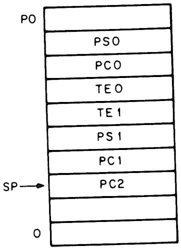
3. Process 1 uses stack for temporary storage (TEO, TE1).



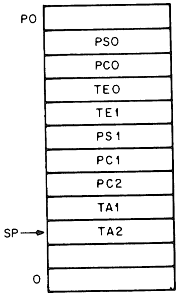
4. Process 1 interrupted with PC = PC1 and status = PS1; process 2 is started.



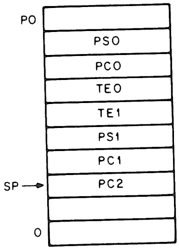
5. Process 2 is running and does a JSR R7,A to subroutine A with PC = PC2.



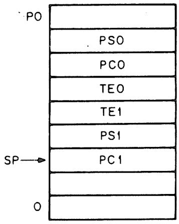
6. Subroutine A is running and uses stack for temporary storage.



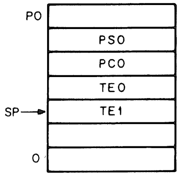
7. Subroutine A releases the temporary storage holding TA1 and TA2.



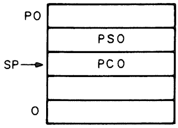
8. Subroutine A returns control to process 2 with an RTS R7; PC is reset to PC2.



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



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



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

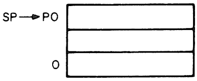
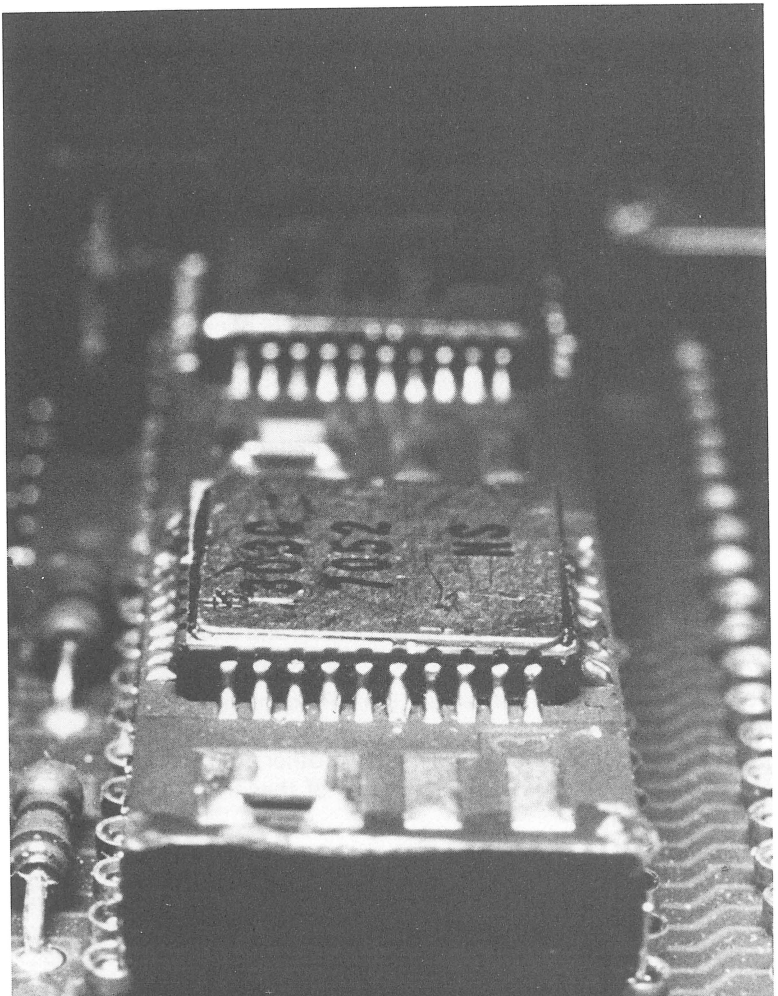


Figure 8-1 Nested Interrupt Service Routines and Subroutines

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



CHAPTER 9

MAPPING TO MEMORY AND BUSSES

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, or peripherals) 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 often 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.

To aid the executive in this task, the CPU contains various hardware features which make it easy to multiprogram and ensure that each program is protected from corruption by other programs. The Memory Management Unit (MMU) provides two of these features: relocation and protection.

CONCEPTS

Before describing the memory and bus-mapping 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 seen by the user's program. For instance, a program written for a PDP-11 processor sees 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 64 Kbytes (200000 octal bytes) is know 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 64 Kbytes. Figure 9-1 illustrates several user programs and their associated virtual address space.

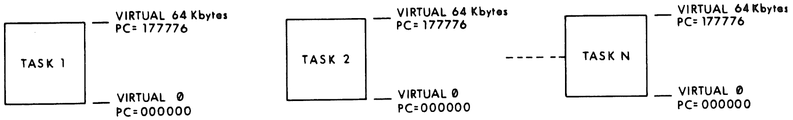


Figure 9-1 Program Virtual Address Space

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 64 Kbytes and the 18-bit space yields a total of 256 Kbytes. Since devices on the UNIBUS are addressable via an 18-bit address, it is clear that in both of the above cases (16-bit and 18-bit physical address space), main memory may be physically attached to the UNIBUS. The 22-bit space yields a total of 4096 Kbytes. 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 peripheral devices have been assigned addresses in the top 8 Kbytes of physical address space. The diagram in Figure 9-2 illustrates a typical physical address space including main memory and UNIBUS peripheral device register (I/O page) space.

The diagram in Figure 9-2 will be explained more fully during the discussion of 16-, 18-, and 22-bit mapping of processor addresses.

Address Relocation

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 (that is, its virtual address space ranges from 000000 upwards). When the processsor 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 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 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 noncontiguous segments known as pages. When this situation occurs, each individual page is assigned a different relocation constant.)

To illustrate this point, let's look at a simplified memory relocation example. In Figure 9-3, Program A's 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 6400_8, which is the second item of Program A, to be accessed. When Program B is executing, the relocation constant is changed to 100000_8. Then Program B's virtual addresses are relocated by the relocation constant 100000_8.

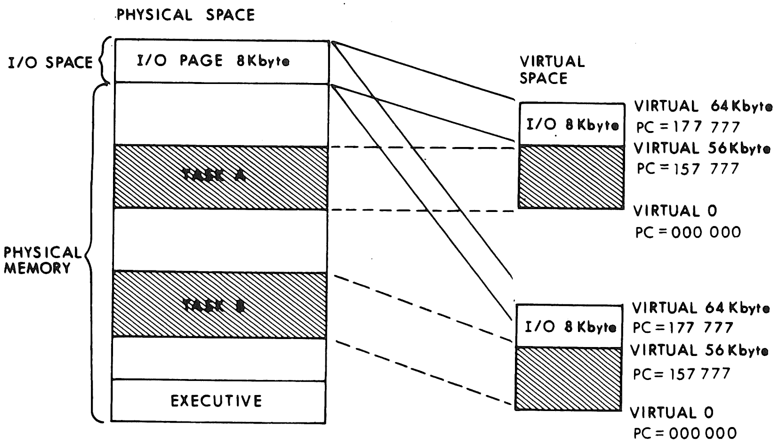


Figure 9-2 Physical Address Space

MEMORY MANAGEMENT

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

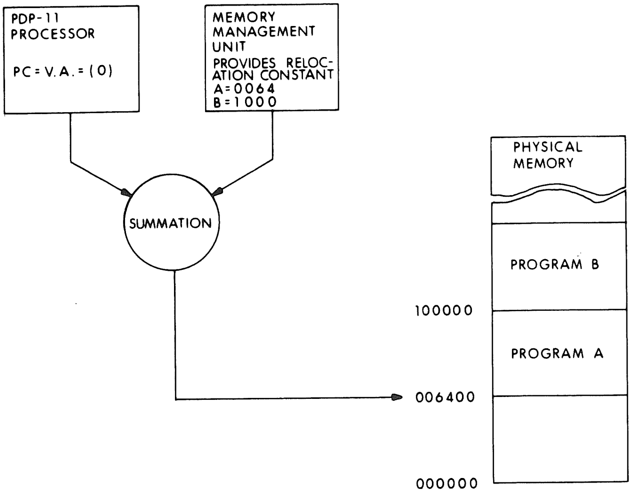


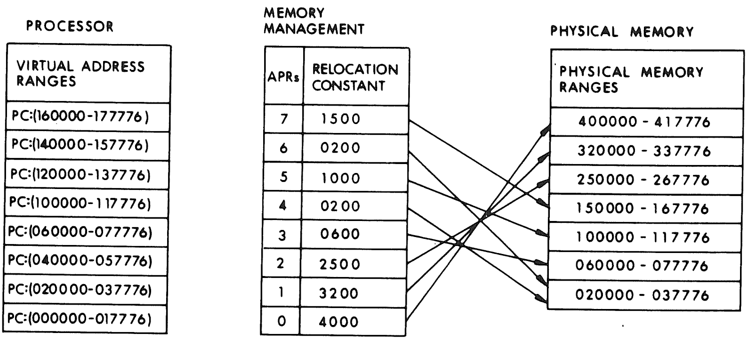
Figure 9-3 Simplified Memory Relocation Example

The basic function of memory management is to perform memory relocation and provide extended memory addressing capability for systems with greater than 56 Kbytes of physical memory. In order to perform this basic function, 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 users' 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 64 bytes in length. Thus the maximum length of a page is 8192 (128 × 64) bytes, and the maximum number of pages in the program is eight (8192 bytes × 8 = 64 Kbytes). Memory management contains 8 APRs for mapping virtual pages into physical memory. Using all of the eight available active page registers in a set, a maximum program length of 65,536 bytes 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 64 bytes. However, for pages that are smaller than 8 Kbytes, only the memory actually allocated to the page may be accessed.

The relocation example shown in Figure 9-4 illustrates several points about memory relocation. These points are:

1. Although the program appears to be in contiguous address space to the processor, the 64 Kbyte virtual address space is actually scattered through several separate areas of physical memory. As long as the total 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 9-4, 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 nonzero).
3. All of the pages shown in the example start on 64-byte 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 9-4, 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.



PROCESSOR

VIRTUAL ADDRESS RANGES
PC:(160000-177776)
PC:(140000-157776)
PC:(120000-137776)
PC:(100000-117776)
PC:(060000-077776)
PC:(040000-057776)
PC:(020000-037776)
PC:(000000-017776)

MEMORY MANAGEMENT

APRs	RELOCATION CONSTANT
7	1500
6	0200
5	1000
4	0200
3	0600
2	2500
1	3200
0	4000

PHYSICAL MEMORY

PHYSICAL MEMORY RANGES
400000 - 417776
320000 - 337776
250000 - 267776
150000 - 167776
100000 - 117776
060000 - 077776
020000 - 037776

Figure 9-4 Relocation of a 64-Kbyte Program into 248 Kbytes of Physical Memory

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, or operating system.

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

As mentioned above, 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 nonresident. To more fully understand these access control types, let's look at the memory requirements of a typical application program. If the application program can be contained within three pages of virtual space (24 Kbytes), then only three pages of main memory need be allocated by memory management as resident for that program. All other pages are assigned nonresident status. Therefore, the nonresident access key can be used to allocate physical memory efficiently. If the kernel contains an area that could be used but must be nonmodifiable, then that area is designated as read-only. It is also wise to mark any pure program code as read-only. However, if there is a database or a common data area in the users' space that must be updated constantly, e.g., a database of digital data or A/D conversion data, the database 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. They enhance the memory protection scheme and increase the flexibility and power of timesharing and multiprogramming environments. These different execution modes are sometimes available independently of the rest of memory management.

Kernel mode is the most privileged of the three modes and allows execution of any instruction. In an operating system featuring multiprogramming, 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 inadvertent 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 has the same privileges as the user mode, but uses different mapping. Supervisor mode 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 prohibits the execution of instructions such as HALT and RESET as does supervisor mode. A multiprogramming operating system will typically restrict execution of user programs to user mode to prevent a single user from harming the system as a whole. The user's virtual address space permits writing only into one's own area of memory. 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, data-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's page address register set. Relocation of trap addresses means that the hardware is capable of recovering from a failure in the first physical bank of memory.

When a trap, abort, or interrupt occurs, the "push" of the old PC and old 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 directly to a supervisor or user mode program by setting the CPU mode bits of the new PSW in the vector to return control to the appropriate mode. (Caution: This does not apply to some older PDP-11 processors. See Appendix B for details.)

Instruction and Data Space

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 64 to 128 Kbytes. 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 128 Kbytes consisting of 64 Kbytes of pure instructions or procedure code and 64 Kbytes 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 64 Kbytes of instructions and eight D space APRs accommodate up to 64 Kbytes of data. By using the separate I and D space APRs, a maximum 128 Kbyte 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 9-1.
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 calling sequences, for example, cannot be mapped entirely 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 9-1 illustrates the separation of I and D references for all address modes and all registers. Note that the 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 I space, and one for D space for each supported mode of operation. Figure 9-5 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 or D space.) When the memory management unit is turned on, the upper three bits, <15:13>, of the virtual address generated by the processor 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 PAR, illustrated in Figure 9-6, contains the Page Address Field (PAF) specifying the starting (base) address of the page as a block number in physical memory. The following processors have a 16-bit PAF:

- MICRO/PDP-11
- PDP-11/23 PLUS
- PDP-11/24
- PDP-11/44
- PDP-11/70

The PAR may be thought of as a relocation register containing a relocation constant, or as a base register containing a base address. Either way, the PAR is an important relocation tool.

Page Descriptor Register (PDR)

The PDR, illustrated in Figure 9-7, contains information relative to page expansion, page length, and access control.

Table 9-1 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 present in the CPU or 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.

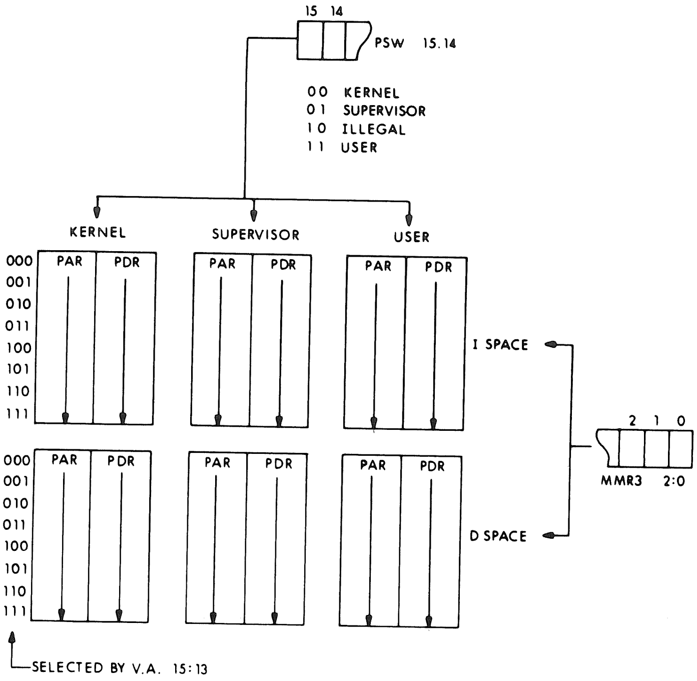


Figure 9-5 (PAR/PDR) Register Set

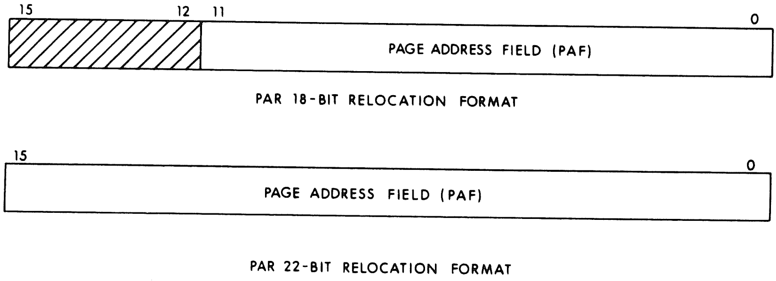


Figure 9-6 The Page Address Register

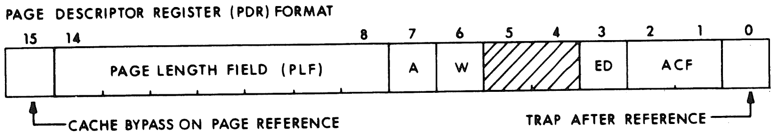

PAGE DESCRIPTOR REGISTER (PDR) FORMAT
15
14
8
7
6
5
4
3
2
1
0
PAGE LENGTH FIELD (PLF)
A
W
ED
ACF
CACHE BYPASS ON PAGE REFERENCE
TRAP AFTER REFERENCE

Figure 9-7 The Page Descriptor Register

Access Control Field (ACF) — Bits <2:0> of the PDR contain 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	nonresident	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	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 9-8 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 9-9 illustrates an example of downward page expansion.

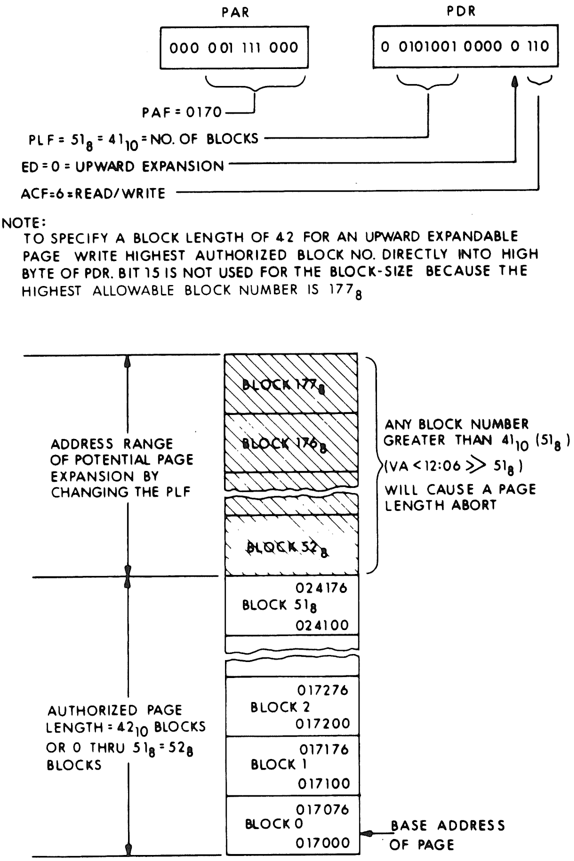
Access Information Bits — Bit <6> of the PDR, the Written Into (W) bit, indicates whether the page has been written into since it was loaded in memory. A logical 1 in bit <6> indicates a modified page. The W bit is automatically cleared when the PAR or PDR of that page is written into.

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; the disk copy is valid.

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 specificed 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.

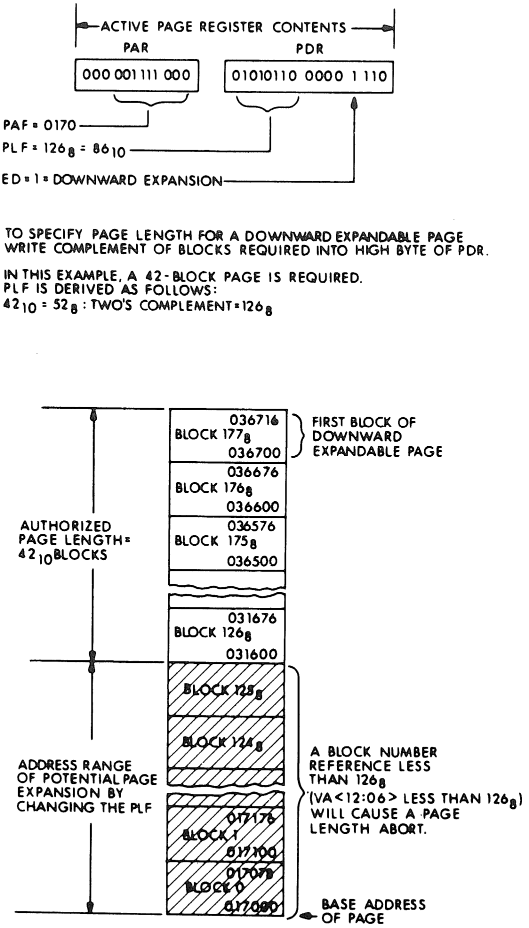

PAR
000 001 111 000
PDR
0 0101001 0000 0 110
PAF = 0170
PLF = 51_8 = 41_{10} = NO. OF BLOCKS
ED = 0 = UPWARD EXPANSION
ACF=6 = READ/WRITE
BLOCK 177_8
BLOCK 176_8
BLOCK 52_8
ADDRESS RANGE OF POTENTIAL PAGE EXPANSION BY CHANGING THE PLF
ANY BLOCK NUMBER GREATER THAN 41_{10} (51_8) (VA <12:06 >> 51_8) WILL CAUSE A PAGE LENGTH ABORT
024176
BLOCK 51_8
024100
017276
BLOCK 2
017200
017176
BLOCK 1
017100
017076
BLOCK 0
017000
AUTHORIZED PAGE LENGTH = 42_{10} BLOCKS OR 0 THRU 51_8 = 52_8 BLOCKS
BASE ADDRESS OF PAGE

NOTE:
TO SPECIFY A BLOCK LENGTH OF 42 FOR AN UPWARD EXPANDABLE PAGE WRITE HIGHEST AUTHORIZED BLOCK NO. DIRECTLY INTO HIGH BYTE OF PDR. BIT 15 IS NOT USED FOR THE BLOCK-SIZE BECAUSE THE HIGHEST ALLOWABLE BLOCK NUMBER IS 177_8

Figure 9-8 Upward 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 56 Kbytes of main memory and 8 Kbytes 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


ACTIVE PAGE REGISTER CONTENTS
PAR
PDR
000 001 111 000
01010110 0000 1 110
PAF = 0170
PLF = 126_8 = 86_{10}
ED = 1 = DOWNWARD EXPANSION
036716
BLOCK 177_8
036700
FIRST BLOCK OF DOWNWARD EXPANDABLE PAGE
036676
BLOCK 176_8
036600
036576
BLOCK 175_8
036500
AUTHORIZED PAGE LENGTH = 42_{10} BLOCKS
031676
BLOCK 126_8
031600
BLOCK 125_8
BLOCK 124_8
ADDRESS RANGE OF POTENTIAL PAGE EXPANSION BY CHANGING THE PLF
017176
BLOCK 1
017100
017076
BLOCK 0
017000
A BLOCK NUMBER REFERENCE LESS THAN 126_8 (VA<12:06> LESS THAN 126_8) WILL CAUSE A PAGE LENGTH ABORT.
BASE ADDRESS OF PAGE

TO SPECIFY PAGE LENGTH FOR A DOWNWARD EXPANDABLE PAGE WRITE COMPLEMENT OF BLOCKS REQUIRED INTO HIGH BYTE OF PDR.

IN THIS EXAMPLE, A 42-BLOCK PAGE IS REQUIRED.
PLF IS DERIVED AS FOLLOWS:
42_{10} = 52_8 : TWO'S COMPLEMENT = 126_8

Figure 9-9 Downward Page Expansion

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 64 bytes, each page contains a maximum of 8192 bytes.

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 9-10.

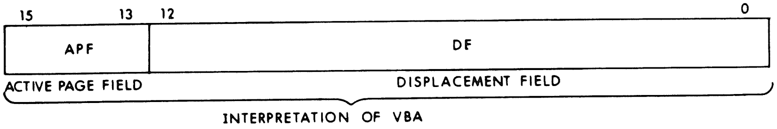


Figure 9-10 Interpretation of a Virtual Address with Memory Management Enabled

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

- The Active Page Field (APF)—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)—a 13-bit field, <12:0>, containing an address relative to the beginning of a page. This permits page lengths up to 8 Kbytes. The displacement field is further subdivided into two fields as illustrated in Figure 9-11.

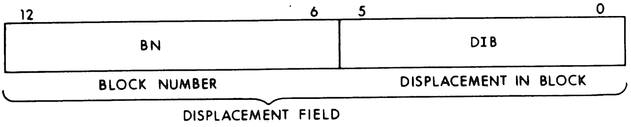


Figure 9-11 Interpretation of Displacement Field

The displacement field (DF) consists of:

- The physical memory Block Number (BN)—a 7-bit field, <12:06>, which is interpreted as the block number (0-127) within the current page.
- The Displacement in the Block (DIB)—a 6-bit field, <5:07>, which contains the displacement within the block (0-63 bytes) 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 9-6, 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 9-5.)
2. The APF of the Virtual Bus Address (VBA) is used to select a PAR (PAR0-PAR7). (Refer to Figure 9-10).
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 9-6.)
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 9-12.

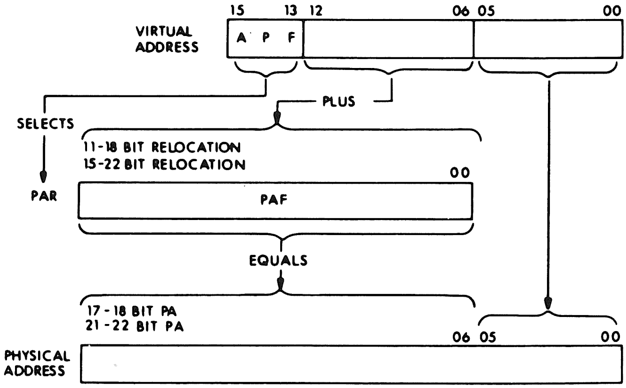


Figure 9-12 Virtual to Physical Address Translation

At this point, let's 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, 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 9-13.

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 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 9-6.) The actual flow of translation is illustrated in Figure 9-14.

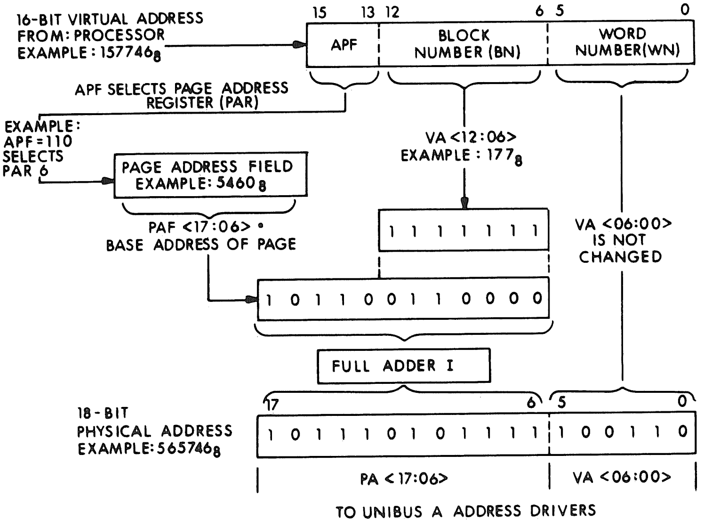


Figure 9-13 16-Bit Virtual to 18-Bit Physical Address Translation

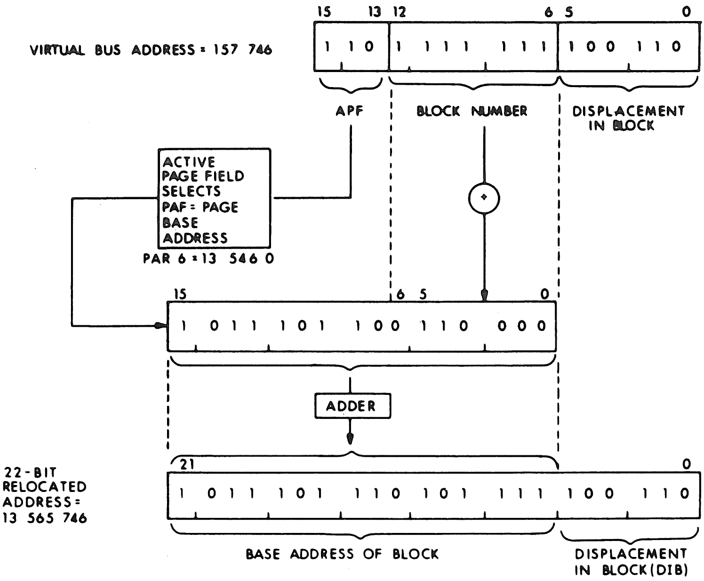


Figure 9-14 16-Bit Virtual to 22-Bit Physical Address Translation

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 that only processors supporting a 22-bit physical address space use the UNIBUS map.) Before introducing specific mapping diagrams, let's look at functional block diagrams of several PDP-11 processors and the physical address space supported by each.

Figure 9-15 illustrates the LSI-11/2 processor. This processor contains neither a memory management unit nor a UNIBUS map. It can access a maximum of 56 Kbytes of main memory and the 8-Kbyte I/O page.

The physical address space supported by the LSI-11/2 is illustrated in Figure 9-16. Main memory is physically attached to the LSI-11 Bus.

Figure 9-17 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 248 Kbytes of main memory in addition to the 8 Kbyte I/O page.

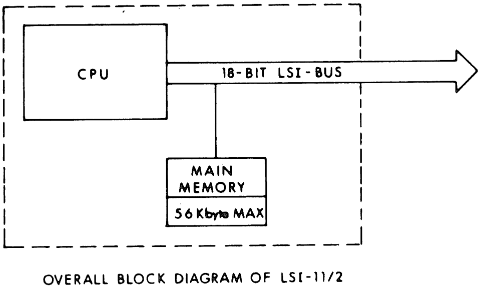


Figure 9-15 Block Diagram of the LSI-11/2

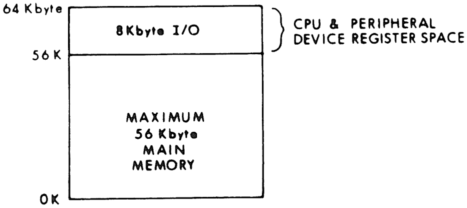

64 Kbyte
56 K
0 K
8 Kbyte I/O
CPU & PERIPHERAL DEVICE REGISTER SPACE
MAXIMUM 56 Kbyte MAIN MEMORY

Figure 9-16 LSI-11/2 Physical Address Space

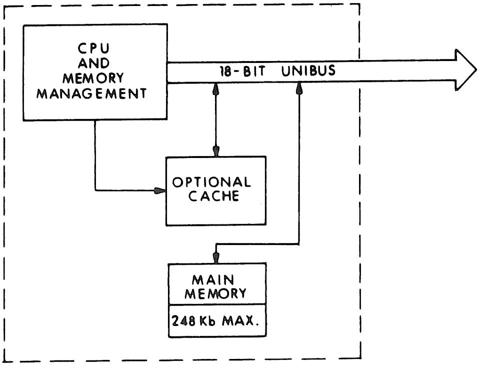


Figure 9-17 Block Diagram of the PDP-11/34A

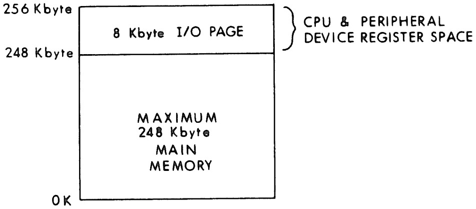

256 Kbyte
248 Kbyte
0K
8 Kbyte I/O PAGE
CPU & PERIPHERAL DEVICE REGISTER SPACE
MAXIMUM 248 Kbyte MAIN MEMORY

Figure 9-18 PDP-11/34A Physical Address Space

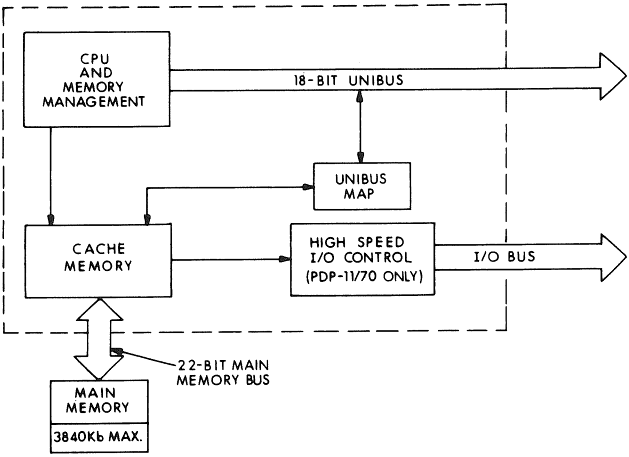


Figure 9-19 Block Diagram of the PDP-11/70

The physical address space supported by the PDP-11/34A is illustrated in Figure 9-18. Main memory is physically attached to the UNIBUS.

This next section describes memory management for the PDP-11/24, PDP-11/44, and PDP-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 has the same functions. Figure 9-19 illustrates a simplified block diagram of a typical PDP-11/70 processor. The memory management hardware translates the user's 16-bit virtual addresses into 22-bit physical addresses. The UNIBUS 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, PDP-11/44, and PDP-11/70 CPUs is illustrated in Figure 9-20.

Referring to Figure 9-20, observe the following points:

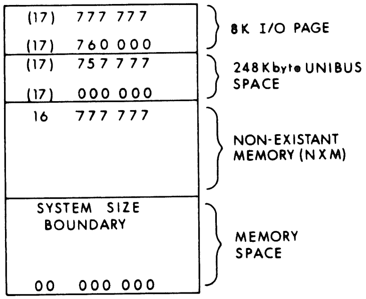


Figure 9-20 PDP-11/24 and 11/44 Physical Address Space

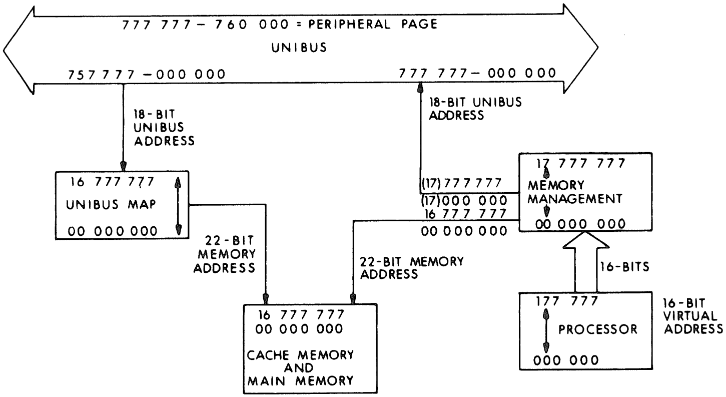


Figure 9-21 22-bit Address Space Bus Configuration

1. UNIBUS references include 256K 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, reserved for UNIBUS device registers, consists of 8 K physical byte addresses, 17 760 000 - 17 777 777 (UNIBUS addresses760 000 - 777 777).

   b. The remaining 248K 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's main memory. There may be no discontinuity in main memory. 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 3840 Kbytes (2^{21} — 2^{17} = 3,832,160 or 4096 Kbytes – 256K = 3840K).
3. NoneXistent Memory (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 9-21.

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 9-21, 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 9-20, 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 (this does not apply to certain MICRO/J-11 processors). Therefore all addresses within the range of 00 000 000 and 16 777 777 are directed to cache (if present) 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 four bits are all set to 1. Therefore, after the hardware strips off the most significant four bits (17_8), we are left with the familiar 18-bit (256 Kbyte) 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 addresses 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 8 Kbytes of the I/O page are not touched.) The UNIBUS map is described later in this chapter.

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 9-22 illustrates the 16-bit mapping scheme for processors such as the LSI-11/2 and PDP-11/34A. This illustration shows fixed relocation mapping from virtual to physical addresses. The lowest 56K of virtual addresses are treated as corresponding to the same lower 56K of physical addresses. With the PDP-11/24, PDP-11/44, and PDP-11/70 in 16-bit mode, the lower 56K of virtual addresses address main memory (not attached to the UNIBUS). However, the top 8K virtual addresses always cause UNIBUS cycles to address the top 8K physical addresses no matter what size the physical address space might be. In this example, the top 8K virtual addresses reference physical addresses 248K - 256K.

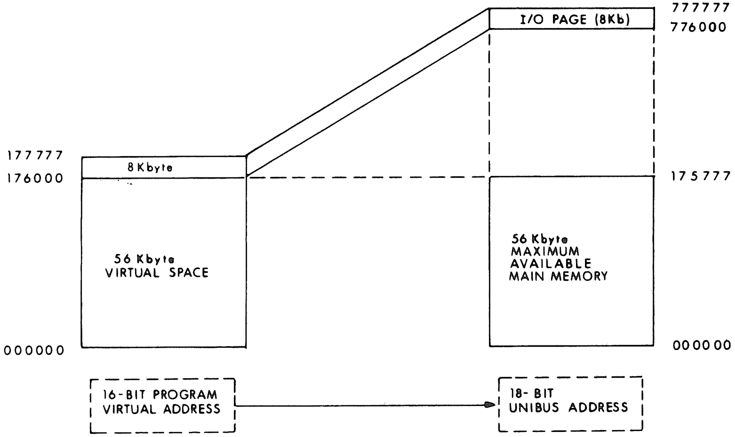


Figure 9-22 16-Bit Mapping within 18-Bit Physical Address Space

18-Bit Physical Address Space

Figure 9-23 illustrates the 18-bit mapping scheme for processors with memory constraints of 248 Kbytes. Figure 9-23 depicts the fact that with memory management enabled, the user's virtual address space of 56 Kbytes can be relocated anywhere in available main memory (in 8K word pages—if necessary, refer back to Figure 9-4 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 9-22. With the PDP-11/24 and PDP-11/44 18-bit memory management mode, the lower 56K of virtual addresses address main memory (not attached to the UNIBUS) using relocation.

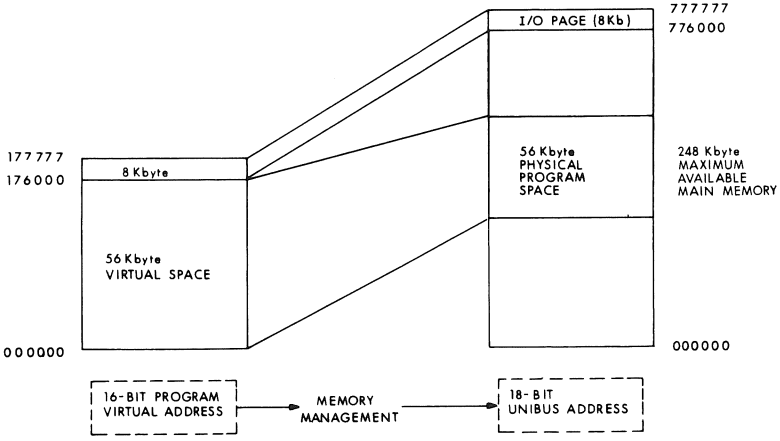


Figure 9-23 18-Bit Mapping within 18-Bit Physical Address Space

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, PDP-11/44, or PDP-11/70 system is running in 16-bit mapping mode, then the 16-bit mapping scheme (memory management disabled) is illustrated in Figure 9-24. If 18-bit memory management is enabled, the mapping scheme is illustrated in Figure 9-25. The 22-bit mapping structure is illustrated in Figure 9-26. The solid arrow lines in Figure 9-26 represent a one-to-one correspondence between physical address and physical location.

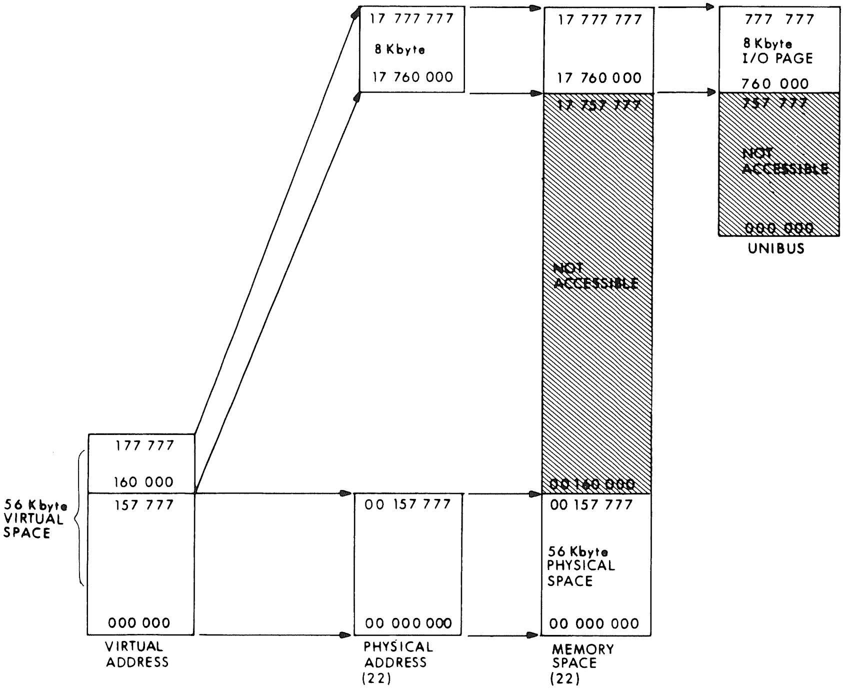


Figure 9-24 16-Bit Mapping Structure for 22-Bit Physical Address Space

LSI-11 Bus Physical Addressing

The LSI-11 Bus allows the direct use of the 22-bit physical address created by the memory management unit. Unlike the UNIBUS, no distinction need be made between references intended for memory (via the memory bus) and references intended for I/O devices (via the UNIBUS). No UNIBUS exists, therefore no address space need be reserved for it. Up to 4088 Kbytes of physical memory may exist; above that is the standard 8 Kbyte I/O page.

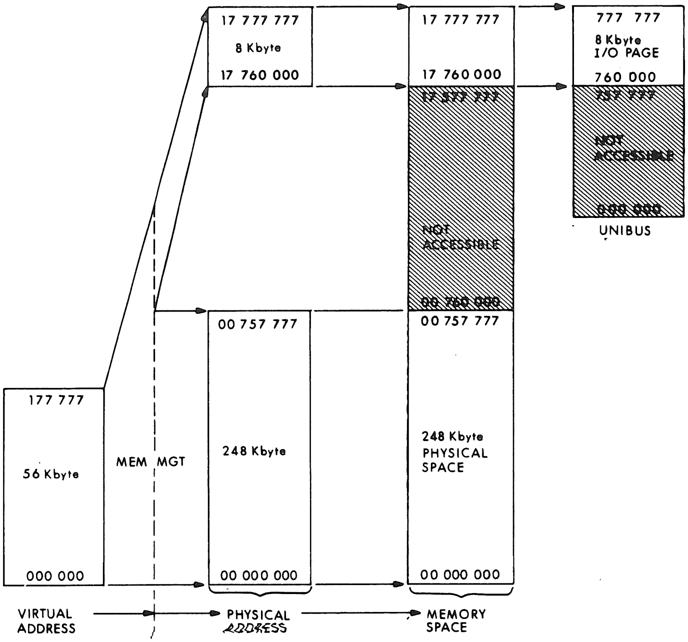


Figure 9-25 18-Bit Mapping Structure for 22-Bit Physical Address Space

I/O EXTENDED ADDRESSING

History

When the PDP-11 was first developed, the virtual and physical address space accessible by the processor was 64 Kbytes. This limit was imposed by the 16-bit word length. However, it was envisioned that the PDP-11 might grow into a family of both larger and smaller machines. In order to accommodate this, the physical I/O bus—the UNIBUS—was designed to accommodate 18-bit addressing, allowing a maximum of 248 Kbytes of memory and an 8 Kbyte I/O region. Thus, any I/O device doing Direct Memory Access (DMA) could address the entire 256 Kbyte UNIBUS space. At the time, this was thought to be more than adequate.

However, history has shown that any computer concept grows to fill and then surpass its original intent. As users' needs grew, the PDP-11 family was expanded; the PDP-11/40 added the necessary hardware to

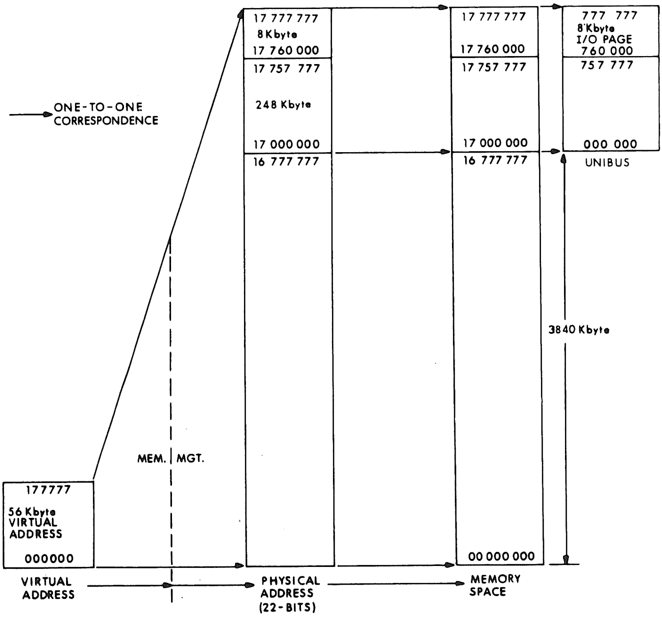


Figure 9-26 22-Bit Mapping Structure for 22-Bit Physical Address Space

allow the user to access the full 248 Kbyte physical memory space. The PDP-11/45 allowed the user to expand the virtual address space. With the appearance of the 11/70, the physical memory space was expanded to 3840 Kbytes (representing 22 address bits). The original 18-bit I/O bus had been outgrown.

Two solutions were developed.

22-Bit I/O Controllers

The 11/70 contains a 22-bit path for physical addresses. This path is known variously as the memory bus or the cache bus. The 11/70 also contains embedded, high-speed, general purpose I/O controllers called RH70's. These interface to the memory bus and create a bus known as the MASSBUS, to which disks, tapes, and other mass-storage devices may be attached. Since the RH70 Massbus controllers have a direct path to memory and are capable of directly generating a 22-bit address, they allow DMA to occur anywhere in the physical address space. This is illustrated in Figure 9-27. Note that the 22-bit bus within the 11/70 is bounded (cannot interface to certain devices, unlike the UNIBUS).

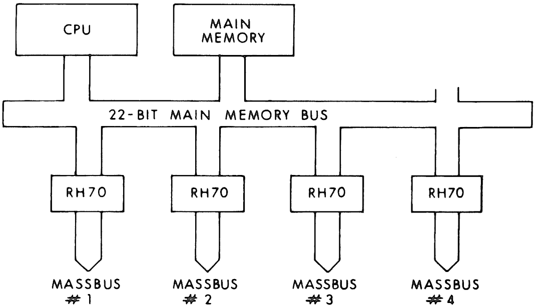


Figure 9-27 RH70 22-bit I/O Controllers

The UNIBUS Map

The presence of RH70s in the 11/70 should not imply that Digital abandoned the UNIBUS. It was necessary to support all the devices which pre-dated the arrival of the 11/70; in addition, the UNIBUS is much easier and less expensive to interface to than the Massbus. This meant that new I/O devices would continue to be developed to use the UNIBUS. Also, the 11/70 was limited to a maximum of 4 RH70 22-bit controllers; the UNIBUS, on the other hand, allows all the traditional expansion capacity. In order to accommodate those devices which connect to the 18-bit UNIBUS, the PDP-11/70 contains a scheme whereby the 18-bit UNIBUS address can be mapped to the 22-bit memory address. This action is performed by the UNIBUS Map. The UNIBUS Map is illustrated in Figure 9-28.

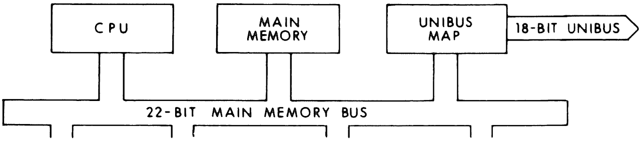


Figure 9-28 UNIBUS Map

Subsequently, the PDP-11/24 and PDP-11/44 were introduced. These processors contain no integral Massbus adapter but do provide a UNIBUS Map.

LSI-11 Bus

As the LSI-11 bus developed, it grew just as the UNIBUS has grown. Initially a 16-bit bus with reserved capacity to take it to 18-bits, the bus was subsequently expanded to 22-bits. No LSI-11 Bus map was created; rather, all LSI-11 Bus DMA controllers behave like the RH70 in that they directly issue 22-bit addresses. Figure 9-29 illustrates the similarity between the PDP-11/70/RH70 scheme and the LSI-11 Bus scheme. Note that, unlike the PDP-11/70, the LSI-11 Bus does not impose a limit on the number of 22-bit controllers which may be installed.

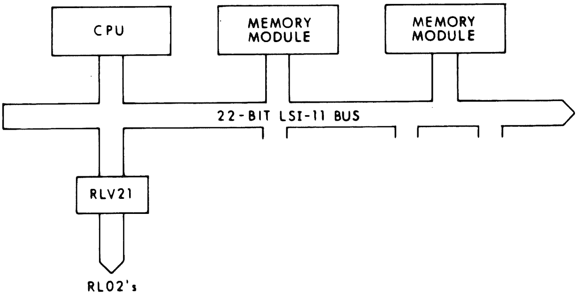


Figure 9-29 LSI-11 Bus 22-Bit I/O Controllers

Operation of the RH70 and LSI-11 Bus Controllers

DMA operation of either an RH70 or a LSI-11 Bus controller is straightforward. The controller arbitrates for use of the bus; when it wins use of the bus, it asserts a 22-bit address directly from its bus address register to the bus. This 22-bit address controls which physical memory location is accessed by the DMA.

Operation of the UNIBUS Map

The UNIBUS Map operates in a manner very similar to the CPU's memory management unit. The 256 Kbyte UNIBUS space is divided into 32 pages of 8 Kbytes each. Each of these pages maps via a 22-bit relocation (base) register. This is very similar to the way the CPU maps its 64 Kbyte address space via 8 pages, also of 8 Kbytes each.

The mapping operation is illustrated in more detail in Figure 9-30. An 18-bit address on the UNIBUS is broken into two fields. The upper 5 bits select one of the 32 pages, this in turn selects a mapping register.

The 22-bit contents of the mapping register is then added to the remaining 13-bits of the UNIBUS address to derive a 22-bit main-memory address. The actual DMA then takes place at that address.

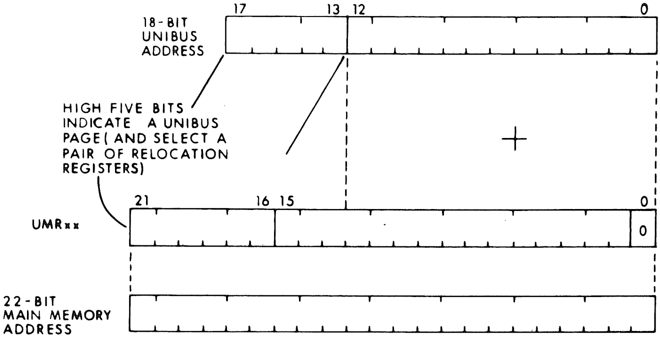


Figure 9-30 Construction of a Physical Address via the UNIBUS Map

Figures 9-31 and 9-32 offer a practical example. Assume that the UNIBUS Map has been enabled, and that the first mapping- register pair contains the value 01 760 000. If a UNIBUS device attempts to access UNIBUS location 001002, that address will be picked up by the map. The high 5 bits are examined; in this case they are 00000. This indicates that the UNIBUS address is in the first 8 Kbyte page and will be mapped via the first UNIBUS mapping register pair. Now the content of the mapping register is fetched. That content is added to the low 13 bits of the UNIBUS address. The result of the addition is 01 761 002. That is the address used in accessing main memory. Figure 9-31 shows this graphically; figure 9-32 shows the actual addition.

Although the map may contain 32 sets of relocation registers, only 31 are actually used. The 32nd represents the topmost 8 Kbytes of the UNIBUS, known as the I/O page. The UNIBUS Map always relocates references in that area to the topmost 8 Kbytes of the memory bus. You can think of the 32nd register as having a fixed value.

Since the mapping registers must contain a 22-bit value, they are actually implemented as pairs of 16-bit registers. The registers are pictured in Figure 9-33; their addresses may be found in Table 9-2. Note that the bit <00> of the relocation constant cannot be set. This means that all word-aligned transfers on the UNIBUS remain word-aligned on the memory bus.

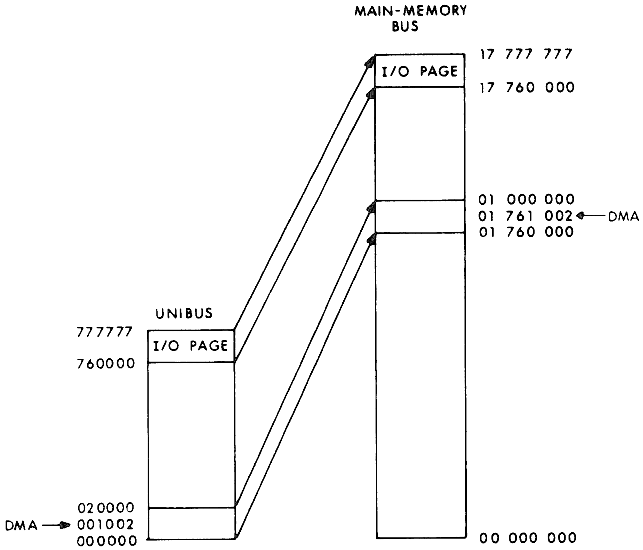


Figure 9-31 Example of Address Mapping with UNIBUS Map *ON*

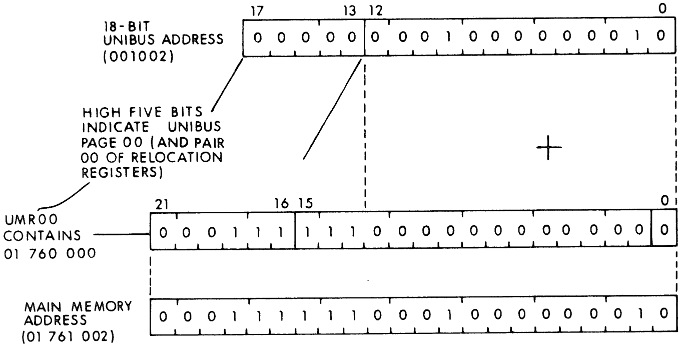


Figure 9-32 Mathematics of Address Mapping with UNIBUS Map *ON*

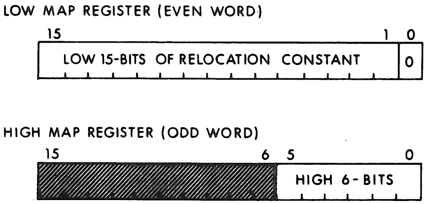

LOW MAP REGISTER (EVEN WORD)
15
1
0
LOW 15-BITS OF RELOCATION CONSTANT
0
HIGH MAP REGISTER (ODD WORD)
15
6
5
0
HIGH 6-BITS

Figure 9-33 UNIBUS Mapping Register Pair

Table 9-2 Addresses of UNIBUS Map Registers

UNIBUS Page	Low Word Address	High Word Address
0	17 770 200	17 770 202
1	17 770 204	17 770 206
2	17 770 210	17 770 212
3	17 770 214	17 770 216
4	17 770 220	17 770 222
5	17 770 224	17 770 226
6	17 770 230	17 770 232
7	17 770 234	17 770 236
10	17 770 240	17 770 242
11	17 770 244	17 770 246
12	17 770 250	17 770 252
13	17 770 254	17 770 256
14	17 770 260	17 770 262
15	17 770 264	17 770 266
16	17 770 270	17 770 272
17	17 770 274	17 770 276
20	17 770 300	17 770 302
21	17 770 304	17 770 306
22	17 770 310	17 770 312
23	17 770 314	17 770 316

UNIBUS Page	Low Word Address	High Word Address
24	17 770 320	17 770 322
25	17 770 324	17 770 326
26	17 770 330	17 770 332
27	17 770 334	17 770 336
30	17 770 340	17 770 342
31	17 770 344	17 770 346
32	17 770 350	17 770 352
33	17 770 354	17 770 356
34	17 770 360	17 770 362
35	17 770 364	17 770 366
36	17 770 370	17 770 372
37*	17 770 374	17 770 376

* This register pair is read/write but is not used for UNIBUS mapping.

When the processor is first started, the map is disabled. In this mode it relocates the low 248 Kbytes of the UNIBUS to the low 248 Kbytes of main memory. In addition, the top 8 Kbytes of the UNIBUS are relocated to the top 8 Kbytes of the memory bus. This is shown graphically in Figure 9-34. In this mode, our DMA occurring at UNIBUS address 001002 is passed directly to main memory address 00 001 002.

This mode of operation allows programs written prior to UNIBUS-mapping to continue to operate correctly. However, DMA cannot access any main memory address above 00 760 000. A program desiring to use the full expansion and relocation capabilities of the map can enable the map, operating henceforth with all the new capabilities.

The entire mapping operation is controlled by a single bit in memory management CSR3. The bit is zeroed after the processor starts and must be set to 1 by the program. The register also contains other bits; one enables the CPU to use 22-bit mapping. It is convenient to set all these bits in one reference. MMU CSR3 is illustrated in Figure 9-35; note that the implemented bits vary between different CPUs.

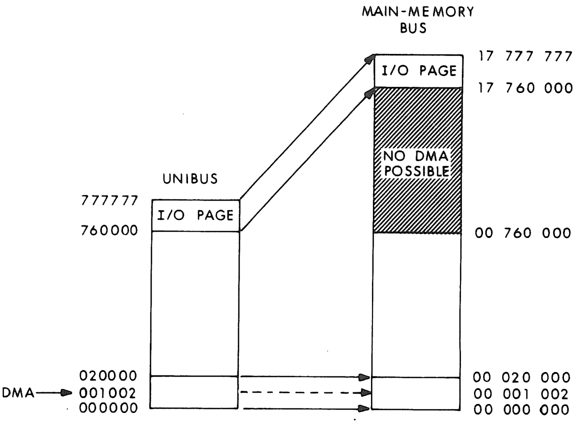


Figure 9-34 Address Mapping with UNIBUS Map *OFF*

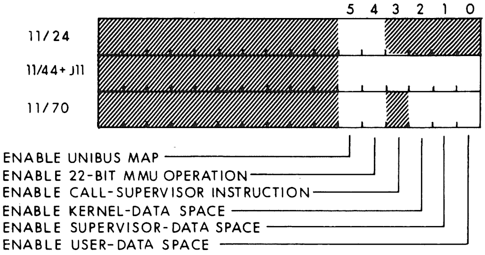


Figure 9-35 MMU CSR3 (17 772 516)

Mapping and DIGITAL Operating Systems

When using standard DIGITAL system software, all the processor-specific details of the I/O system are concealed from the user. The operating system's device drivers are responsible for handling the mapping scheme.

Within the operating system, UNIBUS mapping register pairs are typically allocated like any other system resource (via static assignment and queues). If you choose to write your own device driver, routines within the executive of each operating system can help you allocate and deallocate UNIBUS mapping registers.

Use of UNIBUS Memory or Memory Look-Alikes

Any address presented on the UNIBUS is ordinarily picked up by the map and translated to a main memory address. This is illustrated in Figure 9-36. If the translated address represents an existing location in main memory, that location will respond; no UNIBUS addresses are blocked by a 'fence'. Generally, in a correctly running system, any UNIBUS page in use is mapped to an existing main memory page.

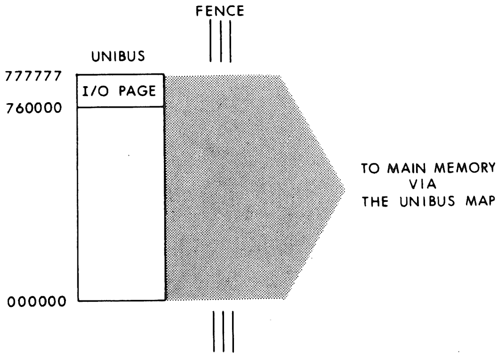


Figure 9-36 Normal UNIBUS Mapping

However, certain applications place memory, or devices that act like memory, on the UNIBUS. Examples of such applications include :

- Shared (multi-ported) UNIBUS memory
- Certain graphics devices (bit-mapped graphics, or devices which fetch instructions from PDP-11 memory)
- Bus windows (DA11-F)

In these cases, an address presented on the UNIBUS may actually be intended for the memory on the UNIBUS. It would be undesirable for the UNIBUS Map to pick up such an address and translate it to a main memory address. Therefore, some 'fence' must exist to block the UNIBUS Map from picking up and translating those addresses represented by your UNIBUS memory or memory look-alike. Figure 9-37 illustrates the UNIBUS address-space in this situation.

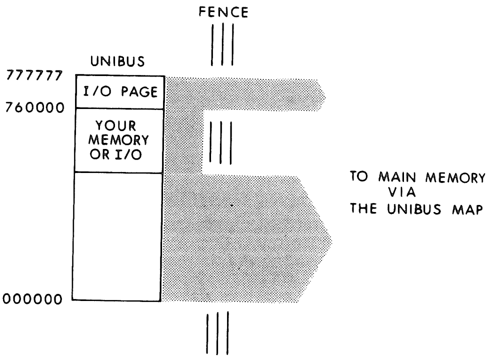


Figure 9-37 Disallowing Mapping of Part of the UNIBUS Address Space

Two methods exist to provide the 'fence' and prevent the address conflict described above:

1. If you have less than 3084 Kbytes of main memory, you can ensure that you set up the UNIBUS Map such that the area of the UNIBUS address-space occupied by UNIBUS Memory is mapped to nonexistent main memory. This ensures that main memory won't respond. However, if you add more memory to your system, that area of main memory may actually exist, and your system may fail.
2. Jumpers on the UNIBUS Map disallow its mapping operation on a page-by-page basis. The method varies from CPU to CPU, but in all cases, a contiguous group of pages can be reserved for the UNIBUS memory. Generally, these pages should be immediately below the I/O page (as shown in the previous figure).

Further Information

A detailed discussion of UNIBUS Mapping as implemented in the PDP-11/70 may be found in the KB11-C Maintenance Manual (EK-KB11C-TM-001).

Each processor's specific user's manual and technical manual can provide details on the setup and operation of that processor's UNIBUS Map.

FAULT RECOVERY (STATUS) REGISTERS

Aborts and traps generated by the memory management hardware are vectored through the kernel's 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.

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 because flags to the right are less significant and should be ignored. That is, a service routine for "Fault: nonresident" would ignore the length and memory management access control flags. A page length service routine would also ignore memory management access control faults.

Bits <15:13>, when set (by 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 set under program control. No abort will occur, but the contents of the memory management registers will be locked up as in an abort. Once the fault has been handled, the program should clear the offending bit(s).

Abort—Nonresident Bit 15 — — Bit <15> is the abort nonresident bit. It is set by attempting to acess 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: neither kernel, supervisor, nor user).

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.

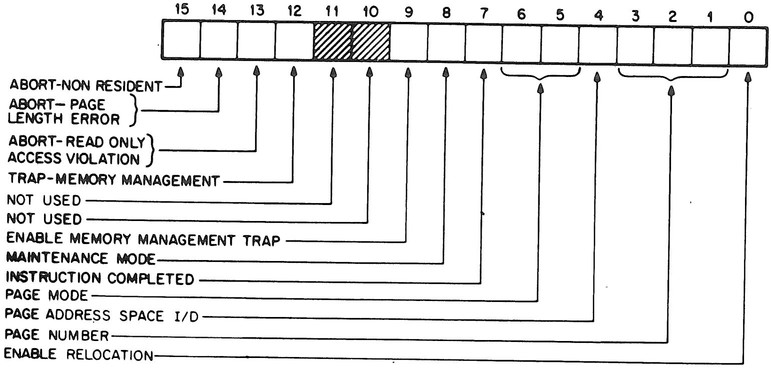


Figure 9-38 Memory Management Register 0 (MMR0)

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.

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

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.

Processor 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)
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 man-

agement unit is inoperative and addresses are neither relocated nor protected.

Memory Management Register 1 (MMR1) (PDP-11/44, J-11)
MMR1 records any autoincrement or autodecrement 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 two'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 or autodecrement 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 9-39.

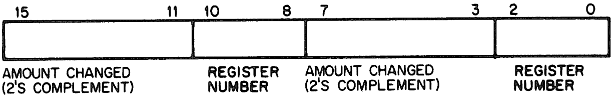

15
11
10
8
7
3
2
0
AMOUNT CHANGED (2'S COMPLEMENT)
REGISTER NUMBER
AMOUNT CHANGED (2'S COMPLEMENT)
REGISTER NUMBER

Figure 9-39 Memory Management Register 1 (MMR1)

NOTE

For the MICRO/PDP-11, LSI-11/23, PDP-11/23 PLUS, and 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) (J-11, MICRO/PDP-11, LSI-11/23, PDP-11/23 PLUS, PDP-11/24, and PDP-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 J-11 and PDP-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 9-40.

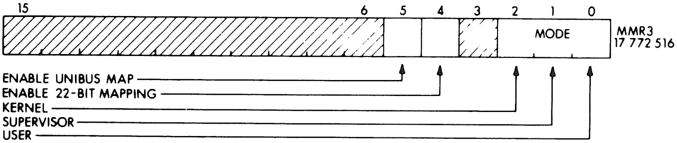


Figure 9-40 Memory Management Register #3 (MMR3)

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

Bit	State	Operation
3	1	Enable the Call Supervisor instruction
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.

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 adressing 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.

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

CHAPTER 10

PDP-11 BUS STRUCTURES

PDP-11s and LSI-11s both use an asynchronous bus for I/O. Implementation details differ between the LSI-11 Bus and the UNIBUS, but the architecture underlying the two busses is the same:

- Each bus operates with a strict master/slave relationship. When a device needs to use the bus, it arbitrates with the other contenders. When a device is the highest priority request, it wins control of the bus. It becomes the bus master and controls all data transfers, until it releases the bus. In performing transfers, it addresses another device, which is designated the slave during that bus-cycle.
- Each bus operates asynchronously: each transfer executes as quickly as the master and slave are able. Conversely, a slow slave can take as long as is required, only slowing down those bus cycles in which it is directly involved.
- Each bus overlaps the current data cycle with the arbitration for the next cycle. This enhances system performance.
- Each bus reserves the top 8 Kbytes of its address space for I/O and peripheral devices. DIGITAL implements some controllers at fixed addresses within this space; other controllers' addresses "float" based on a particular system's configuration.

  Memory may or may not exist on the same bus; however, if implemented on the same bus, its access protocol is the same as for I/O. Memory may be located on a private bus to provide faster data access, and/or more address space.

  The UNIBUS and the LSI-11 Bus can each be divided into four sections:

- Initialization
- Arbitration
- Data transmission
- Miscellaneous

The **initialization** lines of the bus provide the information required to start the processor after powerup, and cause an orderly shutdown of the processor during power failures. In addition, they allow the processor to reset the I/O subsystem.

The **arbitration** lines control access to the data transmission portion of the bus.

The **data transmission** lines allow words or bytes to be moved about on the bus. Transmission of data is always done with one device acting as master and the other acting as slave. The master controls the direction and length of transmission.

The **miscellaneous** lines provide other functions not described above. These functions include: processor control, memory refresh, and timekeeping.

Most of the signal lines are implemented as open-collector, wire-OR'ed signals and are asserted by being pulled "low" (hence the "L" which follows the signal name). The arbitration grant lines are not wired-OR; rather, they are passed from one I/O module to the next in daisy-chained fashion. Each I/O module either passes or receives/retransmits these grant signals. On the UNIBUS, these grant signals are active while "high."

Each of the fast signals on the bus is carried on a 120Ω circuit.

INITIALIZATION

The three lines of the initialization section are listed below. The UNIBUS lines are named first, and the LSI-11 Bus lines are in parentheses.

- BUS DCLO L (BDCOK H)
- BUS ACLO L (BPOK H)
- BUS INIT L (BINIT L)

The **BUS DCLO L** line indicates whether or not there is sufficient DC power for the computer to operate correctly. While low, there is insufficient power to operate and vice versa.

The **BUS ACLO L** line indicates whether or not line power is available. When the line makes the transition from high to low, it indicates to the CPU that it must begin its powerfail sequence, since AC power has just failed. The CPU will have at least 2 mS to perform necessary actions.

The **BUS INIT L** signal is used to initialize all bus devices. It is automatically driven at powerup, when the CPU is manually started, or when the RESET instruction is executed.

Figure 10-1 shows the approximate, relative timing among these three signals. The actual DC power is shown for reference.

ARBITRATION

There are eleven arbitration lines on the UNIBUS; by distributing the arbitration logic onto the various I/O modules, the LSI-11 Bus performs the same functions using only eight lines. The four interrupt request lines are represented collectively as: BUS BRx L (or BIRQx L on the LSI-11 Bus), where x is a number from 4 through 7. The four UNIBUS interrupt grant lines are similarly shown as: BUS BGx H (again, x is a number from 4 through 7). The LSI-11 Bus contains only one interrupt grant line: BIAKI L. Here are the arbitration lines:

- BUS NPR L (BDMR L)
- BUS NPG H (BDMGI L)
- BUS BRx L (BIRQx L)
- BUS BGx H (BIAKI L)
- BUS SACK L (BSACK L)

The **BUS NPR L** line is used by a peripheral to request the data section of the bus for a Direct Memory Access (DMA) transfer. (The acronym NPR stands for NonProcessor Request.)

The **BUS NPG H** line indicates to the peripheral that it may use the data section of the bus for a DMA transfer as soon as the current user is finished.

The **BUS BRx L** lines tell the processor that a peripheral would like to interrupt at level x. On the UNIBUS, the BUS BGx H lines indicate to the peripheral that it may interrupt the processor at level x, as soon as the data section becomes available. On the LSI-11 Bus, the BIAKI L line indicates that the processor acknowledges an interrupt from one of the four levels. Each device must examine the levels above it to ensure they are idle, before it can know that it 'owns' the grant. The processor interrupt follows immediately.

On the UNIBUS, the **BUS SACK L** line lets a device claim the bus after winning arbitration. The bus will not be given to any other device while BUS SACK L is asserted. On the LSI-11 Bus, BSACK L is used only for DMA transfers (not for interrupts).

DATA

It is in the data section where the two busses differ most. The UNIBUS provides a unique line for each signal; the LSI-11 BUS multiplexes addresses and data on the same lines. The UNIBUS uses voltage levels for its control signals; the LSI-11 Bus uses voltage transitions. For these (and other) reasons, we will discuss the UNIBUS and LSI-11 Bus separately.

UNIBUS

The UNIBUS data section consists of address and control lines, data lines, and synchronization lines. The address and control lines control the address at which a transfer occurs, and the type of transfer. The address lines are named BUS A00 L through BUS A17 L. The control lines include: BUS C0 L and BUS C1 L. Transfer types include:

- DATI—word read
- DATIP—word read with no restore
- DATO—word write
- DATOB—byte write
- (no name)—vector passing

The UNIBUS data lines carry 16 bits of data and two bits of parity error information. The UNIBUS data lines are named: BUS D00 L through BUS D15 L. The parity error information lines are named BUS PA L and BUS PB L.

The UNIBUS synchronization lines include:

- BUS MSYNC L
- BUS SSYN L
- BUS INTR L
- BUS BBSY L

The **BUS MSYNC L** line is asserted by the master to indicate that it has placed an address on the bus and has waited long enough to insure its validity. If the operation is a write, data is also placed on the bus.

The **BUS SSYN L** line is asserted by the slave to indicate that it recognizes an address as its own. If the operation is a read, the slave is now presenting valid data; if the operation is a write, the slave has stored the master's data.

The **BUS INTR L** line is asserted by the master to indicate that it has placed the address of an interrupt vector on the bus. The processor will respond with SSYN and will interrupt through that vector.

The **BUS BBSY L** line is asserted by the master to indicate that a master owns the data section of the UNIBUS. No other device should use the data section while BUS BBSY L is asserted.

LSI-11 Bus

The LSI-11 Bus data section consists of address/data lines, synchronization lines, and control lines. The address/data lines first pass an address, then pass one or more data words with associated parity error information. The address/data lines are named: BDAL21 L through

BDAL00 L. The transfer types are controlled by the synchronization and control lines, and include:

- DATI—word read
- DATO—word write
- DATOB—byte write
- DATIO—word read/write
- DATIOB—byte read/write
- DATBI—block read
- DATBO—block write

The LSI-11 Bus synchronization and control lines are:

- BSYNC L
- BDIN L
- BDOUT L
- BWTBT L
- BBS7 L
- BREF L
- BRPLY L

The **BSYNC L** line is asserted by the master when it has placed an address on the bus and has waited long enough to insure its validity.

The **BDIN L** line is asserted by the master during a DATI, DATIO(B), or DATBI cycle when it is ready to receive data from the slave on the BDAL lines (a read operation).

The **BDOUT L** line is asserted by the master during a DATO(B), DATIO(B), or DATBO cycle when it has placed data on the BDAL lines (a write operation) and has waited long enough to insure its validity.

The **BWTBT L** line serves two purposes. During the address cycle, it indicates that the data cycles will be writes to the slave (DATO, DATOB, DATBO). During the data cycle, it indicates that the write operation will be to a byte, rather than a word (DATOB).

The **BBS7 L** line serves two purposes. When the master gates an address onto the BDAL lines, it asserts BBS7 if the address of the slave is contained in the I/O page. When BBS7 is asserted, the slave decodes address lines 12 through 0 only. During DATBI cycles, the master asserts BBS7 L to indicate that it is a block mode master and that it has at least one more read cycle to perform. The bus master will reassert BDIN only if it has asserted BBS7 and if the slave has asserted BREF.

The use of the **BREF L** line depends on the memory refresh capability of a system. In systems with memories that do not perform their own refresh, the master asserts BREF when the current cycle is for memory refresh. In systems which support block mode, BREF is asserted by the slave to indicate that it can accept an additional BDIN or BDOUT signal.

The **BRPLY L** line is asserted by the slave that recognizes an address as its own. When responding to BDIN, the slave indicates that it will transmit data within an appropriate time. When responding to BDOUT, the slave indicates that it has received the incoming data.

MISCELLANEOUS

The miscellaneous lines perform a number of functions, including processor control, memory refresh, and timekeeping. The miscellaneous lines are listed below, with the lines for UNIBUS systems first, and the equivalent LSI-11 Bus lines following in parentheses. Note that the UNIBUS does not include these lines, they are connected separately to each backplane.

- HALT L (BHALT L)
- Not applicable (BREF L)
- LTC (BEVNT)
- BOOT ENB L (Not applicable)

The **HALT L** line causes the processor to halt at the completion of the current instruction. This line may be driven by the front panel, or the console terminal interface.

The **LTC** line provides a realtime clock input for the system. It pulses with each cycle of the line current.

The **BOOT ENB L** line controls the processor action when power is restored following a power failure. If unasserted, the battery backup unit has maintained the contents of memory, and the processor restarts through the vector at 24. If asserted, memory contents have been lost, and the processor reboots.

BUS TIMING

For timing diagrams of UNIBUS and LSI-11 Bus cycles, refer to Appendices D and E respectively. Further details of the bus timing cycles may be found in the technical documentation for the various processors.

BUS ERRORS

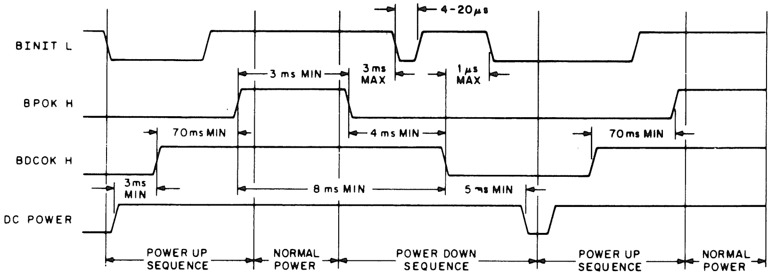
On any given bus cycle either of two error conditions may result. It is the current master's responsibility to handle these errors.

Timeout Errors

The bus architecture is asynchronous; a cycle terminates when the slave responds with SSYN. If the master places an address on the bus which does not correspond to any slave, no SSYN will ever result. It is the master's responsibility to note that no slave has responded, and to terminate the failed transfer. Normally, a master will wait 7—25 μs for a slave's response. How the master handles bus timeout is up to the master. Processors initiate a trap through the vector at 4; most I/O devices set a bit called NXM (non-existant memory) and then stop.

Parity Errors

If the slave (usually memory) detects an internal parity error upon a read operation, it will use the parity-error information lines to pass this information back to the current master. Again, action is the master's responsibility; processors trap through the vector at 114.



NOTE
Once a power down sequence is started, it must be completed before a power-up sequence is started

Figure 10-1 Power-Up/Power-Down Timing

APPENDIX A

ASSIGNMENT OF BUS ADDRESSES AND VECTORS

Throughout this appendix, both LSI-11 Bus and UNIBUS devices are listed. LSI-11 Bus devices may be distinguished by a three-letter prefix ending in the letter "V" (e.g. DLV11).

I/O PAGE DEVICE ADDRESS

Fixed CSR Address Assignments

Device	Address	Size	Number	
AA11	776750	8	1	(first unit)
AA11	776400	8	4	(extra units)
AAV11	770440	4	1	
AD01	776770	4	1	
ADF11	770460	8	1	
ADV11-A	770400	2	1	
AFC11	772570	4	1	
AR11	770400	8	1	
BDV11-CSR	777520	3	1	
BDV11-LTC	777546	1	1	
BDV11-ROM	773000	256	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/YC	773000	256	1	
CD11	777160	4	1	
CM11	777160	4	1	
CMR11	764070	4	1	(CSS device)
CR11	777160	4	1	
CSR11	764000	4	1	(CSS device)
CSS/User	764000	1024	1	
DC11	774000	4	32	
DC14-D	777360	8	1	
Diagnostics	760000	4	1	
DL/DLV11-A/B	777560	4	1	(console)
DL/DLV11-A/B	776500	4	16	
DL11-C/D/E	775610	4	31	
DLV11-E	775610	4	31	
DLV11-F	776500	4	16	
DLV11-J	776500	16	4	
DL11-W(LTC)	777546	1	1	(line clock, first unit only)
DL11-W	777560	4	1	(console)

Fixed CSR Address Assignments (Cont.)

Device	Address	Size	Number	
DL11-W	776500	4	16	
DM11	775000	4	16	
DM11-BB/BA	770500	4	16	(modem control for DM11)
DN11-AA	775200	4	16	
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	
DRV11	767750	4	−3	(assigned backwards)
DRV11-B	772410	4	3	
DRV11-J	764120	8	−3	(assigned backwards)
DS11	775400	67	1	
DT07	777420	1	8	
DV11	775000	16	4	
DX11	776200	16	2	
Floating CSRs	760010	1020	1	
FP11	772160	8	1	
GT40	772000	4	4	
IBV11	760150	2	1	
ICR/ICS11	771000	256	1	
IEX	764130	8	1	(CSS device)
IP11/IP300	771000	128	2	
KE11	777300	8	2	
KG11	770700	4	8	
KL11	776500	4	16	
KL11	777560	4	1	(console)
KPV11(LTC)	777546	1	1	
KT11	772200	64	1	
KT11-SR3	772516	1	1	
KU116-AA	777540	3	1	
KW11-L	777546	1	1	
KW11-P	772540	4	1	
KW11-W	772400	4	1	
KWV11-A	770420	2	1	
LAV/LPV11	777514	2	1	
LP/LS/LV11	777514	2	1	(LP0)

Fixed CSR Address Assignments (Cont.)

Device	**Address**	**Size**	**Number**	
LP/LS/LV11	764004	2	1	(LP1)
LP/LS/LV11	764014	2	1	(LP2)
LP/LS/LV11	764024	2	1	(LP3)
LP/LS/LV11	764034	2	1	(LP4)
LP/LS/LV11	764044	2	1	(LP5)
LP/LS/LV11	764054	2	1	(LP6)
LP/LS/LV11	764064	2	1	(LP7)
LP20	775400	32	2	
LPA11-K	770460	8	1	
LPS11	770400	16	1	
M792	773000	32	8	
M7930	777510	4	1	
M9301-XX	765000	256	1	
M9301-XX	773000	256	1	
ML11	776400	22	1	(RH70/RH11)
MM11	772100	1	16	
MR11-DB	773100	64	1	
MRV11-11	77300	256	1	
MS11/MSV11	772100	1	16	
NCV11	772760	8	1	
RB730	775606	1	1	
RDRX	774340	8	2	
OST	772500	6	1	
PA611_readers	772600	4	8	(2 per PA611)
PA611_punches	772700	4	8	(2 per PA611)
PC11/PCV11	777550	4	1	
PCL11	764200	16	4	(CSS Device)
PDP11	777570	68	1	
PR11	777550	4	1	
QNA	774440	8	2	
RC11	777440	8	1	
Reserved	770100	32	1	
Reserved	770440	8	1	
Reserved	772154	2	1	
Reserved	772514	1	1	
Reserved	772550	8	1	
Reserved	775606	1	1	
Reserved	777000	56	1	
Reserved	777200	32	1	
Reserved	777526	1	1	
REV11	773000	256	1	
REV11	765000	256	1	

Fixed CSR Address Assignments (Cont.)

Device	Address	Size	Number	
RF11	777460	8	1	
RH70/11_alt	776300	32	1	(Alternate RS/RP/RM/TJ)
RK611/RK711	777440	16	1	
RK11/RKV11	777400	8	1	
RL11/RLV11	774400	4	1	
RLV12	774400	8	1	
RM03/04/05	776700	22	1	(RH70/RH11)
RP04/05/06	776700	22	1	(RH70/RH11)
RP11	776700	16	1	
RS04	772040	16	1	(RH70/RH11)
RX11/211	777170	4	1	
RXV11/21	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	
TU78	775400	32	1	(RH70/RH11)
TU81	774500	2	1	
UDA	772150	2	1	(All UDA class disks)
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	

INTERRUPT AND TRAP VECTORS

Fixed Vector Address Assignments

Device	**Address**	**Size**	
AA11	140	4	
AD01	130	2	
ADV11	400	4	
AFC11	134	2	
CD11	230	2	
CM11	230	2	
CMR11	170	2	(CSS device)
Console	060	4	
CR11	230	2	
CSR11	270	2	(CSS device)
DIP11	260	2	
DL11(1)	060	4	
DR11-B, DRV11-B	124	2	
Floating Vectors	300	*	
FPP/FIS exception	244	2	
IBV11	420	4	
ICS/ICR11, IP11/IP300	234	2	
KT11 Error	250	2	
KW11-A	440	4	
KW11-L	100	2	
KW11-P	104	2	
KWV11	440	4	
LAV11/LPV11	200	2	
LP/LS/LV11 (#0)	200	2	
LP/LS/LV11 (#1)	170	2	
LP/LS/LV11 (#2)	174	2	
LP/LS/LV11 (#3)	270	2	
LP/LS/LV11 (#4)	274	2	
LP20(1)	200	2	
LP20(2)	210	2	
Memory System errors	114	2	
PC11	070	4	
PDP11-Reserved	000	2	
PDP11-CPU Errors	004	2	
PDP11-Reserved Instructions	010	2	
PDP11-Breakpoint/Trace traps	014	2	
PDP11-IOT Trap	020	2	
PDP11-Power Fail	024	2	
PDP11-EMT Trap	030	2	
PDP11-TRAP Trap	034	2	
PDP11-PIRQ	240	2	
RB730	250	2	

Fixed Vector Address Assignments

Device	Address	Size	
RDRX#0	130	2	
RDRX#1	134	2	
RC11	210	2	
RF11	204	2	
RK611/RK711	210	2	
RK11/RKV11	220	2	
RL11/RLV11	160	2	
RLV12	160	2	
Alternate RS/RP/RM/TJ	150	2	(RH70/RH11)
RS03/04 (RH11/RH70)	204	2	
RM02/03/05 (RH11/RH70)	254	2	
RP04/5/6 (RH11/RH70)	254	2	
RP11	254	2	
Reserved for System Software	110	1	
Reserved for System Software	040	8	
RSTS/E (crash-dump)	144	2	
RSTS/E (statistics ptr)	234	1	
RX11/211, RXV11/21	264	2	
TA11	260	2	
TC11	214	2	
TM11	224	2	
TS11	224	2	
TU16/45, TE16, TU77 (RH11/RH70)	224	2	
TU78 (RH11/RH70)	260	2	
TU81	260	2	
UDA	154	2	
UDC11	234	2	
UNUSED—Reserved for Digital	164	2	
USER/CSS RESERVED	170	4	
USER/CSS RESERVED	270	4	
XY11	120	2	

NOTES:

1. ADV11, KWV11, and IBV11 use non-standard vectors in floating vector space. These devices may not be autoconfigured in standard systems.
2. RSTS does not support the AA11, UDC11, and ICS/ICR11 which require vectors by RSTS for other purposes.
3. User/CSS Reserved vectors are also used for additional line printers.

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	Device	Size	Octal Modulus	
1	DC11	4	10	
1	TU58	4	10	(See Note)
2	KL11	4	10	
2	DL11-A	4	10	
2	DL11-B	4	10	
2	DLV11-J	16	10	
2	DLV11, DLV11-F	4	10	
3	DP11	4	10	
4	DM11-A	4	10	
5	DN11	2	4	
6	DM11-BB/BA	2	4	
7	DH11 modem control	2	4	
8	DR11-A, DRV11-B	4	10	
9	DR11-C, DRV11	4	10	
10	PA611 (reader + punch)	8	10	
11	LPD11	4	10	
12	DT07	4	10	
13	DX11	4	10	
14	DL11-C	4	10	
14	DL11-D	4	10	
14	DL11-E/DLV11-E	4	10	
15	DJ11	4	10	
16	DH11	4	10	
17	GT40	8	10	
17	VSV11	8	10	
18	LPS11	12	10	

Rank	Device	Size	Octal Modulus	
19	DQ11	4	10	
20	KW11-W, KWV11	4	10	
21	DU11, DUV11	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/DZS11/DZV11, DZ32	4	10	(DZ11 before DZ32)
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	TS11	2	4	(after the first)
36	LPA11-K	4	10	
37	IP11/IP300	2	4	(after the first)
38	KW11-C	4	10	
39	RX11/RX211 RXV11/RXV21	2	4	(after the first) (RX11 before RX211)
40	DR11-W	2	4	
41	DR11-B	2	4	(after the first)
42	DMP11	4	10	
43	DPV11	4	10	
44	ML11	2	4	(MASSBUS device)
45	ISB11	4	10	
46	DMV11	4	10	
47	DEUNA	2	4	
48	UDA50	2	4	(after the first)
49	DMF32	16	4	
50	KMS11	6	10	
51	PCL11-B	4	10	
52	VS100	2	4	
53	TU81	2	4	(after the first)

NOTES:

1. There is no standard configuration for systems with both DC11 and TU58.
2. A KL11 or DL11 used as the console uses a fixed vector.

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:

Floating CSR Address Assignments

Rank	Device	Size	Octal Modulus	
1	DJ11	4	10	
2	DH11	8	20	
3	DQ11	4	10	
4	DU11, DUV11	4	10	
5	DUP11	4	10	
6	LK11A	4	10	
7	DMC11/DMR11	4	10	(DMC before DMR)
8	DZ11/DZV11, DZS11, DZ32	4	10	(DZ11 before DZ32)
9	KMC11	4	10	
10	LPP11	4	10	
11	VMV21	4	10	
12	VMV31	8	20	
13	DWR70	4	10	
14	RL11, RLV11	4	10	(after first)
15	LPA11-K	8	20	(after first)
16	KW11-C	4	10	
17	Reserved	4	10	
18	RX11/RX211 RXV11/RXV21	4	10	(after first) (RX11 before RX211)
19	DR11-W	4	10	
20	DR11-B	4	10	(after second)
21	DMP11	4	10	
22	DPV11	4	10	
23	ISB11	4	10	
24	DMV11	8	20	
25	DEUNA	4	10	
26	UDA50	2	4	(after first)
27	DMF32	16	40	
28	KMS11	6	20	
29	VS100	8	20	
30	TU81	2	4	(after first)

NOTES:

1. DZ11-E and DZ11-F are treated as two DZ11s.

DEVICE ADDRESSES

776 000 760 006	Diagnostics	
760 010	(Start of floating addresses)	
760 150 760 152	IBV11	
763 776	(Top of floating addresses)	
764 000	TR79	
764 004 764 066	LP/LS/LV11 (Units 1-7)	Customer
764 070 764 076	CMR11	
764 120 764 176	DRV11-J	
764 200 764 376	PCL11	
765 000 765 776	M9301	
767 600 767 776	DR11-A/C	
770 000 770 076	Testers	
770 100 770 176	Reserved	

770 700	KG11	#1
770 776		#8
771 000	UDC Functional I/O Units	ICR/ICS11
771 776		IP11/IP300
771 774	UDC11	ICR/ICS11
771 776		IP11/IP300
772 000	GT40 (#1-#4)	
	VSV11 (#1-#4)	
772 736	VT48	
772 040	RS04	
772 076		
772 100	UNIBUS Memory Parity	MM11-LP #1
772 136		MS11-LP #16
772 140	UNIBUS Tester	
772 146		
772 150	Reserved	
772 156		
772 160	FP11 Registers	
772 176		
772 200	Supervisor Instruction Descriptor PDR, reg 0-7	
772 216		
772 220	Supervisor Data Descriptor PDR, reg 0-7	
772 236		

770 200	UNIBUS Map	
770 376		
770 400	AR11	LPS11
770 416	ADV11-A	
770 420	KWV11-A	
770 422		
770 436		
770 440	AAV11	Reserved
770 446		
770 456		
770 460	ADF11/LPA11-K	
770 476		
770 500	DM11-BB/BA	#1
770 676		#16
770 700	KG11	#1
770 776		#8
771 000	UDC Functional I/O Units	ICR/ICS11
771 776		IP11/IP300
771 774	UDC11	ICR/ICS11
771 776		IP11/IP300
772 000	GT40 (#1-#4)	
	VSV11 (#1-#4)	
772 036	VT48	

772 040 772 076	RS04
772 100 772 136	Memory Parity Registers
772 140 772 146	UNIBUS Tester
772 150 772 152	UDA
772 154 772 156	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 PAR, 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/W(#1)	DRV11-B
772 420 772 426	Reserved	
772 430 772 436	DR11-B/W(#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	KW11-P				
772 546					
772 550	Reserved				
772 566					
772 570	AFC11				
772 576					
772 600	PA611 Typeset Readers				VTV01
772 676					
772 700		PA611 Typeset Punches			
772 760	NCV11				
772 776					
773 000	BM792-YA	BM873-YA	BDV11 ROM BM873-YB BM873-YC M792 M9301/9312-XX MRV11-11 REV11		
773 076					
773 100	MR11-DB				
773 276					
773 376					
773 476					
773 776					
774 000	DC11,			#1	
774 376				#32	
774 400	RL11/ RLV11	RLV12	DP11	#1	
774 406					
774 416					
774 776				#32	

775 000	DM11,	#1	DV11, #1-#4
775 176		#16	
775 200	DN11-AA/DN11-DA	#1	
775 376		#16	
775 400	TU78	LP20	DS11
775 476			
775 576			
775 604			
775 606	Reserved		
775 610	DL11-C, -D, -E DLV11-E	#1	
776 176		#31	
776 200	DX11		
776 276			
776 300	alternate RH70/RH11		
776 376			
776 400	ML11	AA11,	#2
776 452			
776 476			#5
776 500	TU58	KL11, DL11-A, -B, -W DLV11-A, -B, -F, -J	#1
776 676			#16

776 700 776 736	RP11	RM03/04/05, RP04/05/06/07

776 750
776 752
776 766
	AA11, #1	

776 770 776 776	AD01	
777 000 777 156	Reserved	
777 160 777 166	CD11, CM11 CR11	
777 170 777 176	RX11/RX211 RXV11/RXV21	
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/RKV11	

777 420	DT07	
777 436		
777 440	RC11	RK611/RK711
777 456		
777 460	RF11	
777 476		
777 500	TA11/DIP11-A	
777 506		
777 510	Reserved	
777 512		
777 514	LP/LS/LV11 M7930 LAV/LPV11	
777 516		
777 520	BDV11-CSR	Reserved
777 524		
777 526		
777 530	XY11	
777 536		
770 540	KU116-AA	
777 544		
777 546	BDV11/DL11-W/KPV11/KW11-L, line clock	
777 550	PC11/PCV11/PR11	
777 556		

777 560	KL11 DL11-A/B/W DLV11-A/B/J	Console Terminal Interfaces
777 566		
777 570	Console Switch & Display Register	
777 572	Memory Mgt. reg	(MMR0)
777 574		(MMR1)
777 576		(MMR2)
777 600	User Instruction PDR, reg 0-7	
777 616		
777 620	User Data PDR, reg 0-7	
777 636		
777 640	User Instruction PAR, reg 0-7	
777 656		
777 660	User Data PAR, reg 0-7	
777 676		
777 700	General registers, Set 0	R0
777 701		R1
777 702		R2
777 703		R3
777 704		R4
777 705		R5
777 706	Kernel	R6(SP)
777 707		R7(PC)
777 710	General registers Set 1	R0
777 711		R1
777 712		R2
777 713		R3
777 714		R4
777 715		R5
777 716	Supervisor	R6(SP)
777 717	User	R6(SP)

777 720	Reserved		
777 726			
777 730	Memory and Cache Control		
777 756			
777 760	Lower Size	System Size	(PDP-11/70)
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 and LSI-11 Bus addresses are reserved by Digital.

APPENDIX B

PDP-11 FAMILY DIFFERENCES TABLE

The table that follows illustrates the issues involved in software migration between different members of the PDP-11 family. Each member of the family has some slight differences in the way instructions are executed. Any program developed using PDP-11 operating systems with higher level languages will migrate with very little difficulty. However, some applications written in assembly language may have to be modified slightly.

Since the instruction set for all F-11 based processors is identical, the 23/24 column refers to the PDP-11/23 PLUS, the PDP-11/24, the LSI-11/23, the MICRO/PDP-11, and the F-11 chip itself.

The LSI-11 column includes the LSI-11/2.

The T-11 column also refers to the FALCON SBC-11/21.

The VAX column refers to the PDP-11 Compatibility Mode available on VAX-11 processors.

PROCESSORS

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
1. OPR %R, (R) +; OPR %R, – (R) using the same register as both source and destination: contents of R are incremented (decremented) by 2 before being used as the source operand.	X						X	X			X	X	X	
OPR %R, (R) +; OPR %R, – (R) using the same register as both register and destination: initial contents of R are used as the source operand.		X	X	X	X	X			X	X				X
2. OPR %R, @ (R) +; OPR %R, @ – (R) using the same register as both source and destination: contents of R are incremented (decremented) by 2 before being used as the source operand.	X						X	X			X	X	X	
OPR %R, @ (R) +; OPR %R, @ – (R) using the same register as both source and destination: initial contents of R are used as the source operand.		X	X	X	X	X			X	X				X
3. OPR PC, X (R); OPR PC, @ X (R); OPR PC, @ A; OPR PC, A: location A will contain the PC of OPR +4.	X						X	X			X	X	X	
OPR PC, X (R); OPR PC, @ X (R), OPR PC, A; OPR PC, @ A: location A will contain the PC of OPR +2.		X	X	X	X	X			X	X				X
4. JMP (R) + or JSR reg, (R) +: contents of R are incremented by 2, then used as the new PC address.						X	X							
JMP (R) + or JSR reg, (R) +: initial contents of R are used as the new PC.	X	X	X	X	X			X	X	X	X	X	X	X

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
5. JMP %R or JSR reg, %R traps to 10 (illegal instruction).	X		X	X	X	X	X	X			X		X	NA
JMP %R or JSR reg, %R traps to 4 (illegal instruction).		X							X	X		X		NA
6. SWAB does *not* change V.							X							
SWAB clears V.	X	X	X	X	X	X		X	X	X	X	X	X	X
7. Register addresses (177700–177717) are valid program addresses when used by CPU.						X							—[1]	—[1]
Register addresses (177700–177717) time out when used as a program address by the CPU. Can be addressed under console operation.		X	X	X			X	X	X	X	X			NA
Register addresses (177700–177717) time out when used as an address by CPU or console.	X				X							X		
8. Basic instructions noted in PDP-11 processor handbook.	X	X	X	X	X	X	X	X	X	X	X	X	X	X
SOB, MARK, RTT, SXT instructions*	X	X		X	X			X	X	X	X	X	X	—[2]
ASH, ASHC, DIV, MUL, XOR	X	X		X	X			X	X	X	X	X		X
Floating Point instructions in base machine.											X	X		
MFPT Instruction.	X	X										X		
The external option KE11-A provides MUL, DIV, SHIFT operation in the same data format.						X	X							

* RTT instruction is available in 11/04 but is different than other implementations.

[1] Register addresses (177700–177717) are handled as regular memory addresses in the I/O page.
[2] All but MARK.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
The KE11-E (Expansion Instruction Set) provides the instructions MUL, DIV, ASH, and ASHC. These new instructions are 11/45 compatible.								X						
The KE11-F (Floating Instruction Set) adds unique stack ordered oriented point instructions: FADD, FSUB, FMUL, FDIV.								X						
The KEV-11 adds EIS/FIS instructions					X									
MFP, MTP instructions	X	X		X				X		X	X	X		
SPL Instruction		X							X	X		X		
CSM Instruction		X										X		
9. Power fail during RESET instruction is not recognized until after the instruction is finished (70 milliseconds). RESET instruction consists of 70 millisecond pause with INIT occurring during first 20 milliseconds.							X	X			X			
Power fail immediately ends the RESET instruction and traps if an INIT is in progress. A minimum INIT of 1 microsecond occurs if instruction aborted. PDP11-04/34/44 are similar with no minimum INIT time.		X	X	X					X	X				
Power fail acts the same as 11/45 (22 milliseconds with about 300 nanoseconds minimum). Power fail during RESET fetch is fatal with no power down sequence.						X								

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
RESET instruction consists of 10 microseconds of INIT followed by a 90 microsecond pause. Reset instruction consists of a minimum 8.4 microseconds followed by a minimum 100 nanosecond pause. Power fail not recognized until the instruction completes.	X				X							X		
10. No RTT instruction						X	X							
If RTT sets the "T" bit, the "T" bit trap occurs after the instruction following RTT.	X	X	X	X	X			X	X	X	X	X	X	X
11. If RTI sets "T" bit, "T" bit trap is acknowledged after instruction following RTI.						X	X							X
If RTI sets "T" bit, "T" bit trap is acknowledged immediately following RTI.	X	X	X	X	X			X	X	X	X	X	X	
12. If an interrupt occurs during an instruction that has the "T" bit set, the "T" bit trap is acknowledged before the interrupt.	X	X	X	X	X	X	X	X			X	X	X	NA[1]
If an interrupt occurs during an instruction and the "T" bit is set, the interrupt is acknowledged before "T" bit trap.									X	X				NA
13. "T" bit trap will sequence out of WAIT instruction.	X	X	X	X		X	X	X			X	X	X	NA
"T" bit trap will not sequence out of WAIT instruction. Waits until an interrupt.					X				X	X				

[1] Interrupts not visible to VAX compatibility mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
14. Explicit reference (direct access) to PS can load "T" bit. Console can also load "T" bit.			X			X	X							
Only implicit references (RTI, RTT, traps and interrupts) can load "T" bit. Console cannot load "T" bit.	X	X		X	X			X	X	X	X	X	X	X
15. Odd address/non-existent references using the SP cause a HALT. This is a case of double bus error with the second error occurring in the trap servicing the first error. Odd address trap not implemented in LSI-11, 11/23 or 11/24.		X	X	X	X	X	X							
Odd address/non-existent references using the stack pointer cause a fatal trap. On bus error in trap service, new stack created at 0/2.	X							X	X	X	X	X	—[1]	—[2]
16. The first instruction in an interrupt routine will not be executed if another interrupt occurs at a higher priority level than assumed by the first interrupt.	X	X	X	X	X	X		X	X	X	X	X	X	X
The first interrupt in an interrupt service is guaranteed to be executed.							X							
17. Single general purpose register set implemented.	X	X	X	X	X	X	X	X			X		X	X
Dual general purpose register set implemented.									X	X		X		

[1] Odd address/non-existent references using SP do not trap.
[2] Odd address aborts to native mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
18. PSW address, 177776, not implemented; must use instructions MTPS (move to PS) and MFPS (move from PS).					X								X	—[3]
PSW address implemented, MTPS and MFPS not implemented.		X	X			X	X	X	X	X	X			
PSW address and MTPS and MFPS implemented.	X			X								X		
19. Only one interrupt level (BR4) exists.					X									
Four interrupt levels exist.	X	X	X	X		X	X	X	X	X	X	X	X	NA
20. Stack overflow not implemented.					X								X	X
Some sort of stack overflow implemented.	X	X	X	X		X	X	X	X	X	X	X		
21. Odd address trap not implemented.	X				X								X	
Odd address trap implemented.		X	X	X		X	X	X	X	X	X	X		X
22. FMUL and FDIV instructions implicity use R6 (one push and pop); hence R6 must be set up correctly.					X									
FMUL and FDIV instructions do not implicitly use R6.								X						NA
23. Due to their execution time, EIS instructions can abort because of a device interrupt.					X									X
EIS instructions do not abort because of a device interrupt.	X	X		X				X	X	X	X	X		NA
24. Due to their execution time, FIS instructions can abort because of a device interrupt.					X			X						NA

[3] Can reference PSW only from native mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
25. Due to their execution time, FP11 instructions can abort because of a device interrupt*	X													
FP11 instructions do not abort because of a device interrupt.		X		X					X	X	X	X		NA
26. EIS instructions do a DATIP and DATO bus sequence when fetching source operand.					X									
EIS instructions do a DATI bus sequence when fetching source operand.	X	X		X				X	X	X	X	X		NA
27. MOV instruction does just a DATO bus sequence for the last memory cycle.	X	X		X	X			X	X	X	X	X		—[1]
MOV instruction does a DATIP and DATO bus sequence for the last memory cycle.			X			X	X						—[2]	
28. If PC contains non-existent memory and a bus error occurs, PC will have been incremented.	X	X	X	X	X	X	X		X	X		X		
If PC contains non-existent memory address and a bus error occurs, PC will be unchanged.								X					—[3]	X
29. If register contains non-existent memory address in mode 2 and a bus error occurs, register will be incremented.	X				X	X	X	X	X	X		X	—[3]	
Same as above but register is unchanged.		X	X	X										

* Integral floating point assumed on 11/23 and 11/24; FP11E assumed for 11/60.

[1] Implementation dependent.

[2] MOV instruction does a DATI and a DATO bus sequence for last memory cycle.

[3] Does not support bus errors.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
30. If register contains an odd value in mode 2 and a bus error occurs, register will be incremented.	X				X			X	X	X		X	—[3]	—[4]
If register contains an odd value in mode 2 and a bus error occurs, register will be unchanged.		X	X	X		X	X							
31. Condition codes restored to original values after FIS interrupt abort (EIS doesn't abort on 35/40).								X						
Condition codes that are restored after EIS/FIS interrupt abort are indeterminate.					X									NA
32. Opcodes 075040 through 075377 unconditionally trap to 10 as reserved opcodes.	X	X	X	X		X	X	X	X	X	X	X	X	—[1]
If KEV-11 option is present, opcodes 75040 through 07533 perform a memory read using the register specified by the low order 3 bits as a pointer. If the register contents are a non-existent address, a trap to 4 occurs. If the register contents are an existent address, a trap to 10 occurs.					X									
33. Opcodes 210 thru 217 trap to 10 as reserved instructions.	X	X	X	X		X	X	X	X	X	X	X	X	—[1]
Opcodes 210 thru 217 are used as a maintenance instruction.					X									

[3] Does not support bus errors.

[4] Unpredictable.

[1] Traps to native mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
34. Opcodes 75040 thru 75777 trap to 10 as reserved instructions.	X	X	X	X		X	X	X	X	X	X	X	X	—[1]
If KEV-11 options is present, opcodes 75040 thru 75577 can be used as escapes to user microcode. If no user microcode exists, a trap to 10 occurs.					X									
35. Opcodes 170000 thru 177777 trap to 10 as reserved instructions.			X			X	X	X					X	—[1]
Opcodes 170000 thru 177777 are implemented as floating point instructions.	X	X		X					X	X	X	X		
Opcodes 170000 thru 177777 can be used as escapes to user microcode. If no user microcode exists, a trap to 10 occurs.					X									
Opcode 076600 used for maintenance.											X			
36. CLR and SXT do just a DATO sequence for the last bus cycle.	X											X		—[1]
CLR and SXT do DATIP-DATO sequence for the last bus cycle.		X	X	X	X	X	X	X	X	X	X		—[2]	
37. MEM MGT maintenance mode MMR0 bit 8 is implemented.		X		X				X	X	X	X			
MEM MGT maintenance mode MMR0 bit 8 is not implemented.	X											X		NA
38. PS<15:12>, non-kernel mode, non-kernel stack pointer and MTPx and MFPx instructions exist even when MEM MGT is not configured.	X	X							X	X	X	X		

[1] Traps to native mode.

[1] Unpredictable.
[2] CLR and SXT do DATI-DATO.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
PS<15:12>, non-kernel mode, non-kernel stack pointer, and MTPx and MFPx instructions exist only when MEM MGT is configured.								X						NA
39. Current mode PS bits <15:14> set to 01 or 10 will cause a MEM MGT trap upon any memory reference.	X			X				X				X		NA
Current mode PS bits <15:14> set to 10 will be treated as kernel mode (00) and not cause a MEM MGT trap.	X													
Current mode PS bits <15:14> set to 10 will cause a MEM MGT trap upon any memory reference.		X							X	X				
40. MTPS in user mode will cause MEM MGT trap if PS address 177776 not mapped. If mapped, PS <7:5> and <3:0> affected.				X										NA
MTPS in non-user mode will not cause MEM MGT trap and will only affect PS <3:0> regardless of whether PS address 177776 is mapped.	X											X		
41. MFPS in user mode will cause MEM MGT if PS address 177776 not mapped. If mapped, PS <7:0> are accessed.				X										NA
MTPS in user mode will not trap regardless of whether PS address 177776 is mapped.	X											X		

[1] Unpredictable.

[2] CLR and SXT do DATI-DATO.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
42. Programs cannot execute out of internal processor registers.												X		
Programs can execute out of internal processor registers.	X	X		X				X	X	X	X			
43. A HALT instruction in user or supervisor mode will trap thru location 4.		X							X	X		X		
A HALT instruction in user or supervisor mode will trap thru location 10.	X			X				X			X		—[1]	—[2]
44. PDR bit <0> implemented.									X	X				
PDR bit <0> not implemented.	X	X		X				X			X	X	X	X
45. PDR bit <7> (any access) implemented.									X	X				
PDR bit <7> (any access) not implemented.	X	X		X				X			X	X	X	X
46. Full PAR <15:0> implemented.	X	X								X		X		
Only PAR <11:0> implemented.				X				X	X		X		X	X
47. MMR0<12>—trap-memory management—implemented.									X	X				
MMR0<12> not implemented.	X	X		X				X			X	X	X	X
48. MMR3<2:0>—D space enable—implemented.		X							X	X		X		
MMR3<2:0> not implemented.	X			X				X			X		X	X
49. MMR3<5:4>—IOMAP, 22-bit mapping enabled—implemented.	X	X								X		X		
MMR3<5:4> not implemented.				X				X	X		X		X	X

[1] HALT pushes PC & PSW to stack, loads PS with 340 and PC with <powerup address> + 40.
[2] Traps to native mode.

ITEM	23/24	44	04	34	LSI11	05/10	15/20	35/40	45	70	60	J-11	T-11	VAX
50. MMR3<3>—CSM enable—implemented.		X										X		
MMR3<3> not implemented.	X			X				X	X	X	X		X	X
51. MMR2 tracks instruction fetches and interrupt vectors.									X	X				
MMR2 tracks only instruction fetches.	X	X		X				X			X	X	NA	NA
52. MFPx %6, MTPx when PS<13:12>= 10 gives unpredictable results.	X	X		X				X	X	X	X			
MTPx %6, MTPx %6 when PS<13:12>= 10 uses user stack pointer.												X	NA	NA

[1] HALT pushes PC & PSW to stack, loads PS with 340 and PC with <powerup address> + 40.

[2] Traps to native mode.

APPENDIX C

FLOATING POINT INSTRUCTION SET FIS (LSI-11, LSI-11/2, AND PDP-11/03)

INTRODUCTION

The Floating Point Instruction Set (FIS) option consists of four instructions: Floating Add (FADD), Floating Subtract (FSUB), Floating Multiply (FMUL), and Floating Divide (FDIV). These instructions operate on single-precision floating formats, and are available on the LSI-11, LSI-11/2, and PDP-11/03 only. The KEV11 is the EIS/FIS option for the LSI-11, LSI-11/2, and PDP-11/03.

KEV11 OPTION

FIS Instruction Set

The following floating point instruction opcodes do not conflict with any other instructions and are not compatible with the FP-11 Instruction Set.

Mnemonic	Instruction	Opcode
FADD	Floating Add	07500R
FSUB	Floating Subtract	07501R
FMUL	Floating Multiply	07502R
FDIV	Floating Divide	07503R

The operand format for the FIS is identical to that for FP11 single-precision numbers. This format is explained in Chapter 3, in the FLOATING-POINT DATA FORMAT section.

Registers

There are no preassigned registers for the floating point option. A general-purpose register is used as a pointer to specify a stack address. The contents of the register are used to locate the operands and answer for the floating point operations as follows:

R = high B argument address
R+2 = low B argument address
R+4 = high A argument address
R+6 = low A argument address

After the floating point operation, the answer is stored on the stack as follows:

R+4 = address for high part of answer
R+6 = address for low part of answer

where R is the original contents of the general register used.

After execution of the instruction, the general registers point to the high answer, i.e., R is incremented by 4.

Condition Codes

Condition codes are set or cleared as shown in the instruction descriptions, in the next part of this section. If a trap occurs as a function of a floating point instruction, the condition codes are reinterpreted as follows:

V = 1, if an error occurs
N = 1, if underflow or divide by zero
C = 1, if divide by zero
Z = 0

	V	N	C	Z
Overflow	1	0	0	0
Underflow	1	1	0	0
Divide by 0	1	1	1	0

Traps

Traps occur through vector address 244. Traps occur because of overflow, underflow, or divide by zero conditions.

Following a trap, the general register is unaltered, as are (R), (R + 2), (R + 4), and (R + 6).

The condition codes in the PS that caused a trap to 244 are set in the PS that was used while the FIS instruction was being executed. Following the trap, this PS is pushed onto the stack. The stack must be examined following a trap to retrieve the PS and determine the reason for the trap.

Interrupts

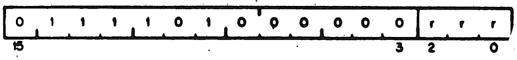
A floating point instruction is aborted if an interrupt request is issued before the instruction is near completion. The program counter points to the aborted floating point instruction so that the interrupt looks transparent.

FIS Instructions

Assembler format is: OPR R

(R) denotes contents of memory location whose address is in R.

FADD
Floating Add 07500R



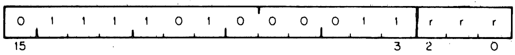
Operation: [(R + 4), (R + 6)] ◂— [(R + 4), (R + 6)] + [(R), (R + 2)]

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: cleared

Description: Adds the A argument to the B argument and stores the result in the A argument position on the stack. General register R is used as the stack pointer for the operation.

A ◂— A + B

FDIV
Floating Divide 07503R



Operation: [(R + 4), (R + 6)] ◂— [(R + 4), (R + 6)] / [(R), (R + 2)]

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: cleared

Description: Divides the A argument by the B argument and stores the result in the A argument position on the stack. If the divisor (B argument) is equal to zero, the stack is left untouched.

A ◂— A/B

Note: The LSI-11 processors push one word onto the stack during execution of the FMUL and FDIV instructions and pop the word from the stack when completed. Thus, the SP (R6) must point to a read/write memory location; otherwise, a bus error (time-out) occurs.

FMUL
Floating Multiply

07502R

0	1	1	1	1	0	1	0	0	0	0	1	0	r	r	r
15												3	2		0

Operation: $[(R+4), (R+6)] \leftarrow [(R+4), (R+6)] \times [(R), (R+2)]$

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: cleared

Description: Multiplies the A argument by the B argument and stores the result in the A argument position on the stack.

$A \leftarrow A \times B$

(refer to note in FDIV)

FSUB
Floating Subtract

07501R

0	1	1	1	1	0	1	0	0	0	0	0	1	r	r	r
15												3	2		0

Operation: $[(R+4), (R+6)] \leftarrow [(R+4), (R+6)] - [(R), (R+2)]$

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: cleared

Description: Subtracts the B argument from the A argument and stores the result in the A argument position on the stack.

$A \leftarrow A - B$

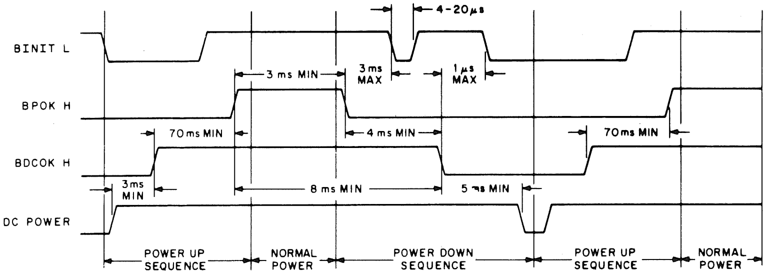
APPENDIX D

UNIBUS TIMING DIAGRAMS

In the timing diagrams of this book, signals are drawn as: *up* while asserted, *down* while de-asserted. The actual voltages on the bus are usually the complement.

POWER UP/DOWN

The power up, power down timing sequence is shown in the diagram below. This diagram is labeled for the LSI-11 Bus; for the UNIBUS, BDCOK H becomes BUS DCLO L, BPOK H becomes BUS ACLO L, and BINIT L becomes BUS INIT L.



NOTE
Once a power down sequence is started, it must be completed before a power-up sequence is started

Figure D-1 Power Up/Power Down Timing

ARBITRATION

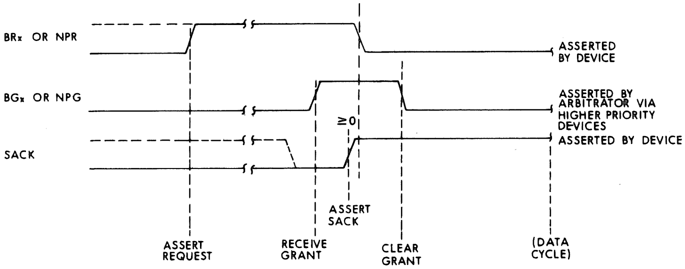


Figure D-2 UNIBUS Arbitration Sequence

Assert Request

The device wishing to become the bus master asserts a request line. This request may be ORed with other requests already present on the BRx/NPR line.

Receive Grant

Sometime later, the bus arbitrator will issue a grant. If no higher priority device at level BRx or NPR wants the grant, it will be passed to our example device.

Assert SACK

As soon as our device sees the grant, it asserts SACK to indicate it acknowledges its selection as the next bus master. It also removes its request.

Clear Grant

The device's assertion of SACK will cause the arbitrator to remove the grant. No further grants will be issued until the device removes SACK.

Data Cycle

One or more data cycles now follow.

DATA CYCLES

DATI/DATIP

DATI/DATIP need differ only by whether or not C0 is asserted. However, if a DATIP cycle is being performed, and the bus master will immediately be ready to perform the subsequent DATO, the master may choose to hold BBSY & SACK asserted. This will save it the trouble (and time) of re-arbitrating for the bus. This may improve overall system performance, as well as that of the device.

The example below is for a single transfer.

Negation of Previous BBSY

During its arbitration for the bus, the device asserted SACK. It now notices that BBSY is de-asserted.

Assert BBSY

Since the data section of bus is now available, the device asserts its mastership by asserting BBSY. It can now allow arbitration to resume by de-asserting SACK.

Assert A and C

When a previous SSYN de-asserts, and the new master has asserted BBSY, it can now assert Address and Control information. It must wait 75 ns for the address and control to deskew on the bus, and an addi-

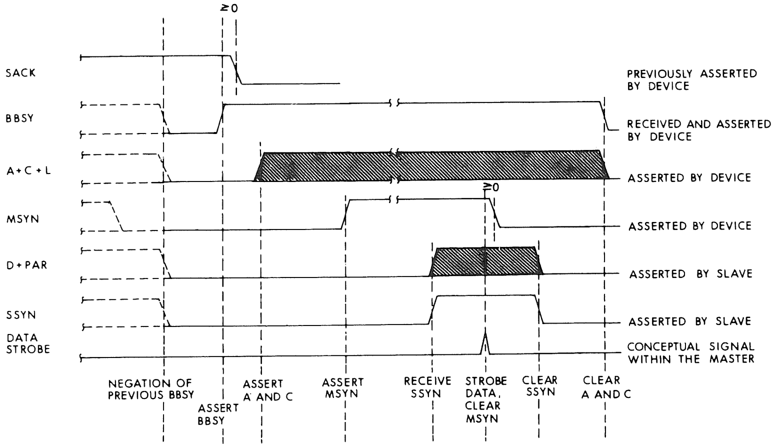


Figure D-3 UNIBUS Single Transfer DATI/DATIP Cycle

tional 75 ns for the slaves to make an address-decoder decision. This period is called "front-end deskew".

Assert MSYN
The master now asserts MSYN, indicating that an address is on the bus and enough time has elapsed for that address to be valid everywhere. If the address successfully selected a slave, the slave begins a read operation. This read operation may take an arbitrary amount of time up to the bus timeout value.

Receive SSYN
When the slave has data, it places that data (and the parity-error information) on the bus and asserts SSYN.

Strobe Data, Negate MSYN
The reception of SSYN at the master signals that a slave has been addressed and data is on the bus. The master must wait a minimum of 75nS, and may then sample the data. This decay is known as "data deskew". Once the master strobes in the data, it removes MSYN.

Clear SSYN
When the slave receives the negation of MSYN, it will remove data and SSYN from the bus.

Clear A and C
The master must hold the address and control asserted for a minimum of 75 ns after the master negates MSYN. This ensures that the "ad-

dress invalid" signal will be seen at all devices prior to the address actually becoming invalid. This prevents false selection of devices as the address delays. This delay is called "tail-end deskew".

DATO/DATOB

These two cycles differ only in the de-assertion/assertion of C0.

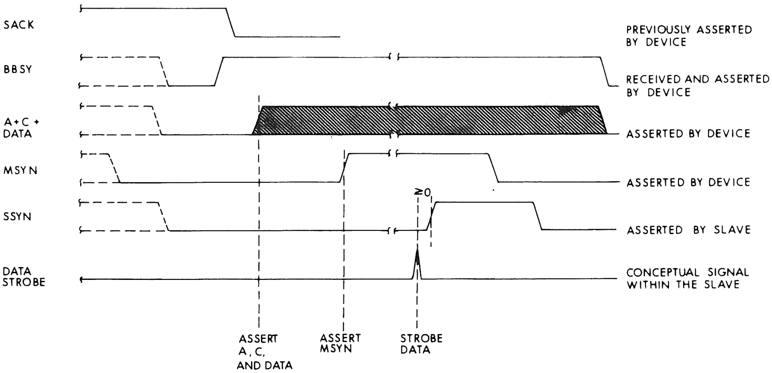


Figure D-4 UNIBUS Single Transfer DATO/DATOB Cycles

Only the differences between DATI/DATIP & DATO/DATOB will be described.

Assert A, C, and Data

The master asserts Address, Control and Data after its assertion of BBSY and the de-assertion of any previous SSYN.

Assert MSYN

Front-end deskew remains at 150 ns total. Note that it now "contains" the 75 ns required by the data to deskew.

Strobe Data

After decoding its address and MSYN, the slave strobes in the data. It then asserts SSYN. The master may now remove the data but must obey the tail-end deskew rules for address and control.

VECTOR-PASSING

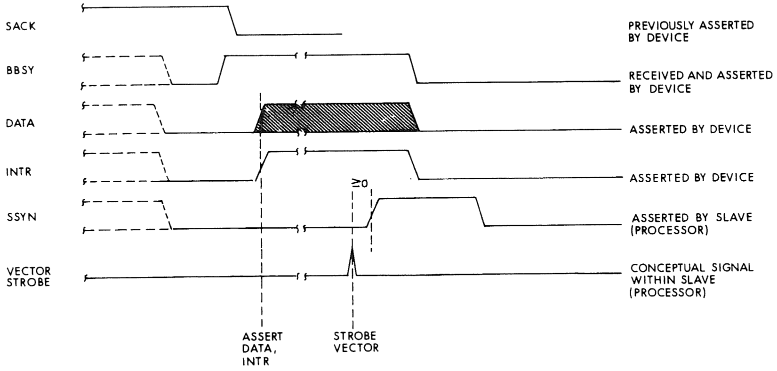

SACK
BBSY
DATA
INTR
SSYN
VECTOR STROBE
≥0
PREVIOUSLY ASSERTED BY DEVICE
RECEIVED AND ASSERTED BY DEVICE
ASSERTED BY DEVICE
ASSERTED BY DEVICE
ASSERTED BY SLAVE (PROCESSOR)
CONCEPTUAL SIGNAL WITHIN SLAVE (PROCESSOR)
ASSERT DATA, INTR
STROBE VECTOR

Figure D-5 UNIBUS Vector-Passing Cycle

Only the differences between vector-passing and DATI/DATIP will be described.

Assert Data, INTR

The master asserts Data and Intr after its assertion of BBSY and the de-assertion of any previous SSYN.

Strobe Vector

The slave (processor) must deskew the vector. After a 75 ns delay from its reception of INTR, the processor strobes in the vector and asserts SSYN.

Note that no address was passed so no front-end/tail-end deskews exist. Note also that this is the one case where a slave must perform the deskewing operation.

Notes

1. A prospective master may assert BBSY as soon as the previous master releases it. However, the new master must not assert any other lines until the previous slave releases SSYN. Because of this, many masters do not assert BBSY until both BBSY and SSYN are unasserted. They then simultaneously assert BBSY, Address, and Control (and Data, if DATO/DATOB).
2. A master may operate as slowly as its design requires, however, you'll pay the penalty in system performance. Optimize the design of masters.

3. A slave may operate as slowly on reads as the master will permit (via the value of its time-out delay). The slave takes this time by delaying the assertion of SSYN.
4. A slave may operate as slowly on writes as necessary. It must strobe in the data, and return SSYN within the time-out window, but it can then delay all other bus operations by holding SSYN asserted. No new master will use the data section of the bus while SSYN is asserted. However, you will pay the penalty in system performance for this technique; optimize the design of slaves.

MULTICYCLE TRANSFERS

Once a device becomes master it is sole controller of the UNIBUS until it chooses to release it. This implies both power and responsibility.

Power

A very fast master can significantly improve its performance (and possibly system performance) by performing multiple data cycles per arbitration of the bus. These multiple cycles may be:

5. A DATO/DATOB following a DATIP.
6. A block of DATI/DATIPs, DATO/DATOBs, or DATIP/DATO/DATOB pairs.

Responsibility

The designer of the master must be sensitive to that device's impact on the system as a whole. The master must not hold the bus so long that other devices cannot get the bus as they need it (for DMA or for passing interrupt vectors). Directly monitoring the bus-request lines is a useful dynamic technique. Alternatively, the designer should choose to only do a small number (1-4) of data cycles per bus arbitration.

Implementation

Ordinarily (for single data cycle bus cycles), the master asserts BBSY and releases SACK at the beginning of the data cycle. The bus is rearbitrated, a new propective master chosen. At the completion of the data cycle, the master releases BBSY and the bus is passed to the new master.

A multicycle master, on the other hand, holds SACK asserted until the beginning of its last data cycle. This assures that the prospective master is chosen based on the most recent arbitration data (rather than the data from n cycles ago).

In addition, after asserting BBSY, the multicycle master never releases it until the completion of the last data cycle. This ensures that the master remains in possession of the bus throughout the data cyles.

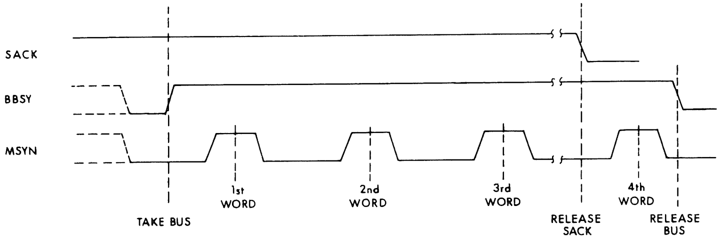


Figure D-6 Simplified Multicycle Timing Diagram

Take the bus, but don't allow re-arbitration yet.

Release SACK, allowing arbitration of the bus based on current requests.

Release BBSY, allowing the next master to take the data section of the bus.

APPENDIX E

LSI-11 BUS TECHNICAL SPECIFICATIONS

The LSI-11 Bus is the low-end member of DIGITAL's bus family. All DIGITAL microcomputers use the LSI-11 Bus. However, in order to use the 22-bit addressing capabilities of the LSI-11/23, the MICRO/PDP-11, and the PDP-11/23-PLUS, the extended LSI-11 Bus is required.

The LSI-11 Bus consists of 42 bidirectional and 2 unidirectional signal lines. These form the lines along which the processor, memory, and I/O devices communicate with each other.

Addresses, data, and control information are sent along these signal lines, some of which contain time-multiplexed information. The lines are divided as follows:

- Sixteen multiplexed data/address lines — BDAL<15:00>
- Two multiplexed address/parity lines — BDAL<17:16>
- Four extended address lines — BDAL<21:18>
- Six data transfer control lines — BBS7, BDIN, BDOUT, BRPLY, BSYNC, BWTBT
- Six system control lines — BHALT, BREF, BEVNT, BINIT, BDCOK, BPOK
- Ten interrupt control and direct memory access control lines — BIAKO, BIAKI, BIRQ4, BIRQ5, BIRQ6, BIRQ7, BDMGO, BDMR, BSACK, BDMGI

In addition, a number of power, ground, and spare lines have been defined for the bus. For a detailed description of these lines, please refer to Table E-1.

The discussion in this chapter applies to the general 22-bit physical address capability. In cases where modules utilize 16- or 18-bit physical address space, this discussion applies to the lines that are utilized by those modules.

Most LSI-11 Bus signals are bidirectional and use terminations for a negated (high) signal level. Devices connect to these lines via high-impedance bus receivers and open collector drivers. *The asserted state is produced when a bus driver asserts the line low.* Although bidirectional lines are electrically bidirectional (any point along the line can be driven or received), certain lines are functionally unidirectional. These lines communicate to or from a bus master (or signal source), but not both. Interrupt acknowledge (BIAK) and direct memory access grant (BDMG) signals are physically unidirectional in a daisy-chain fashion. These signals originate at the processor output signal pins. Each is received on device input pins (BIAKI or BDMGI) and conditionally retransmitted via device output pins (BIAKO or BDMGO). These signals are received from higher-priority devices and are retransmitted to lower-priority devices along the bus, establishing the position-dependent priority scheme.

Master/Slave Relationship

Communication between devices on the bus is asynchronous. A master/slave relationship exists throughout each bus transaction. At any time, there is one device that has control of the bus. This controlling device is termed the bus master. The master device controls the bus when communicating with another device on the bus, termed the slave. The bus master (typically the processor or a DMA device) initiates a bus transaction. The slave device responds by acknowledging the transaction in progress and by receiving data from, or transmitting data to, the bus master. LSI-11 Bus control signals transmitted or received by the bus master or bus slave device must complete the sequence according to bus protocol.

The processor controls bus arbitration, i.e., which device becomes bus master at any given time. A typical example of this relationship is the processor, as master, fetching an instruction from memory, which is always a slave. Another example is a disk, as master, transferring data to memory as slave. Communication on the LSI-11 Bus is interlocked so that for certain control signals issued by the master device, there must be a response from the slave in order to complete the transfer. It is the master/slave signal protocol that makes the LSI-11 Bus asynchronous. The asynchronous operation precludes the need for synchronizing with, and waiting for, clock pulses.

Since bus cycle completion by the bus master requires response from the slave device, each bus master must include a time-out error circuit that will abort the bus cycle if the slave device does not respond to the bus transaction within 10 microseconds. The actual time before a time-out error occurs must be longer than the reply time of the slowest peripheral or memory device on the bus.

See Table E-1 for more detail on signal functions.

Table E-1 Signal Assignments

DATA AND ADDRESS

Nomenclature	Pin Assignment
BDAL0	AU2
BDAL1	AV2
BDAL2	BE2
BDAL3	BF2
BDAL4	BH2
BDAL5	BJ2
BDAL6	BK2
BDAL7	BL2
BDAL8	BM2
BDAL9	BN2
BDAL10	BP2
BDAL11	BR2
BDAL12	BS2
BDAL13	BT2
BDAL14	BU2
BDAL15	BV2
BDAL16	AC1
BDAL17	AD1
BDAL18	BC1
BDAL19	BD1
BDAL20	BE1
BDAL21	BF1

CONTROL

Nomenclature	Pin Assignment
	Data Control
BDOUT	AE2
BRPLY	AF2
BDIN	AH2
BSYNC	AJ2
BWTBT	AK2
BBS7	AP2
	Interrupt Control
BIRQ7	BP1
BIRQ6	AB1
BIRQ5	AA1

BIRQ4	AL2
BIAKO	AN2
BIAKI	AM2
	DMA Control
BDMR	AN1
BSACK	BN1
BDMGO	AS2
BMDGI	AR2
	System Control
BHALT	AP1
BREF	AR1
BEVNT	BR1
BINIT	AT2
BDCOK	BA1
BPOK	BB1

POWER AND GROUND

Nomenclature	**Pin Assignment**
+5B (battery) or +12B(battery)	AS1
+12B	BS1
+5B	AV1
+5	AA2
+5	BA2
+5	BV1
+12	AD2
+12	BD2
−12	AB2
−12	BB2
GND	AC2
GND	AJ1
GND	AM1
GND	AT1
GND	BC2
GND	BJ1
GND	BM1
GND	BT1

SPARES

Nomenclature	Pin Assignment
SSpare1	AE1
SSpare3	AH1
SSpare8	BH1
SSpare2	AF1
MSpareA	AK1
MSpareB	AL1
MSpareB	BK1
MSpareB	BL1
PSpare1	AU1
ASpare2	BU1

DATA TRANSFER BUS CYCLES

Data transfer bus cycles are listed and defined in Table E-2.

Table E-2 Data Transfer Operations

Bus Cycle Mnemonic	Description	Function (with Respect to the Bus Master)
DATI	Data word input	Read
DATO	Data word output	Write
DATOB	Data byte output	Write byte
DATIO	Data word input/output	Read-modify-write
DATIOB	Data word input/ byte output	Read-modify-write byte
DATBI	Data block input	Read block
DATBO	Data block output	Write block

These bus cycles, executed by bus master devices, transfer 16-bit words or 8-bit bytes to or from slave devices. In block mode, multiple words may be transferred to sequential word addresses, starting from a single bus address. The bus signals listed in Table E-3 are used in the data transfer operations described in Table E-2.

Table E-3 Bus Signals For Data Transfers

Mnemonic	Description	Function
BDAL<21:00> L	22 Data/address lines	BDAL<15:00> L are used for word and byte transfers. BDAL<17:16> L are used for extended addressing, memory parity error (16), and memory parity error enable (17) functions. BDAL<21:18> L are used for extended addressing beyond 256 KB.
BSYNC L	Bus Cycle Control	Indicates bus transaction in progress.
BDIN L	Data input indicator	Strobe signals.
BDOUT L	Data output indicator	Strobe signals.
BRPLY L	Slave's acknowledge of bus cycle	Strobe signals.
BWTBT L	Write/byte control	Control signals.
BBS7	I/O device select	Indicates address is in the I/O page.

Data transfer bus cycles can be reduced to five basic types: DATI, DATO(B), DATIO(B), DATBI, and DATBO. These transactions occur between the bus master and one slave device selected during the addressing portion of the bus cycle.

Bus Cycle Protocol

Before initiating a bus cycle, the previous bus transaction must have been completed (BSYNC L negated) and the device must become bus master. The bus cycle can be divided into two parts, an addressing portion, and a data transfer portion. During the addressing portion, the bus master outputs the address for the desired slave device, memory location or device register. The selected slave device responds by latching the address bits and holding this condition for the duration of the bus cycle until BSYNC L becomes negated. During the data transfer portion, the actual data transfer occurs.

Device Addressing — The device addressing portion of a data transfer bus cycle comprises an address setup and deskew time and an address hold and deskew time. During the address setup and deskew time, the bus master does the following:

- Asserts BDAL<21:00> L with the desired slave device address bits
- Asserts BBS7 L if a device in the I/O page is being addressed
- Asserts BWTBT L if the cycle is a DATO(B) or DATBO bus cycle

During this time the address, BBS7 L, and BWTBT L signals are asserted at the slave bus receiver for at least 75 ns before BSYNC goes active. Devices in the I/O page ignore the nine high-order address bits BDAL<21:13> and instead decode BBS7 L along with the thirteen low-order address bits. An active BWTBT L signal during address setup time indicates that a DATO(B) or DATBO operation follows, while an inactive BWTBT L indicates a DATI, DATBI or DATIO(B) operation.

The address hold and deskew time begins after BSYNC L is asserted.

The slave device uses the active BSYNC L bus receiver output to clock BDAL address bits, BBS7 L, and BWTBT L into its internal logic. BDAL<21:00> L, BBS7 L, and BWTBT L will remain active for 25 ns (minimum) after BSYNC L bus receiver goes active. BSYNC L remains active for the duration of the bus cycle.

Memory and peripheral devices are addressed similarly except for the way the slave device responds to BBS7 L. Addressed peripheral devices must not decode address bits on BDAL<21:13> L. Addressed peripheral devices may respond to a bus cycle when BBS7 L is asserted (low) during the addressing portion of the cycle. When asserted, BBS7 L indicates that the device address resides in the I/O page (the upper 4K address space). Memory devices generally do not respond to addresses in the I/O page; however, some system applications may permit memory to reside in the I/O page for use as DMA buffers, read-only memory bootstraps or diagnostics, etc.

DATI — The DATI bus cycle, illustrated in Figure E-1, is a read operation. During DATI, data are input to the bus master. Data consist of 16-bit word transfers over the bus. During the data transfer portion of the DATI bus cycle, the bus master asserts BDIN L 100 ns minimum after BSYNC L is asserted. The slave device responds to BDIN L active as follows:

- Asserts BRPLY L 0 ns minimum (8 μs maximum to avoid bus timeout) after receiving BDIN L and 125 ns (maximum) before BDAL bus driver data bits are valid.
- Asserts BDAL<21:00> L with the addressed data and error information 0 ns minimum after receiving BDIN and 125 ns maximum after assertion of BRPLY.

When the bus master receives BRPLY L, it does the following:

- Waits at least 200 ns deskew time and then accepts input data at BDAL<17:00> L bus receivers. BDAL<17:16> L are used for transmitting parity errors to the master.
- Negates BDIN L 200 ns (minimum) to 2 microseconds (maximum) after BRPLY L goes active.

The slave device responds to BDIN L negation by negating BRPLY L and removing read data from BDAL bus drivers. BRPLY L must be negated 100 ns (maximum) prior to removal of read data. The bus master responds to the negated BRPLY L by negating BSYNC L.

Conditions for the next BSYNC L assertion are as follows:

- BSYNC L must remain negated for 200 ns (minimum)
- BSYNC L must not become asserted within 300 ns of previous BRPLY L negation

Figure E-2 illustrates DATI bus cycle timing.

NOTE

Continuous assertion of BSYNC L retains control of the bus by the bus master, and the previously addressed slave device remains selected. This is done for DATIO(B) bus cycles where DATO or DATOB follows a DATI without BSYNC L negation and a second device addressing operation. Also, a slow slave device can hold off data transfers to itself by keeping BRPLY L asserted, which will cause the master to keep BSYNC L asserted.

DATO(B) — DATO(B), illustrated in Figure E-3, is a write operation. Data are transferred in 16-bit words (DATO) or 8-bit bytes (DATOB) from the bus master to the slave device. The data transfer output can occur after the addressing portion of a bus cycle when BWTBT L has been asserted by the bus master, or immediately following an input transfer part of a DATIO(B) bus cycle.

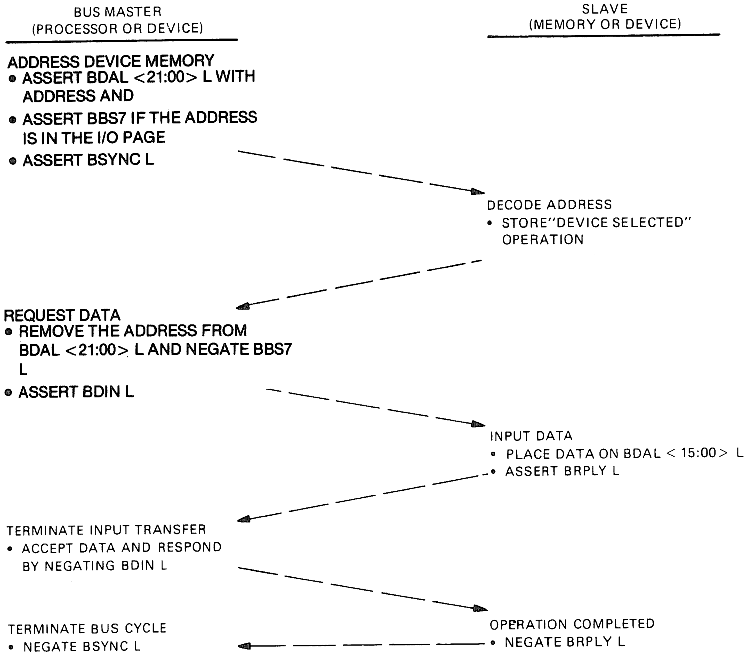
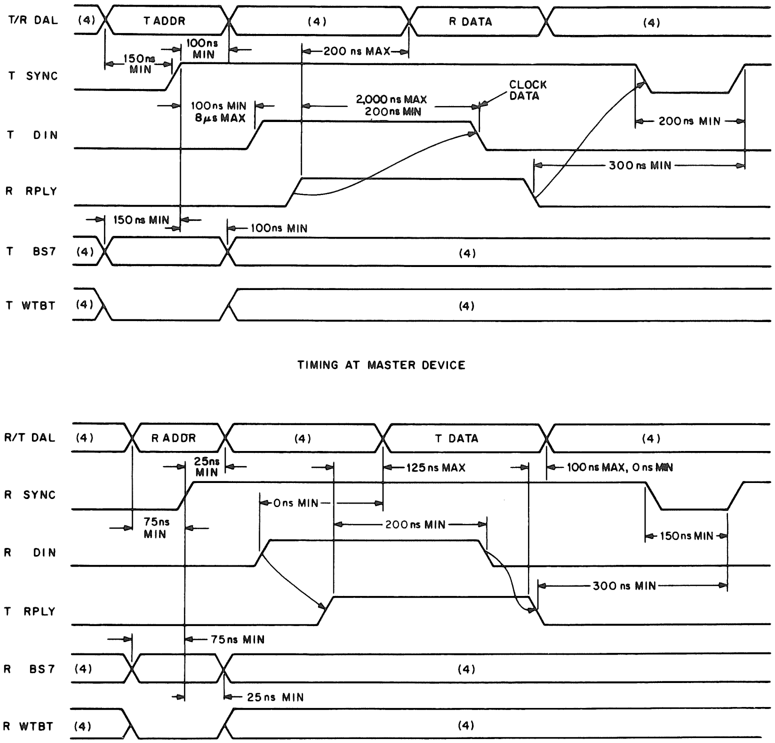

BUS MASTER
(PROCESSOR OR DEVICE)
SLAVE
(MEMORY OR DEVICE)
ADDRESS DEVICE MEMORY
• ASSERT BDAL <21:00> L WITH ADDRESS AND
• ASSERT BBS7 IF THE ADDRESS IS IN THE I/O PAGE
• ASSERT BSYNC L
DECODE ADDRESS
• STORE"DEVICE SELECTED" OPERATION
REQUEST DATA
• REMOVE THE ADDRESS FROM BDAL <21:00> L AND NEGATE BBS7 L
• ASSERT BDIN L
INPUT DATA
• PLACE DATA ON BDAL < 15:00> L
• ASSERT BRPLY L
TERMINATE INPUT TRANSFER
• ACCEPT DATA AND RESPOND BY NEGATING BDIN L
OPERATION COMPLETED
• NEGATE BRPLY L
TERMINATE BUS CYCLE
• NEGATE BSYNC L

Figure E-1 DATI Bus Cycle

The data transfer portion of a DATO(B) bus cycle comprises a data setup and deskew time and a data hold and deskew time.

During the data setup and deskew time, the bus master outputs the data on BDAL<15:00> L at least 100 ns after BSYNC L is asserted if the transfer is a word transfer. If it is a word transfer, the bus master negates BWTBT L at least 100 ns after BSYNC L assertion. BWTBT L remains negated for the length of the bus cycle. If the transfer is a byte transfer, BWTBT L remains asserted. If it is the output of a DATIOB, BTWBT L becomes asserted and lasts the duration of the bus cycle.



TIMING AT MASTER DEVICE

TIMING AT SLAVE DEVICE

NOTES:

1. Timing shown at Master and Slave Device Bus Driver inputs and Bus Receiver Outputs.
2. Signal name prefixes are defined below:
   T = Bus Driver Input
   R = Bus Receiver Output
3. Bus Driver Output and Bus Receiver Input signal names include a "B" prefix.
4. Don't care condition.

Figure E-2 DATI Bus Cycle Timing

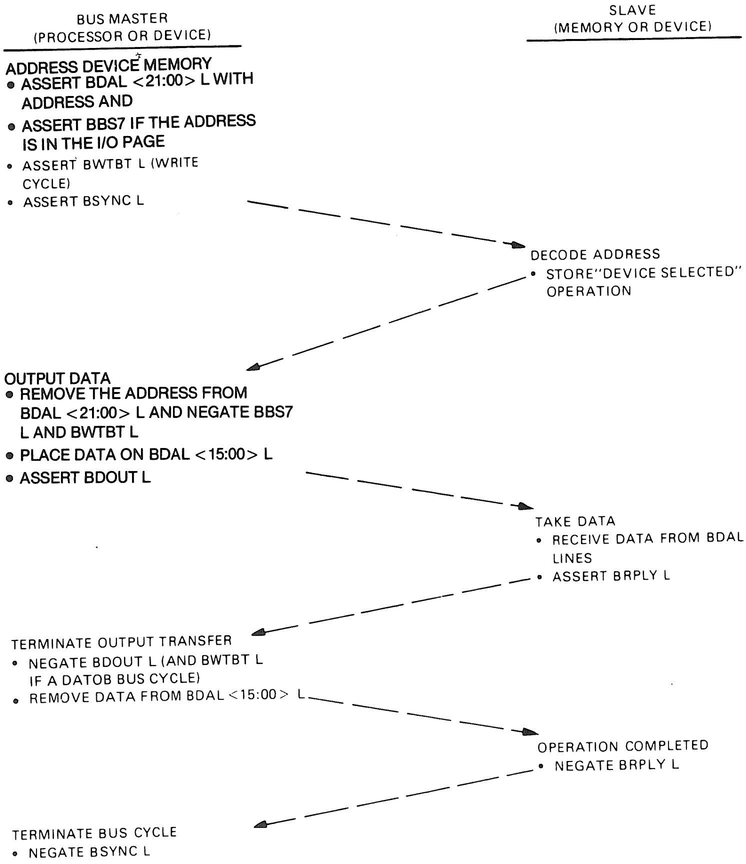
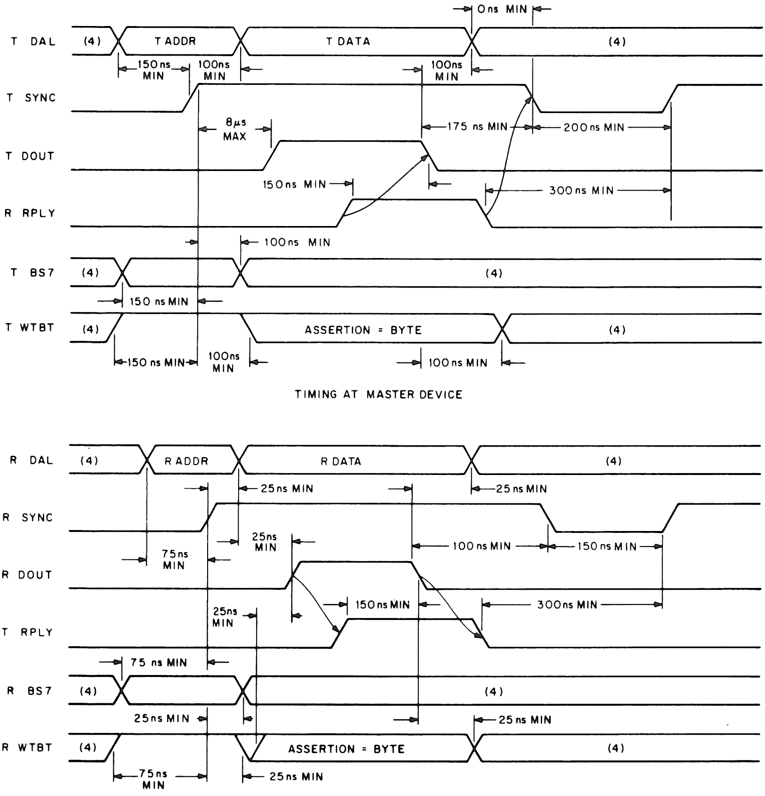

BUS MASTER
(PROCESSOR OR DEVICE)
SLAVE
(MEMORY OR DEVICE)
ADDRESS DEVICE MEMORY
• ASSERT BDAL <21:00> L WITH ADDRESS AND
• ASSERT BBS7 IF THE ADDRESS IS IN THE I/O PAGE
• ASSERT BWTBT L (WRITE CYCLE)
• ASSERT BSYNC L
DECODE ADDRESS
• STORE "DEVICE SELECTED" OPERATION
OUTPUT DATA
• REMOVE THE ADDRESS FROM BDAL <21:00> L AND NEGATE BBS7 L AND BWTBT L
• PLACE DATA ON BDAL <15:00> L
• ASSERT BDOUT L
TAKE DATA
• RECEIVE DATA FROM BDAL LINES
• ASSERT BRPLY L
TERMINATE OUTPUT TRANSFER
• NEGATE BDOUT L (AND BWTBT L IF A DATOB BUS CYCLE)
• REMOVE DATA FROM BDAL <15:00> L
OPERATION COMPLETED
• NEGATE BRPLY L
TERMINATE BUS CYCLE
• NEGATE BSYNC L

Figure E-3 DATO or DATOB Bus Cycle

During a byte transfer, BDAL <00> L selects the high or low byte. This occurs while in the addressing portion of the cycle. If asserted, the high byte (BDAL<15:08> L) is selected; otherwise, the low byte (BDAL<07:00> L) is selected. An asserted BDAL16 L at this time will force a parity error to be written into memory if the memory is a parity-type memory. BDAL17 L is not used for write operations. The bus master asserts BDOUT L at least 100 ns after BDAL and BWTBT L bus drivers are stable. The slave device responds by asserting BRPLY L within 10 microseconds to avoid bus time-out. This completes the data setup and deskew time.



TIMING AT MASTER DEVICE

TIMING AT SLAVE DEVICE

NOTES

1 Timing shown at Master and Slave Device Bus Driver Inputs and Bus Receiver Outputs

2 Signal name prefixes are defined below

T = Bus Driver Input
R = Bus Receiver Output

3 Bus Driver Output and Bus Receiver Input signal names include a "B" prefix

4 Don't care condition

Figure E-4 DATO or DATOB Bus Cycle Timing

During the data hold and deskew time the bus master receives BRPLY L and negates BDOUT L. BDOUT L must remain asserted for at least 150 ns from the receipt of BRPLY L before being negated by the bus master. BDAL<17:00> L bus drivers remain asserted for at least 100 ns after BDOUT L negation. The bus master then negates BDAL inputs.

During this time, the slave device senses BDOUT L negation. The data are accepted and the slave device negates BRPLY L. The bus master responds by negating BSYNC L. However, the processor will not negate BSYNC L for at least 175 ns after negating BDOUT L. This completes the DATO(B) bus cycle. Before the next cycle BSYNC L must remain unasserted for at least 200 ns. Figure E-4 illustrates DATO(B) bus cycle timing.

DATIO(B) — The protocol for a DATIO(B) bus cycle is identical to the addressing and data transfer portions of the DATI and DATO(B) bus cycles, and is illustrated in Figure E-5. After addressing the device, a DATI cycle is performed as explained earlier; however, BSYNC L is not negated. BSYNC L remains active for an output word or byte transfer [DATO(B)]. The bus master maintains at least 200 ns between BRPLY L negation during the DATI cycle and BDOUT L assertion. The cycle is terminated when the bus master negates BSYNC L, as described for DATO(B). Figure E-6 illustrates DATIO(B) bus cycle timing.

DIRECT MEMORY ACCESS

The direct memory access (DMA) capability allows direct data transfer between I/O devices and memory. This is useful when using mass storage devices (e.g., disks) that move large blocks of data to and from memory. A DMA device needs to know only the starting address in memory, the starting address in mass storage, the length of the transfer, and whether the operation is read or write. When this information is available, the DMA device can transfer data directly to or from memory. Since most DMA devices must perform data transfers in rapid succession or lose data, DMA devices are provided the highest priority.

DMA is accomplished after the processor (normally bus master) has passed bus mastership to the highest-priority DMA device that is requesting the bus. The processor arbitrates all requests and grants the bus to the DMA device located electrically closest to it. A DMA device remains bus master indefinitely until it relinquishes its mastership. The following control signals are used during bus arbitration.

BDMGI L	DMA Grant Input
BDMGO L	DMA Grant Output
BDMR L	DMA Request Line
BSACK L	Bus Grant Acknowledge

DMA Protocol

A DMA transaction can be divided into three phases:

- Bus mastership acquisition phase

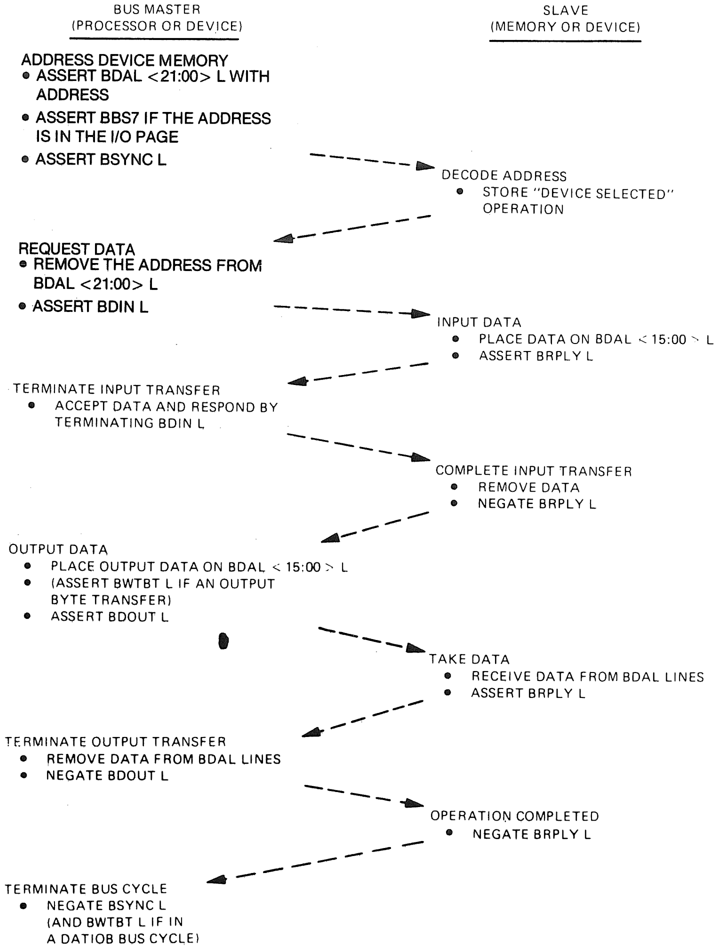
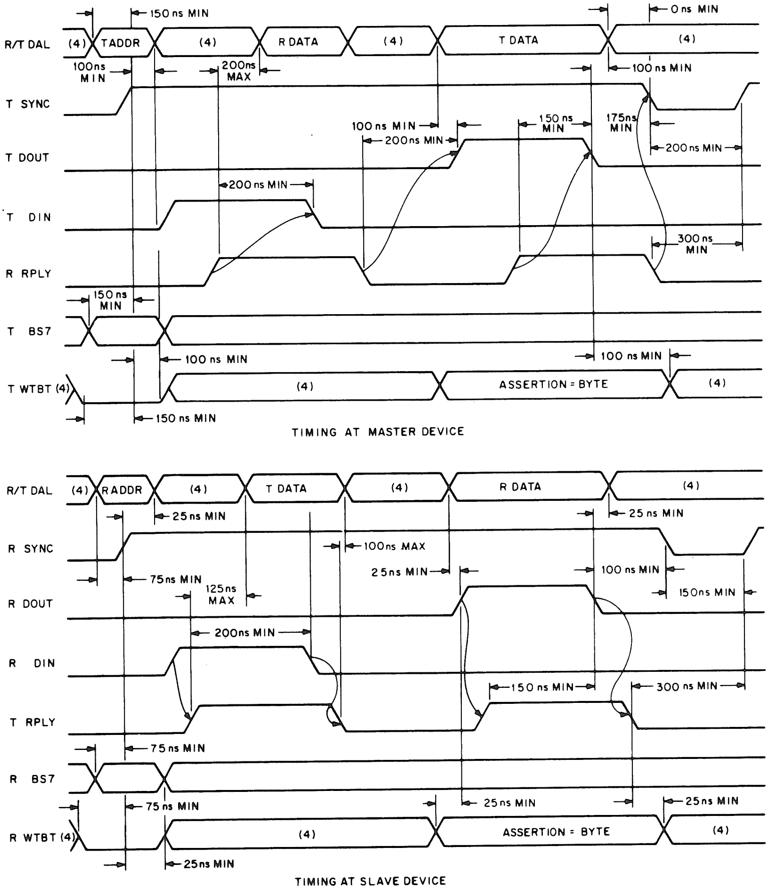

BUS MASTER
(PROCESSOR OR DEVICE)
SLAVE
(MEMORY OR DEVICE)
ADDRESS DEVICE MEMORY
● ASSERT BDAL <21:00> L WITH ADDRESS
● ASSERT BBS7 IF THE ADDRESS IS IN THE I/O PAGE
● ASSERT BSYNC L
DECODE ADDRESS
● STORE "DEVICE SELECTED" OPERATION
REQUEST DATA
● REMOVE THE ADDRESS FROM BDAL <21:00> L
● ASSERT BDIN L
INPUT DATA
● PLACE DATA ON BDAL < 15:00 > L
● ASSERT BRPLY L
TERMINATE INPUT TRANSFER
● ACCEPT DATA AND RESPOND BY TERMINATING BDIN L
COMPLETE INPUT TRANSFER
● REMOVE DATA
● NEGATE BRPLY L
OUTPUT DATA
● PLACE OUTPUT DATA ON BDAL < 15:00 > L
● (ASSERT BWTBT L IF AN OUTPUT BYTE TRANSFER)
● ASSERT BDOUT L
TAKE DATA
● RECEIVE DATA FROM BDAL LINES
● ASSERT BRPLY L
TERMINATE OUTPUT TRANSFER
● REMOVE DATA FROM BDAL LINES
● NEGATE BDOUT L
OPERATION COMPLETED
● NEGATE BRPLY L
TERMINATE BUS CYCLE
● NEGATE BSYNC L (AND BWTBT L IF IN A DATIOB BUS CYCLE)

Figure E-5 DATIO or DATIOB Bus Cycle

- Data transfer phase
- Bus mastership relinquish phase

During the bus mastership acquisition phase, a DMA device requests the bus by asserting BDMR L. The processor arbitrates the request and initiates the transfer of bus mastership by asserting BDMGO L.



NOTES

1 Timing shown at Requesting Device Bus Driver Inputs and Bus Receiver Outputs

2 Signal name prefixes are defined below

T = Bus Driver Input
R = Bus Receiver Output

3. Bus Driver Output and Bus Receiver Input signal names include a "B" prefix

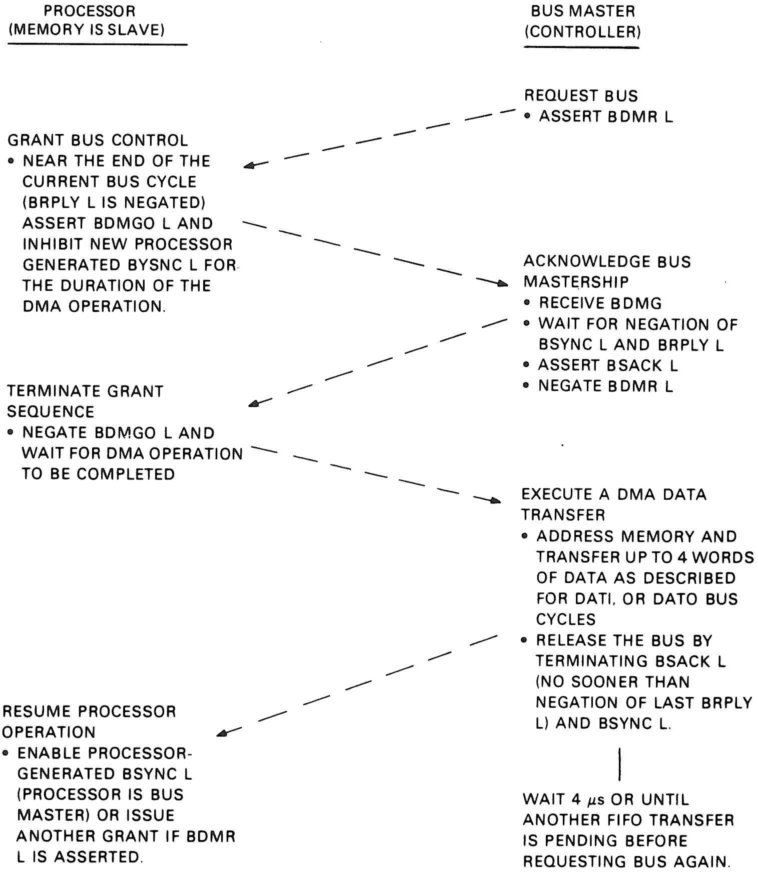
4 Don't care condition

Figure E-6 DATIO or DATIOB Bus Cycle Timing

The maximum time between BDMR L assertion and BDMGO L assertion is DMA latency. This time is processor-dependent. BDMGO L/ BDMGI L is one signal that is daisy-chained through each module in the backplane. It is driven out of the processor on the BDMGO L pin, enters each module on the BDMGI L pin and exits on the BDMGO L pin. This signal passes through the modules in descending order of priority until it is stopped by the requesting device. The requesting device blocks the output of BMDGO L and asserts BSACK L. If BDMR L is continuously asserted, the bus will be hung.

During the data transfer phase, the DMA device continues asserting BSACK L. The actual data transfer is performed as described earlier.

The DMA device can assert BSYNC L for a data transfer 250 ns (minimum) after it receives BDMGI L and its BSYNC L bus receiver becomes negated.



PROCESSOR
(MEMORY IS SLAVE)

BUS MASTER
(CONTROLLER)

REQUEST BUS
- ASSERT BDMR L

GRANT BUS CONTROL
- NEAR THE END OF THE CURRENT BUS CYCLE (BRPLY L IS NEGATED) ASSERT BDMGO L AND INHIBIT NEW PROCESSOR GENERATED BYSNC L FOR THE DURATION OF THE DMA OPERATION.

ACKNOWLEDGE BUS MASTERSHIP
- RECEIVE BDMG
- WAIT FOR NEGATION OF BSYNC L AND BRPLY L
- ASSERT BSACK L
- NEGATE BDMR L

TERMINATE GRANT SEQUENCE
- NEGATE BDMGO L AND WAIT FOR DMA OPERATION TO BE COMPLETED

EXECUTE A DMA DATA TRANSFER
- ADDRESS MEMORY AND TRANSFER UP TO 4 WORDS OF DATA AS DESCRIBED FOR DATI, OR DATO BUS CYCLES
- RELEASE THE BUS BY TERMINATING BSACK L (NO SOONER THAN NEGATION OF LAST BRPLY L) AND BSYNC L.

WAIT 4 μs OR UNTIL ANOTHER FIFO TRANSFER IS PENDING BEFORE REQUESTING BUS AGAIN.

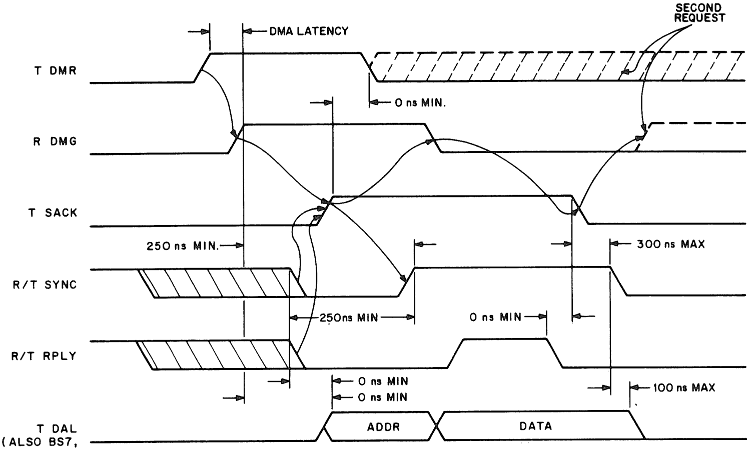
RESUME PROCESSOR OPERATION
- ENABLE PROCESSOR-GENERATED BSYNC L (PROCESSOR IS BUS MASTER) OR ISSUE ANOTHER GRANT IF BDMR L IS ASSERTED.

Figure E-7 DMA Protocol

During the bus mastership relinquish phase, the DMA device relinquishes the bus by negating BSACK L. This occurs after completing (or aborting) the last data transfer cycle (BRPLY L negated). BSACK L may be negated up to a maximum of 300 ns before negating BSYNC L. Figure E-7 illustrates the DMA protocol and Figure E-8 illustrates DMA request/grant timing.

NOTE

If multiple data transfers are performed during this phase, consideration must be given to the use of the bus for other system functions, such as memory refresh (if required).



NOTES:

1. Timing shown at requesting device bus driver inputs and bus receiver outputs.
2. Signal name prefixes are defined below:
   T = Bus Driver Input
   R = Bus Receiver Output
3. Bus Driver Output and Bus Receiver Input signal names include a "B" prefix.

Figure E-8 DMA Request/Grant Timing

BLOCK MODE DMA

For increased throughput, block mode DMA may be implemented on a device for use with memories that support this type of transfer. In a block mode transaction, the starting memory address is asserted, followed by data for that address, and data for consecutive addresses.

By eliminating the assertion of the address for each data word, the transfer rate is almost doubled. The DATBI and DATBO bus cycles are described below.

DATBI

The device addressing portion of the cycle is the same as described earlier for other bus cycles. The bus master gates BDAL <21:00>, BBS7, and the negation of BWTBT onto the bus.

The master asserts the first BDIN 100 ns after BSYNC, and asserts BBS7 a maximum of 50 ns after asserting BDIN for the first time. BBS7 is a request to the slave for a block mode transfer. BBS7 remains asserted until a maximum of 50 ns after the assertion of BDIN for the last time. BBS7 may be gated as soon as the conditions for asserting BDIN are met.

The slave asserts BRPLY a minimum of 0 ns (8 μs maximum to avoid bus timeout) after receiving BDIN. It asserts BREF concurrently with BRPLY if it is a block mode device capable of supporting another BDIN after the current one. The slave gates BDAL <15:00> onto the bus a minimum of 0 ns after the assertion of BDIN and 125 ns maximum after the assertion of BRPLY.

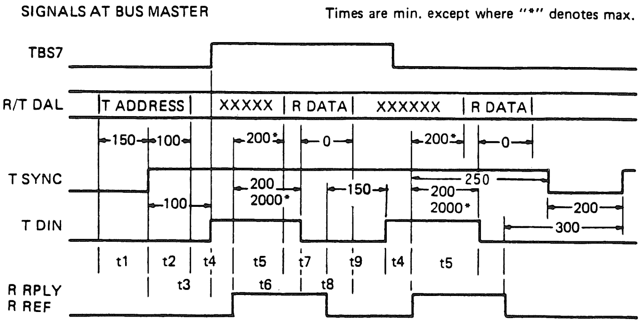
The master receives the stable data from 200 ns maximum after the assertion of BRPLY until 20 ns minimum after the negation of BDIN. It negates BDIN a minimum of 200 ns after the assertion of BRPLY.

The slave negates BRPLY a minimum of 0 ns after the negation of BDIN. If BBS7 and BREF are both asserted when BRPLY is negated, the slave prepares for another BDIN cycle. BBS7 is stable from 125 ns after BDIN is asserted until 150 ns after BRPLY is negated. The master asserts BDIN a minimum of 150 ns after BRPLY is negated and the cycle is continued as before. (BBS7 remains asserted and the slave responds to BDIN with BRPLY and BREF.) BREF is stable from 75 ns after BRPLY is asserted until a minimum of 20 ns after BDIN is negated.

If BBS7 and BREF are not both asserted when BRPLY is negated, the slave removes the data from the bus a minimum of 0 ns and 100 ns maximum after negating BRPLY. The master negates BSYNC a minimum of 250 ns after the assertion of the last BRPLY and a minimum of 0 ns after the negation of that BRPLY.

DATBO

The device addressing portion of the cycle is the same as described earlier. The bus master gates BDAL <21:00>, BBS7, and the assertion of BWTBT onto the bus.



t1 = address to T SYNC 150ns MIN.

t2 = address hold 100ns min

t3 = T SYNC to T DLN 100ns min

t4 = T DIN to R RPLY
T (drive) + T (prop) + T (receive) + T (delay)
+ T (drive) + T (prop) + T (receive)

t5 = R RPLY to data 200ns max

t6 = R RPLY to T DIN 200ns min

t7 = T DIN to R RPLY
= T (drive) + (prop) + T (receive) + T (delay)
+ T (drive + T (prop) + T (receive)

t8 = R RPLY to data 0ns min

t9 = R RPLY to T DIN 150ns min

T cell = t4 + t6 + t7 + t9 —since t6 must be > t5 for master to have valid data - and t9 >t8

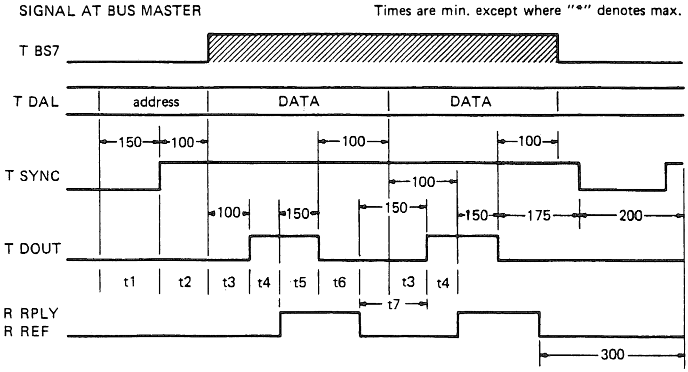
Figure E-9 DATBI Bus Cycle Timing

A minimum of 100 ns after BSYNC is asserted, data on BDAL <15:00> and the negated BWTBT are put onto the bus. The master then asserts BDOUT a minimum of 100 ns after gating the data.

The slave receives stable data and BWTBT from a minimum of 25 ns before the assertion of BDOUT to a minimum of 25 ns after the negation of BDOUT. The slave asserts BRPLY a minimum of 0 ns after receiving BDOUT. It also asserts BREF concurrently with BRPLY if it is a block mode device capable of supporting another BDOUT after the current one.

The master negates BDOUT 150 ns minimum after the assertion of BRPLY. If BREF was asserted when BDOUT was negated and the master wants to transmit more data in this block mode cycle, then the new data is gated onto the bus 100 ns minimum after BDOUT is negated. BREF is stable from 75 ns maximum after BRPLY is asserted until 20 ns minimum after BDOUT is negated. The master asserts BDOUT 100 ns minimum after gating new data onto the bus and 150 ns minimum after BRPLY negates. The cycle continues as before.

If BREF was not asserted when BDOUT was negated or if the bus master does not want to transmit more data in this cycle, then the master removes data from the bus a minimum of 100 ns after negating BDOUT. The slave negates BRPLY a minimum of 0 ns after negating BDOUT. The bus master negates BSYNC a minimum of 175 ns after negating BDOUT, and a minimum of 0 ns after the negation of BRPLY.



t1	=	address to T SYNC	150ns min
t2	=	address hold	100ns min
t3	=	data to T DOUT	100ns min
t4	=	T DOUT to R RPLY	
	=	T (drive) + T (prop) + T (receive) + T (delay) + T (drive) + T (prop) + T (receive)	
t5	=	R RPLY to T DOUT	150ns min
t6	=	T DOUT to R RPLY	
	=	T (drive) + T (prop) + T (receive) + T (delay) + T (drive) + T (prop) + T (receive)	
t7	=	R RPLY to T DOUT	150ns min
T cell	=	t3 + t4 + t5 + t6 + t7	—since t3 < t7

Figure E-10 DATBO Bus Cycle Timing

DMA Guidelines

- Systems with memory refresh over the bus must not include devices that perform more than one transfer per acquisition.
- Bus masters that do not use block mode are limited to four DATI, four DATO, or two DATIO transfers per acquisition.
- Block mode bus masters that do not monitor BDMR are limited to eight transfers per acquisition.
- If BDMR is not asserted after the seventh transfer, block mode bus masters that do monitor BDMR may continue making transfers until the bus slave fails to assert BREF or until they reach the total maximum of 16 transfers. Otherwise, they stop after eight transfers.

INTERRUPTS

The interrupt capability of the LSI-11 Bus allows any I/O device to temporarily suspend (interrupt) current program execution and divert processor operation to service the requesting device. The processor inputs a vector from the device to start the service routine (handler). Like the device register address, hardware fixes the device vector at locations within a designated range below location 001000. The vector indicates the first of a pair of addresses. The content of the first address is read by the processor and is the starting address of the interrupt handler. The content of the second address is a new processor status word (PS). The new PS can raise the interrupt priority level, thereby preventing lower- level interrupts from breaking into the current interrupt service routine. Control is returned to the interrupted program when the interrupt handler is ended. The original interrupted program's address (PC) and its associated PS are stored on a stack. The original PC and PS are restored by a return from interrupt (RTI or RTT) instruction at the end of the handler. The use of the stack and the LSI-11 Bus interrupt scheme can allow interrupts to occur within interrupts (nested interrupts), depending on the PS.

Interrupts can be caused by LSI-11 Bus options or the CPU. Those interrupts that originate from within the processor are called traps. Traps are caused by programming errors, hardware errors, special instructions, and maintenance features.

The LSI-11 Bus signals used in interrupt transactions are:

BIRQ4 L	Interrupt request priority level 4
BIRQ5 L	Interrupt request priority level 5
BIRQ6 L	Interrupt request priority level 6
BIRQ7 L	Interrupt request priority level 7

BIAKI L	Interrupt acknowledge input
BIAKO L	Interrupt acknowledge output
BDAL<21:00> L	Data/address lines
BDIN L	Data input strobe
BRPLY L	Reply

Device Priority

The LSI-11 Bus supports the following two methods of device priority:

- Distributed Arbitration—priority levels are implemented on the hardware. When devices of equal priority level request an interrupt, priority is given to the device electrically closest to the processor.
- Position-Defined Arbitration—priority is determined solely by electrical position on the bus. The closer a device is to the processor, the higher its priority is.

Interrupt Protocol

Interrupt protocol on the LSI-11/23 has three phases: interrupt request phase, interrupt acknowledge, and priority arbitration phase, and interrupt vector transfer phase. Figure E-11 illustrates the interrupt request/acknowledge sequence.

The interrupt request phase begins when a device meets its specific conditions for interrupt requests. For example, the device is ready, done, or an error has occurred. The interrupt enable bit in a device status register must be set. The device then initiates the interrupt by asserting the interrupt request line(s). BIRQ4 L is the lowest hardware priority level and is asserted for all interrupt requests for compatibility with previous LSI-11 processors. The level a device is configured at must also be asserted. A special case exists for level 7 devices which must also assert level 6. See the arbitration discussion below involving the 4-level scheme for an explanation.

Interrupt Level	**Lines Asserted by Device**
4	BIRQ4 L
5	BIRQ4 L, BIRQ5 L
6	BIRQ4 L, BIRQ6 L
7	BIRQ4 L, BIRQ6 L, BIRQ7 L

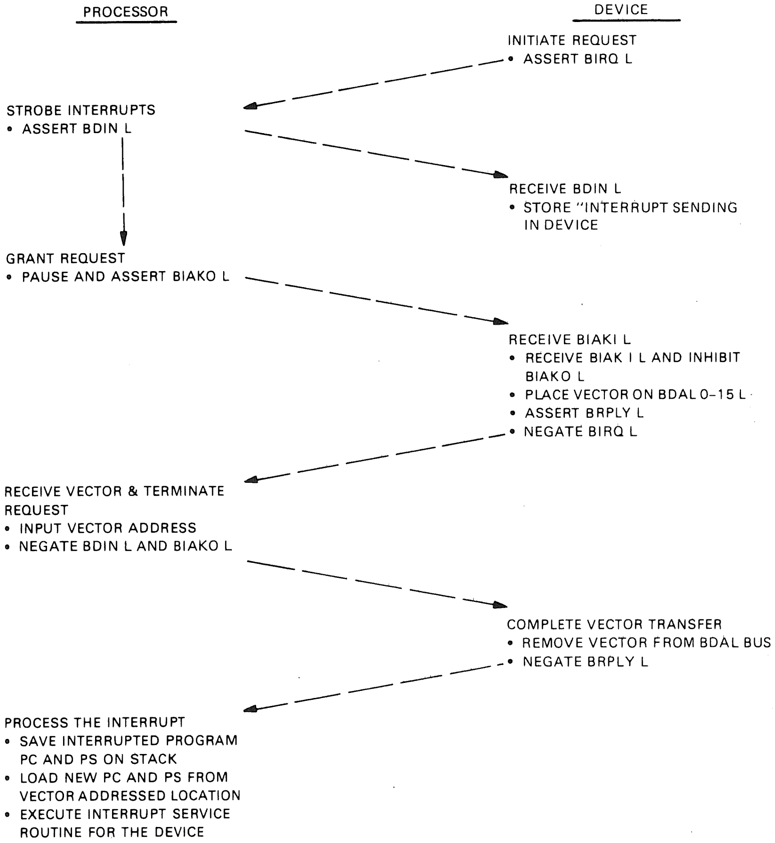
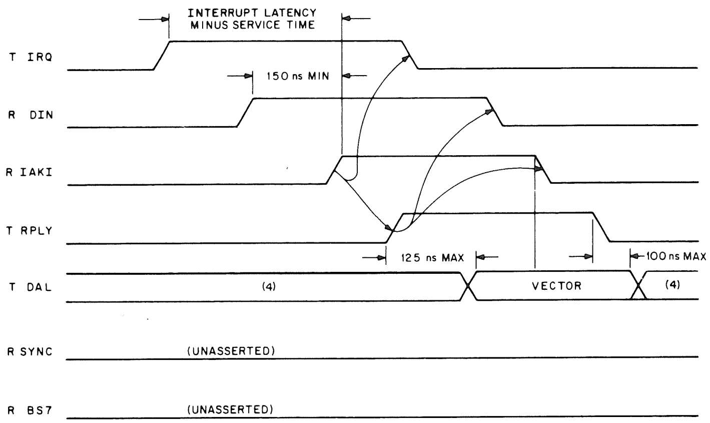


Figure E-11 Interrupt Request/Acknowledge Sequence

The interrupt request line remains asserted until the request is acknowledged.

During the interrupt acknowledge and priority arbitration phase the LSI-11/23 processor will acknowledge interrupts under the following conditions:

1. The device interrupt priority is higher than the current PS<7:5>.
2. The processor has completed instruction execution and no additional bus cycles are pending.



NOTES

1 Timing shown at Requesting Device Bus Driver Inputs and Bus Receiver Outputs

2 Signal Name Prefixes are defined below
T = Bus Driver Input
R = Bus Receiver Output

3 Bus Driver Output and Bus Receiver Input signal names include a "B" prefix.

4. Don't care condition

Figure E-12 Interrupt Protocol Timing

The processor acknowledges the interrupt request by asserting BDIN L, and 150 ns (minimum) later asserting BIAKO L. The device electrically closest to the processor receives the acknowledge on its BIAKI L bus receiver.

At this point the two types of arbitration must be discussed separately. If the device that receives the acknowledge uses the 4-level interrupt scheme, it reacts as described below:

1. If not requesting an interrupt, the device asserts BIAKO L and the acknowledge propagates to the next device on the bus.
2. If the device is requesting an interrupt, it must check to see that no higher-level device is currently requesting an interrupt. This is done by monitoring higher-level request lines. The table below lists the lines that need to be monitored by devices at each priority level.

   In addition to asserting levels 7 and 4, level 7 devices must drive level 6. This is done to simplify the monitoring and arbitration by level 4 and 5 devices. In this protocol, level 4 and 5 devices need not monitor level 7 since level 7 devices assert level 6. Level 4 and 5 devices will become aware of a level 7 request since they monitor the level 6 request. This protocol has been optimized for level 4, 5, and 6 devices, since level 7 devices very seldom are necessary.

Device Priority Level	Line(s) Monitored
4	BIRQ5, BIRQ6
5	BIRQ6
6	BIRQ7
7	—

3. If no higher-level device is requesting an interrupt, the acknowledge is blocked by the device. (BIAKO L is not asserted.) Arbitration logic within the device uses the leading edge of BDIN L to clock a flip-flop that blocks BIAKO L. Arbitration is won, and the interrupt vector transfer phase begins.
4. If a higher-level request line is active, the device disqualifies itself and asserts BIAKO L to propagate the acknowledge to the next device along the bus.

Signal timing must be carefully considered when implementing 4-level interrupts. Note Figure E-12.

If a single-level interrupt device receives the acknowledge, it reacts as follows:

- If not requesting an interrupt, the device asserts BIAKO L and the acknowledge propagates to the next device on the bus.
- If the device was requesting an interrupt, the acknowledge is blocked using the leading edge of BDIN L and arbitration is won. The interrupt vector transfer phase begins.

The interrupt vector transfer phase is enabled by BDIN L and BIAKI L. The device responds by asserting BRPLY L and its BDAL<15:00> L bus driver inputs with the vector address bits. The BDAL bus driver inputs must be stable within 125 ns (maximum) after BRPLY L is asserted. The processor then inputs the vector address and negates BDIN L and BIAKO L. The device then negates BRPLY L and 100 ns (maximum) later removes the vector address bits. The processor then enters the device's service routine.

NOTE

Propagation delay from BIAKI L to BIAKO L must not be greater than 500 ns per LSI-11 Bus slot.

The device must assert BRPLY L within 10 microseconds (maximum) after the processor asserts BIAKI L.

LSI-11/23 Four-Level Interrupt Configurations

If you have high-speed peripherals and desire better software performance, you can use the 4-level interrupt scheme. Both position-independent and position-dependent configurations can be used with the 4-level interrupt scheme.

The position-independent configuration is illustrated in Figure E-13. This allows peripheral devices that use the 4-level interrupt scheme to be placed in the backplane in any order. These devices must send out interrupt requests and monitor higher-level request lines as described. The level 4 request is always asserted by a requesting device regardless of priority, to allow compatibility if an LSI-11 or LSI-11/2 processor is in the same system. If two or more devices of equally high priority request an interrupt, the device physically closest to the processor will win arbitration. Devices that use the single-level interrupt scheme must be modified or placed at the end of the bus for arbitration to function properly.

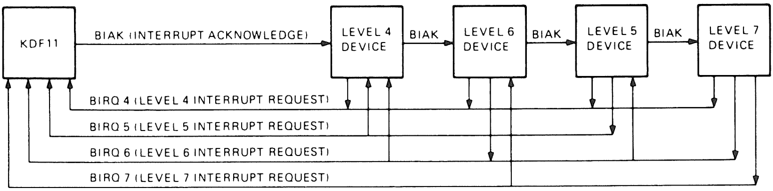

KDF11
BIAK (INTERRUPT ACKNOWLEDGE)
LEVEL 4 DEVICE
BIAK
LEVEL 6 DEVICE
BIAK
LEVEL 5 DEVICE
BIAK
LEVEL 7 DEVICE
BIRQ 4 (LEVEL 4 INTERRUPT REQUEST)
BIRQ 5 (LEVEL 5 INTERRUPT REQUEST)
BIRQ 6 (LEVEL 6 INTERRUPT REQUEST)
BIRQ 7 (LEVEL 7 INTERRUPT REQUEST)

Figure E-13 Position-Independent Configuration

The position-dependent configuration is illustrated in Figure E-14. This configuration is simpler to implement. A constraint is that peripheral devices must be inserted with the highest-priority device located closest to the processor and the remaining devices placed in the backplane in decreasing order of priority, with the lowest-priority devices farthest from the processor. With this configuration each device has to assert only its own level and level 4 (for compatibility with an LSI-11 or LSI-11/2). Monitoring higher level request lines is unnecessary. Arbitration is achieved through the physical positioning of each device on the bus. Single-level interrupt devices on level 4 should be positioned last on the bus.

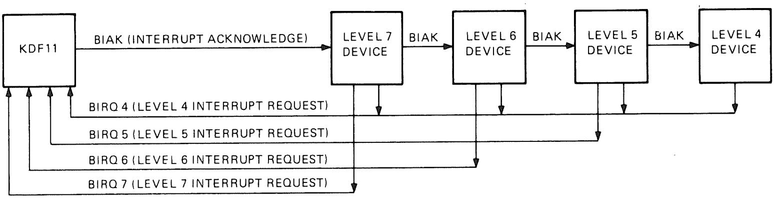

KDF11
BIAK (INTERRUPT ACKNOWLEDGE)
LEVEL 7 DEVICE
BIAK
LEVEL 6 DEVICE
BIAK
LEVEL 5 DEVICE
BIAK
LEVEL 4 DEVICE
BIRQ 4 (LEVEL 4 INTERRUPT REQUEST)
BIRQ 5 (LEVEL 5 INTERRUPT REQUEST)
BIRQ 6 (LEVEL 6 INTERRUPT REQUEST)
BIRQ 7 (LEVEL 7 INTERRUPT REQUEST)

Figure E-14 Position-Dependent Configuration

CONTROL FUNCTIONS

The following LSI-11 Bus signals provide control functions.

BREF L	Memory refresh (also block mode DMA)	
	BHALT L	Processor halt
	BINIT L	Initialize
	BPOK H	Power OK
	BDCOK H	DC power OK

Memory Refresh

If BREF is asserted during the address portion of a bus data transfer cycle, it causes all dynamic MOS memories to be addressed simultaneously. The sequence of addresses required for refreshing the memories is determined by the specific requirements for each memory. The complete memory refresh cycle consists of a series of refresh bus transactions. A new address is used for each transaction. A complete memory refresh cycle must be completed within 1 or 2 ms. Multiple data transfers by DMA devices must be avoided since they could delay memory refresh cycles. This type of refresh is done only for memories which do not perform on-board refresh.

Halt

Assertion of BHALT L for at least 25 μs interrupts the processor, which stops program execution and forces the processor unconditionally into console ODT mode.

Initialization

Devices along the bus are initialized when BINIT L is asserted. The processor can assert BINIT L as a result of executing a RESET instruction as part of a power-up or power-down sequence, or after detection of a G character in ODT. BINIT L is asserted for approximately 10 microseconds when RESET is executed.

Power Status

Power status protocol is controlled by two signals, BPOK H and BDCOK H. These signals are driven by some external device (usually the power supply).

BDCOK H — When asserted, this indicates that dc power has been stable for at least 3 ms. Once asserted, this line remains asserted until the power fails. It indicates that only 5 microseconds of dc power reserve remains.

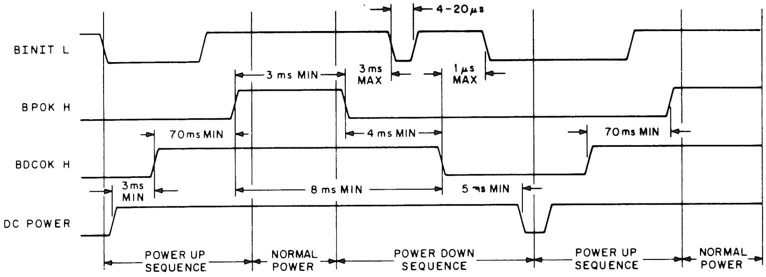
BPOK H — When asserted, this indicates that there is at least an 8 ms reserve of dc power and that BDCOK H has been asserted for at least 70 ms. Once BPOK H has been asserted, it must remain asserted for at least 3 ms. The negation of this line, the first event in the power-fail sequence, indicates that power is failing and that only 4 ms of dc power reserve remains.

Power-Up/Down Protocol

Power-up protocol begins when the power supply applies power with BDCOK H negated. This forces the processor to assert BINIT L. When the dc voltages are stable, the power supply or other external device asserts BDCOK H. The processor responds by clearing the PS, floating point status register (FPS), and floating point exception register (FEC). BINIT L is asserted for 12.6 microseconds and then negated for 110 microseconds. The processor continues to test for BPOK H until it is asserted. The power supply asserts BPOK H 70 ms (minimum) after BDCOK H is asserted. The processor then performs its power-up sequence. Normal power must be maintained at least 3.0 ms before a power-down sequence can begin.

A power-down sequence begins when the power supply negates BPOK H. When the current instruction is completed, the processor traps to a power-down routine at location 24_8. The end of the routine is terminated with a HALT instruction to avoid any possible memory corruption as the dc voltages decay.

When the processor executes the HALT instruction, it tests the BPOK H signal. If BPOK H is negated, the processor enters the power-up sequence. It clears internal registers, generates BINIT L, and continues to check for the assertion of BPOK H. If it is asserted and dc voltages are still stable, the processor will perform the rest of the power-up sequence. Figure E-15 illustrates power-up/P ower-down timing.



NOTE
Once a power down sequence is started, it must be completed before a power-up sequence is started

Figure E-15 Power-Up/P ower-Down Timing

LSI-11 BUS ELECTRICAL CHARACTERISTICS

Signal Level Specification

Input Logic Levels

TTL Logical Low:	0.8 Vdc maximum
TTL Logical High:	2.0 Vdc minimum

Output Logic Levels

TTL Logical Low:	0.4 Vdc maximum
TTL Logical High:	2.4 Vdc minimum

Load Definition

AC loads comprise the maximum capacitance allowed per signal line to ground. A unit load is defined as 9.35 pF of capacitance. DC loads are defined as maximum current allowed with a signal line driver asserted or unasserted. A unit load is defined as 210 μA in the unasserted state.

120 Ohm LSI-11 Bus

The electrical conductors interconnecting the bus device slots are treated as transmission lines. A uniform transmission line, terminated in its characteristic impedance, will propagate an electrical signal without reflections. Since bus drivers, receivers, and wiring connected to the bus have finite resistance and nonzero reactance, the transmission line impedance is not uniform, and introduces distortions into pulses propagated along it. Passive components of the LSI-11 Bus (such as wiring, cabling, and etched signal conductors) are designed to have a nominal characteristic impedance of 120 ohms.

The maximum length of interconnecting cable excluding wiring within the backplane is limited to 4.88 m (16 ft.).

Bus Drivers

Devices driving the 120 ohm LSI-11 Bus must have open collector outputs and meet the following specifications.

DC Specifications

Output low voltage when sinking 70 mA of current: 0.7V maximum.

Output high leakage current when connected to 3.8 Vdc: 25 μA (even if no power is applied, except for BDCOK H and BPOK H).

These conditions must be met at worst-case supply voltage, temperature, and input signal levels.

AC Specifications

Bus driver output pin capacitive load: Not to exceed 10 pF.

Propagation delay: Not to exceed 35 ns.

Skew (difference in propagation time between slowest and fastest gate): Not to exceed 25 ns.

Rise/Fall Times: Transition time (from 10% to 90% for positive transition, and from 90% to 10% for negative transition) must be no faster than 10 ns.

Bus Receivers

Devices that receive signals from the 120 ohm LSI-11 Bus must meet the following requirements.

DC Specifications

Input low voltage (maximum): 1.3V.

Input high voltage (minimum): 1.7V.

Maximum input current when connected to 3.8 Vdc: 80 μA even if no power is applied.

These specifications must be met at worst-case supply voltage, temperature, and output signal conditions

AC Specifications

Bus receiver input pin capacitance load: Not to exceed 10 pF.

Propagation delay: Not to exceed 35 ns.

Skew (difference in propagation time between slowest and fastest gate): Not to exceed 25 ns.

Bus Termination

The 120 ohm LSI-11 Bus must be terminated at each end by an appropriate terminator, as illustrated in Figure E-16. This is to be done as a voltage divider with its Thevenin equivalent equal to 120 ohms and 3.4V nominal. This type of termination is provided by an REV11-A refresh/boot/terminator, BDV11-AA, KPV11-B, TEV11, or by certain backplanes and expansion cards.

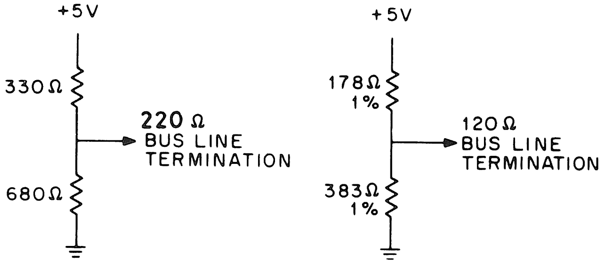


Figure E-16 Bus Line Terminations

Each of the several LSI-11 Bus lines (all signals whose mnemonics start with the letter B) must see an equivalent network with the following characteristics at each end of the bus:

Input impedance (with respect to ground)	120 ohm +5%, −15%
Open circuit voltage	3.4 Vdc +5%
Capacitance Load	Not to exceed 30 pF

NOTE

> The resistive termination may be provided by the combination of two modules (i.e., the processor module supplies 220 ohms to ground. This, in parallel with another 220 ohm card provides 120 ohms.) Both of these terminators must be physically resident within the same backplane.

Bus Interconnecting Wiring

Backplane Wiring — The wiring that connects all device interface slots on the LSI-11 must meet the following specifications:

1. The conductors must be arranged such that each line exhibits a characteristic impedance of 120 ohms (measured with respect to the bus common return).
2. Crosstalk between any two lines must be no greater than 5%. Note that worst-case crosstalk is manifested by simultaneously driving all but one signal line and measuring the effect on the undriven line.
3. DC resistance of the signal path, as measured between the near-end terminator and the far-end terminator module (including all intervening connectors, cables, backplane wiring, connector-module etch, etc.) must not exceed 2 ohms.
4. DC resistance of common return path, as measured between the near-end terminator and the far-end terminator module (including all intervening connectors, cables, backplane wiring, connector-module etch, etc.) must not exceed an equivalent of 2 ohms per signal path. Thus, the composite signal return path dc resistance must not exceed 2 ohms divided by 40 bus lines, or 50 milliohms. Note that although this common return path is nominally at ground potential, the conductance must be part of the bus wiring. The specified low impedance return path must be provided by the bus wiring as distinguished from the common system or power ground path.

Intra-Backplane Bus Wiring — The wiring that connects the bus connector slots within one contiguous backplane is part of the overall bus transmission line. Owing to implementation constraints, the nominal characteristic impedance of 120 ohms may not be achievable. Distributed wiring capacitance in excess of the amount required to achieve the nominal 120 ohm impedance may not exceed 60 pF per signal line per backplane.

Power and Ground — Each bus interface slot has connector pins assigned for the following dc voltages. The maximum allowable current per pin is 1.5 A. +5 Vdc must be regulated to ±5% with a maximum ripple of 100 mV pp. +12 Vdc must be regulated to ±3% with a maximum ripple of 200 mV pp.

- +5Vdc—Three pins (4.5 A maximum per bus device slot)
- +12 Vdc—Two pins (3.0 A maximum per bus device slot)
- Ground—Eight pins (shared by power return and signal return)

NOTE

Power is not bused between backplanes on any interconnecting bus cables.

SYSTEM CONFIGURATIONS

LSI-11 Bus systems can be divided into two types:

1. Systems containing one backplane
2. Systems containing multiple backplanes

Before configuring any system, three characteristics for each module in the system must be known. These characteristics are:

- Power consumption— +5 Vdc and +12 Vdc current requirements.
- AC bus loading—the amount of capacitance a module presents to a bus signal line. AC loading is expressed in terms of ac loads where one ac load equals 9.35 pF of capacitance.
- DC bus loading—the amount of dc leakage current a module presents to a bus signal when the line is high (undriven). DC loading is expressed in terms of dc loads where one dc load equals 210 microamperes (nominal).

Power consumption, ac loading, and dc loading specifications for each module are included in the *Microcomputer Interface Handbook*.

NOTE

The ac and dc loads and the power consumption of the processor module, terminator module, and backplane must be included in determining the total loading of a backplane.

Rules for Configuring Single Backplane Systems

- When using a processor with 220 ohm termination, the bus can accomodate modules that have up to 20 ac loads (total) before additional termination is required. If more than 20 ac loads are included, the other end of the bus must be terminated with 120 ohms, and then up to 35 ac loads may be present.
- With 120 ohm processor termination, up to 35 ac loads can be used without additional termination. If 120 ohm bus termination is added, up to 45 ac loads can be configured in the backplane.
- The bus can accommodate modules up to 20 dc loads (total).
- The bus signal lines on the backplane can be up to 35.6 cm (14 in.) long.

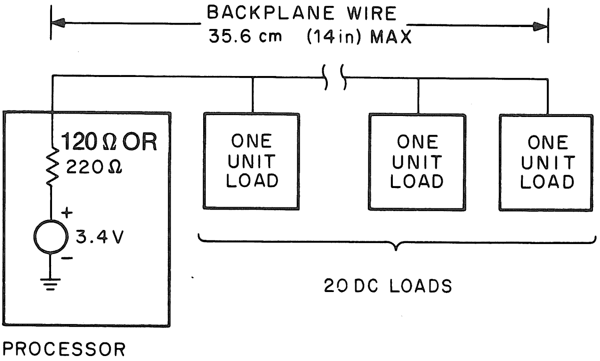
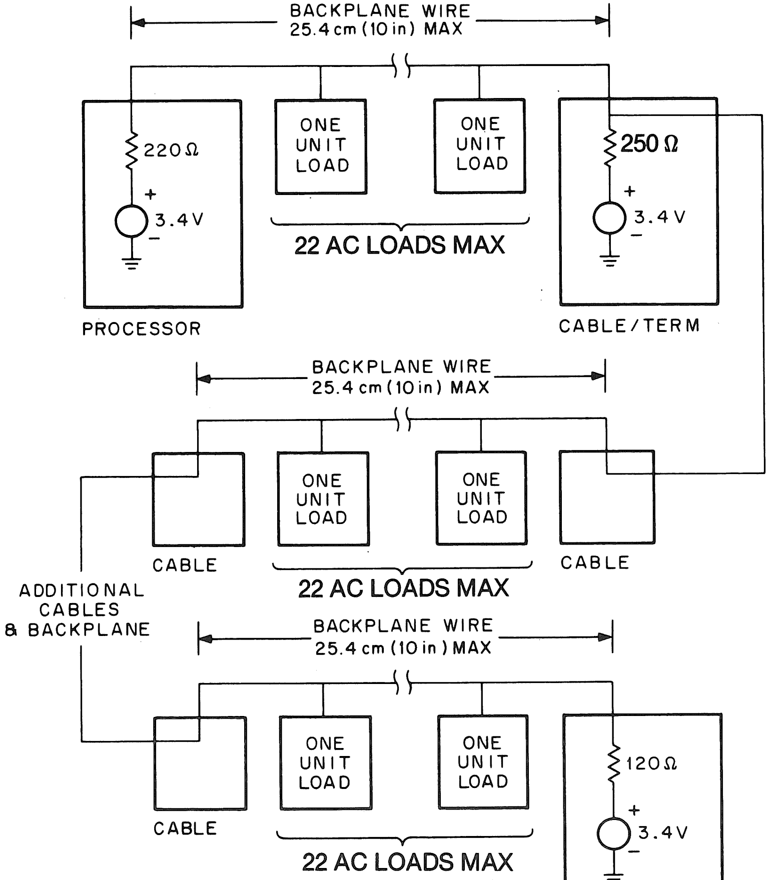


Figure E-17 Single Backplane Configuration

Rules for Configuring Multiple Backplane Systems

- As illustrated in Figure E-18, up to three backplanes may make up the system.
- The signal lines on each backplane can be up to 25.4 cm (10 in.) long.
- Each backplane can accommodate modules that have up to 22 ac loads (total). Unused ac loads from one backplane may not be added to another backplane if the second backplane loading will exceed 22 ac loads. It is desirable to load backplanes equally, or with the highest ac loads in the first and second backplanes.
- DC loading of all modules in all backplanes cannot exceed 20 loads (total).
- Both ends of the bus must be terminated with 120 ohms. This means that the first and last backplane must have an impedance of 120 ohms. To achieve this, each backplane may be lumped together as a single point. The resistive termination may be provided by a combination of two modules in the backplane—the processor providing 220 ohms to ground in parallel with an expansion paddle card providing 250 ohms to give the needed 120 ohm termination. Alternately, a processor with 120 ohm termination would need no additional termination on the paddle card to attain 120 ohms in the first box. The 120 ohm termination in the last box can be provided in two ways. The termination resistors may reside either on the expansion paddle card or on a bus termination card such as the BDV11.

- The cable(s) connecting the first two backplanes are 61 cm (2 ft.) or greater in length.
- The cable(s) connecting the second backplane to the third backplane are 122 cm (4 ft.) longer or shorter than the cable(s) connecting the first and second backplanes.



NOTES:

1. TWO CABLES (MAX.) 4.88m (16 ft) (MAX.) TOTAL LENGTH.
2. 20 DC LOADS TOTAL (MAX)

Figure E-18 Multiple Backplane Configuration

- The combined length of both cables cannot exceed 4.88 m (16 ft.).
- The cables used must have a characteristic impedance of 120 ohms.

Power Supply Loading

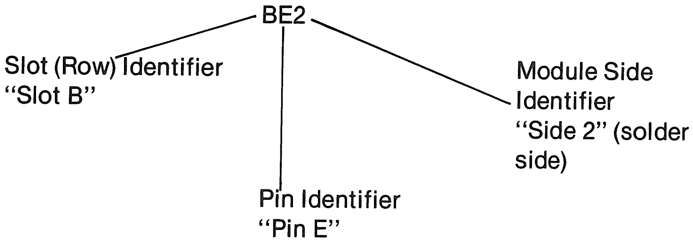
Total power requirements for each backplane can be determined by obtaining the total power requirements for each module in the backplane. Obtain separate totals for +5V and +12V power. Power requirements for each module are specified in the *Microcomputer Interfaces Handbook*.

When distributing power in multiple backplane systems, do not attempt to distribute power via the LSI-11 Bus cables. Provide separate, appropriate power wiring from each power supply to each backplane. Each power supply should be capable of asserting BPOK H and BDCOK H signals according to bus protocol; this is required if automatic power-fail/restart programs are implemented, or if specific peripherals require an orderly power-down halt sequence. The proper use of BPOK H and BDCOK H signals is strongly recommended.

MODULE CONTACT FINGER IDENTIFICATION

DIGITAL plug-in modules all use the same contact finger (pin) identification system. The LSI-11 Bus is based on the use of double-height modules that plug into a 2-slot bus connector. Each slot contains 36 lines (18 each on component and solder sides of circuit board).

Slots, shown as row A and row B in Figure E-19, include a numeric identifier for the side of the module. The component side is designated side 1 and the solder side is designated side 2. Letters ranging from A through V (excluding G, I, O, and Q) identify a particular pin on a side of a slot. Table E-4 lists and identifies the bus pins of the double-height module. The bus pin identifier ending with a 1 is found on the component side of the board, while a bus pin identifier ending with a 2 is found on the solder side of the board. A typical pin is designated as follows.



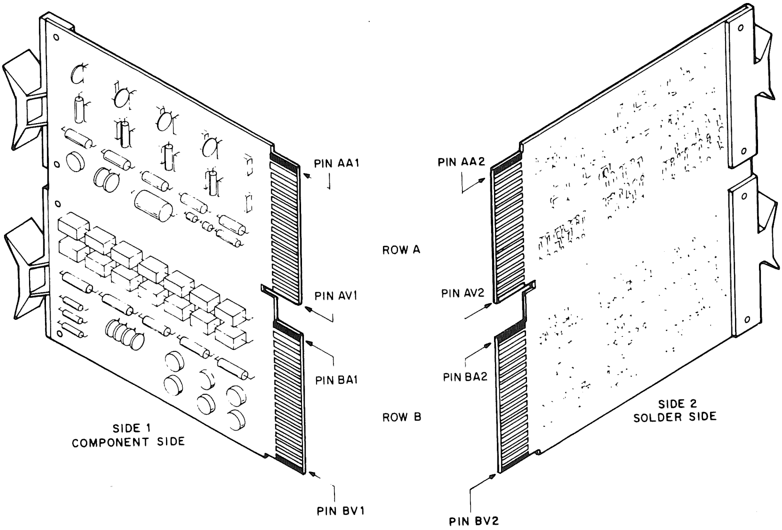


Figure E-19 Double-Height Module Contact Finger Identification

The positioning notch between the two rows of pins mates with a protrusion on the connector block for correct module positioning.

Table E-4 Bus Pin Identifiers

BUS PIN	MNEMONICS	DESCRIPTION
AA1	BIRQ5 L	Interrupt Request Priority Level 5
AB1	BIRQ6 L	Interrupt Request Priority Level 6
AC1	BDAL16 L	Extended address bit during addressing protocol; memory error data line during data transfer protocol.
AD1	BDAL17 L	Extended address bit during addressing protocol; memory error logic enable during data transfer protocol.

BUS PIN	MNEMONICS	DESCRIPTION
AE1	SSPARE1 (Alternate +5B)	Special Spare—not assigned or bused in DIGITAL cable or backplane assemblies; available for user connection. Optionally, this pin may be used for +5V battery (+5B) backup power to keep critical circuits alive during power failures. A jumper is required on LSI-11 Bus options to open (disconnect) the +5B circuit in systems that use this line as SSPARE1.
AF1	SSPARE2	Special Spare—not assigned or bused in DIGITAL cable or backplane assemblies; available for user interconnection. In the highest-priority device slot, the processor may use this pin for a signal to indicate its RUN state.
AH1	SSPARE 3 SRUN simultaneously	Special Spare—not assigned or bused in DIGITAL cable or backplane assemblies; available for user interconnection. An alternate SRUN signal may be connected in the highest-priority set.
AJ1	GND	Ground—System signal ground and dc return.
AK1	MSPAREA	Maintenance Spare—Normally connected together on the backplane at each option location (not bused connection).
AL1	MSPAREB	Maintenance Spare—Normally connected together on the backplane at each option location (not bused connection).

BUS PIN	MNEMONICS	DESCRIPTION
AM1	GND	Ground—System signal ground and dc return.
AN1	BDMR L	Direct Memory Access (DMA) Request—A device asserts this signal to request bus mastership. The processor arbitrates bus mastership between itself and all DMA devices on the bus. If the processor is not bus master (it has completed a bus cycle and BSYNC L is not being asserted by the processor), it grants bus mastership to the requesting device by asserting BDMGO L. The device responds by negating BDMR L and asserting BSACK L.
AP1	BHALT L	Processor Halt—When BHALT L is asserted for at least 25 μs, the processor services the halt interrupt and responds by halting normal program execution. External interrupts are ignored but memory refresh interrupts in LSI-11 are enabled if W4 on M7264 and M7264-YA processor modules is removed and DMA request/grant sequences are enabled. The processor executes the ODT microcode and the console device operation is invoked.
AR1	BREF L	Memory Refresh—Asserted by a DMA device. This signal forces all dynamic MOS memory units requiring bus refresh signals to be activated for each BSYNC L/BDIN L bus transaction. It is also used as a control signal for block mode DMA.

BUS PIN	MNEMONICS	DESCRIPTION
		CAUTION The user must avoid multiple DMA data transfers (burst or "hog" mode) that could delay refresh operation if using DMA refresh. Complete refresh cycles must occur once every 1.6 msec if required.
AS1	+12B or + 5B	+12 Vdc or +5V battery backup power to keep critical circuits alive during power failures. This signal is not bused to BS1 in all DIGITAL backplanes. A jumper is required on all LSI-11 Bus options to open (disconnect) the backup circuit from the bus in systems that use this line at the alternate voltage.
AT1	GND	Ground—System signal ground and dc return.
AU1	PSPARE 1	Spare (Not assigned. Customer usage not recommended.) Prevents damage when modules are inserted upside down.
AV1	+5B	+5V Battery Power—Secondary +5V power connection. Battery power can be used with certain devices.
BA1	BDCOK H	DC Power OK—Power supply-generated signal that is asserted when there is sufficient dc voltage available to sustain reliable system operation.
BB1	BPOK H	Power OK—Asserted by the power supply 70 ms after BDCOK negated when ac pow-

BUS PIN	**MNEMONICS**	**DESCRIPTION**
		er drop below the value required to sustain power (approximately 75% of nominal). When negated during processor operation, a power fail trap sequence is initiated.
BC1	SSPARE4 BDAL 18L (22-bit only)	Special Spare in the LSI-11 Bus—Not assigned. Bussed in 22-bit cable and backplane assemblies; available for use interconnection.
BD1	SSPARE5 BDAL 19L (22-bit only)	Caution. These pins may be used as test points by DIGITAL in some options.
BE1	SSPARE6 BDAL 20L	In the 22-bit LSI-11 Bus, these bussed address lines are Address Lines <21:18> currently not used during data time.
BF1	SSPARE7 BDAL 21L	In the 22-bit LSI-11 Bus these bussed address lines are Address Lines <21:18> currently not used during data time.
BH1	SSPARE8	Special Spare—Not assigned or bused in DIGITAL cable and backplane assemblies; available for user interconnection.
BJ1	GND	Ground—System signal ground and dc return.
BK1	MSPAREB	Maintenance Spare—Normally connected together on the backplane at each option location (not a bused connection).
BL1	MSPAREB	
BM1	GND	Ground—System signal ground and dc return.

BUS PIN	MNEMONICS	DESCRIPTION
BN1	BSACK L	This signal is asserted by a DMA device in response to the processor's BDMGO L signal, indicating that the DMA device is bus master.
BP1	BIRQ7 L	Interrupt request priority level 7
BR1	BEVNT L	External Event Interrupt Request—When asserted, the processor responds by entering a service routine via vector address 100_8. A typical use of this signal is a line time clock interrupt.
BS1	+12B	+12 Vdc battery backup power (not bused to AS1 in all DIGITAL backplanes).
BT1	GND	Ground—System signal ground and dc return.
BU1	PSPARE2	Power Spare 2 (not assigned a function, not recommended for use). If a module is using −12V (on pin AB2) and if the module is accidentally inserted upside down in the backplane, −12 Vdc appears on pin BU1.
BV1	+5	+5V Power—Normal +5 Vdc system power.
AA2	+5	+5V Power—Normal +5 Vdc system power.
AB2	−12	−12V Power— −12 Vdc (optional) power for devices requiring this voltage. **NOTE** LSI-11 modules which require negative voltages contain an inverter circuit (on each module) which generates the re-

BUS PIN	MNEMONICS	DESCRIPTION
		quired voltage(s). Hence, −12V power is not required with DIGITAL-supplied options.
AC2	GND	Ground—System signal ground and dc return.
AD2	+12	+12V Power—12 Vdc system power.
AE2	BDOUT L	Data Output—BDOUT, when asserted, implies that valid data is available on BDAL <0:15> L and that an output transfer, with respect to the bus master device, is taking place. BDOUT L is deskewed with respect to data on the bus. The slave device responding to the BDOUT L signal must assert BRPLY L to complete the transfer.
AF2	BRPLY L	Reply—BRPLY L is asserted in response to BDIN L or BDOUT L and during IAK transactions. It is generated by a slave device to indicate that it has placed its data on the BDAL bus or that it has accepted output data from the bus.
AH2	BDIN L	Data Input—BDIN L is used for two types of bus operation: When asserted during BSYNC L time, BDIN L implies an input transfer with respect to the current bus master, and requires a response (BRPLY L). BDIN L is asserted when the master device is ready to accept data from a slave device. When asserted without BSYNC

BUS PIN	MNEMONICS	DESCRIPTION
		L, it indicates that an interrrupt operation is occurring. The master device must deskew input data from BRPLY L.
AJ2	BSYNC L	Synchronize—BSYNC L is asserted by the bus master device to indicate that it has placed an address on BDAL<0:17> L. The transfer is in process until BSYNC L is negated.
AK2	BWTBT L	Write/Byte—BWTBT L is used in two ways to control a bus cycle: It is asserted at the leading edge of BSYNC L to indicate that an output sequence is to follow (DATO or DATOB), rather than an input sequence. It is asserted during BDOUT L, in a DATOB bus cycle, for byte addressing.
AL2	BIRQ4 L	Interrupt Request Priority Level 4— A level 4 device asserts this signal when its interrupt enable and interrupt request flips-flops are set. If the PS word bit 7 is 0, the processor responds by acknowledging the request by asserting BDIN L and BIAKO L.
AM2 AN2	BIAKI L BIAKO L	Interrupt Acknowledge—In accordance with interrupt protocol, the processor asserts BIAKO L to acknowledge receipt of an interrupt. The bus transmits this to BIAKI L of the

BUS PIN	MNEMONICS	DESCRIPTION
		device electrically closest to the processor. This device accepts the interrupt acknowledge under two conditions: 1) The device requested the bus by asserting BIRQXL, and 2) the device has the highest-priority interrupt request on the bus at that time. If these conditions are not met, the device asserts BIAKO L to the next device on the bus. This process continues in a daisy-chain fashion until the device with the highest-interrupt priority receives the interrupt acknowledge signal.
AP2	BBS7 L	Bank 7 Select—The bus master asserts this signal to reference the I/O page (including that portion of the I/O page reserved for nonexistent memory). The address in BDAL<0:12> L when BBS7 L is asserted is the address within the I/0 page.
AR2 AS2	BDMGI L BDMGO L	Direct Memory Access Grant—The bus arbitrator asserts this signal to grant bus mastership to a requesting device, according to bus mastership protocol. The signal is passed in a daisy-chain from the arbitrator (as BDMGO L) through the bus to BDMGI L of the next priority device (electrically closest device on the bus). This device accepts the grant only if it requested to be bus master (by a BDMR L). If

BUS PIN	MNEMONICS	DESCRIPTION
		not, the device passes the grant (asserts BDMGO L) to the next device on the bus. This process continues until the requesting device acknowledges the grant. **CAUTION** DMA device transfers must not interfere with the memory refresh cycle.
AT2	BINIT L	Initialize—This signal is used for system reset. All devices on the bus are to return to a known, initial state; i.e., registers are reset to zero, and logic is reset to state 0. Exceptions should be completely documented in programing and engineering specifications for the device.
AU2 AV2	BDALO L BDAL1 L	Data/Address lines—These two lines are part of the 16-line data/address bus over which address and data information are communicated. Address information is first placed on the bus by the bus master device. The same device then either receives input data from, or outputs data to the addressed slave device or memory over the same bus lines.
BA2	+5	+5V Power—Normal +5 Vdc system power.
BB2	–12	*–12V Power— –12 Vdc (optional) power for devices requiring this voltage.

* Voltages normally not supplied by DIGITAL.

BUS PIN	MNEMONICS	DESCRIPTION
BC2	GND	Ground—System signal ground and dc return.
BD2	+12	+12V Power—+12V system power.
BE2	BDAL2 L	Data/Address Lines—These 14 lines are part of the 16-line data/address bus previously described.
BF2	BDAL3 L	
BH2	BDAL4 L	
BJ2	BDAL5 L	
BK2	BDAL6 L	
BL2	BDAL7 L	
BM2	BDAL8 L	
BN2	BDAL9 L	
BP2	BDAL10 L	
BR2	BDAL11 L	
BS2	BDAL12 L	
BT2	BDAL13 L	
BU2	BDAL14 L	
BV2	BDAL15 L	

APPENDIX F
PROGRAMMING TECHNIQUES

The LSI-11 and PDP-11 microcomputers offer you a great deal of programming flexibility and power. With the combination of the instruction set, addressing modes, and programming techniques, new software can be developed and old programs utilized effectively. This chapter provides programming techniques examples which illustrate the unique capabilities of PDP-11 processors. The techniques specifically discussed are: stacks, subroutine linkage, and reentrancy.

STACKS

The stack is part of the basic design architecture of the LSI-11 and PDP-11 processors. 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 require only instructions capable of operating on bytes rather than full words. Figure F-1 illustrates both 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
```

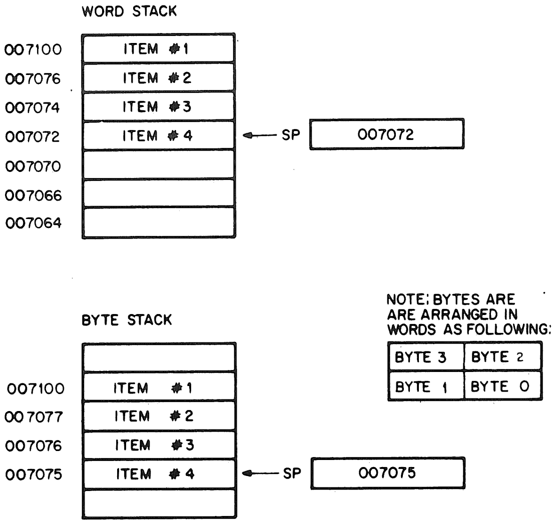


Figure F-1 Word and Byte Stacks

Data are 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 F-2 illustrates the 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.

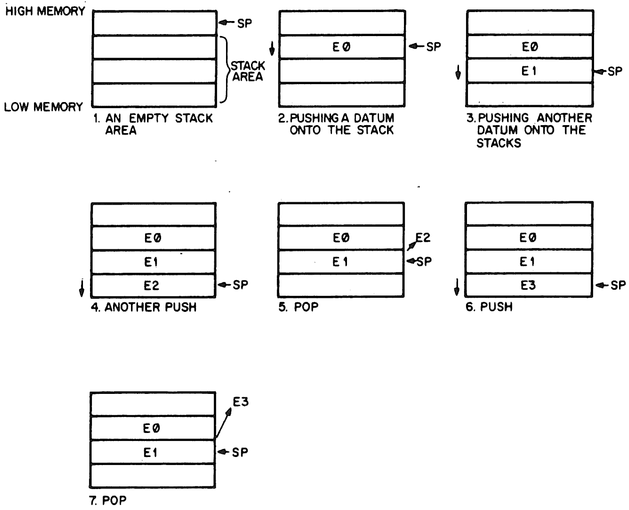


Figure F-2 Push and Pop Operations

- 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 (PSW) 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:

```
                                  Assembler
Address          Octal Code       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           016702           MOV TEMP2,R2    ;restore R2
076416           000004           *
076420           000207           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.

```
                                  Assembler
Address          Octal Code       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) instruction. This example is illustrated in Figure F-3.

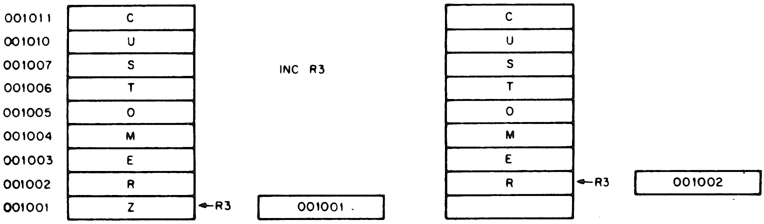


Figure F-3 Byte Stack Used as a Character Buffer

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

DELETING ITEMS FROM A STACK

To delete one item:

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

To delete two items:

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

To delete fifty items from a word stack:

ADD #100.,SP

SUBROUTINE LINKAGE

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

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

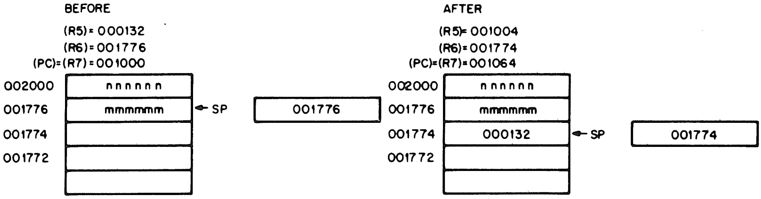


Figure F-4 JSR Instruction

Because the PDP-11 hardware already uses general purpose register R6 to point to a stack for saving and restoring PC and PSW (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.

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.

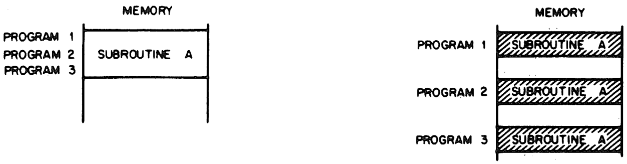


Figure F-5 Reentrant Routines

PDP-11 Approach

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

Conventional Approach

A separate copy of Subroutine A must be provided for each program.

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:

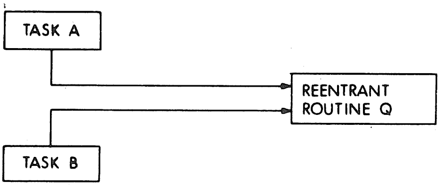


Figure F-6 Sharing Control of a Routine

- 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 PSW, 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.

GLOSSARY

abort An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state such that the instruction cannot necessarily be restarted.

absolute address A binary number that is permanently assigned as the address of a storage location.

absolute mode Autoincrement deferred mode in which the PC is used as the register. The PC contains the address of the location containing the actual operand.

access mode 1. Any of the three processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected: kernel, supervisor, and user. When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. In any other mode, the processor is inhibited from executing privileged instructions. The Processor Status Word contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the executive runs in kernel mode and is most protected. The debugger runs in user mode and is not more protected than normal users programs.

access time The time interval between the instant at which data is called for (or requested to be stored) from a storage device and the instant delivery (or storage) is started.

access type 1. The way in which the processor accesses instruction operands. Access types are: read and write. 2. The way in which a procedure accesses its arguments.

access violation An attempt to reference an address that is not mapped into virtual memory or an attempt to reference an address that is not accessible by the current access mode.

accumulator A 16-bit register or memory location in which the result of an operation is formed.

active release Pertains to the bus. Indicates that the bus control is passed from the bus master to the processor by means of an interrupt operation. See "passive release".

address A number used by the operating system and user software to identify a storage location. See also *virtual address* and *physical address*.

address field That portion of a computer word either containing the address of the operand or containing the information necessary for calculation of the address.

address map A table, chart or drawing showing the absolute addresses of all locations in the core memory.

addressing mode The way in which an operand is specified; for example, the way in which the effective address of an instruction operand is calculated using the general registers. The basic general register addressing modes are called register, register deferred, autoincrement, autoincrement deferred, autodecrement, autodecrement deferred, index, and index deferred. The Program Counter (PC) addressing modes are called immediate (for register deferred mode using the PC), absolute (for autoincrement deferred mode using the PC), relative, and relative deferred.

address space The set of all possible addresses available to a process. Virtual address space refers to the set of all possible virtual addresses. Physical address space refers to the set of all possible physical addresses.

algorithm A prescribed set of well-defined rules or processes for the solution of a problem in a definite sequence.

alphanumeric character An upper or lower case letter (A to Z, a to z), a dollar sign ($), an underscore (__), or a decimal digit (0 to 9).

American Standard Code for Information Interchange (ASCII) A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, control, and other special symbols used in text representation and communications protocol.

ASCII See American Standard Code for Information Interchange.

assemble To translate from a symbolic program to a binary program by substituting binary operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assembler A program that performs the translation from symbolic program to binary program.

autodecrement mode In autodecrement mode addressing, the contents of the selected register are decremented, and the result is used as the address of the actual operand of the instruction. The contents of the register are decremented according to the data type context of the register: 1 for byte, 2 for word, 4 for single-precision floating, and 8 for double-precision floating.

autoincrement deferred mode In autoincrement deferred mode addressing, the specified register contains the address of a word which contains the address of the actual operand. The contents of the register are incremented by one, two, four, or eight, depending on the data type. If the PC is used as the register, this mode is called absolute mode.

autoincrement mode In autoincrement mode addressing, the contents of the specified register are used as the address of the operand; then the contents of the register are incremented by the size of the operand (unless the PC is used, in which case it is always incremented by 2). If the PC is used, this is called immediate mode.

base operand address The address of the base of a table or array referenced by index mode addressing.

base register A general register used to contain the address of the first entry in a list, table, array, or other data structure.

bidirectional Capable of traveling in either direction. Refers to UNIBUS or LSI-11 Bus lines on which signals can be transmitted or received.

binary Pertaining to a number system with a radix of 2.

binary digit One of two states (0 or 1) of the binary number system. Usually referred to as a bit.

bit A shortened form of binary digit; the smallest unit of information.

bit complement (also called *one's complement*) The result of exchanging 0s and 1s in the binary representation of a number. Thus, the bit complement of the binary number 11011001 (217_{10}) is 00100110. Bit complements are used in place of their corresponding binary numbers in some arithmetic computations in computers.

block 1. The smallest addressable unit of data that the specified device can transfer in an I/O operation (512 contiguous bytes for most disk devices) 2. An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information.

block transfer Moving a large amount of data in one operation. For example: data from a disk into memory or vice versa.

bootstrap To start the computer and bring it to the desired state by means of its own action. For example, a routine whose first few instructions are sufficent to bring the rest of itself into the computer from an input device.

bootstrap loader A program that is toggled into the computer to allow a small set of programs in a special tape format to be loaded into the PDP-11.

boundary See "word boundaries."

branch A point in a routine where one of two or more choices is made under control of the routine. The PDP-11 has many branch instructions and one unconditional branch instruction.

buffer A storage device used to compensate for a difference in rate of data flow or time of event occurrence when transmitting data from one device to another.

buffer register see buffer.

bus See LSI-11 Bus and UNIBUS.

bus address The current address on the bus; may be the address of a device, the processor, or a memory location.

bus address register A processor register that holds the address from the process for display and then loads it onto the bus at the required time.

bus device Any external device, including core memory, that is connected to the bus and has an assigned device address and/or priority level.

bus driver A circuit or module used to pass signals to the bus in accordance with the transmission line characteristics of the bus.

bus master The bus device that has control of the bus.

bus receiver A circuit or module used to receive signals from the bus. These circuits use gates with high input impedance and proper logic thresholds to ensure that the received signal is compatible with the rest of the system.

bus request A request from a peripheral for control of the bus in order to become bus master and initiate an interrupt or perform a data transfer.

bus slave The peripheral that is communicating with the bus master.

bus transceiver A module containing both bus driver and receiver circuits.

byte A byte is eight contiguous bits starting on an addressable byte boundary. Bits are numbered from the right, 0 through 7, with bit 0 the low-order bit. When interpreted arithmetically, a byte is a two's complement integer with significance increasing from bits 0 through 6. Bit 7 is the sign bit. The value of the signed integer is in the range -128 to 127 decimal. When interpreted as an unsigned integer, significance increases from bits 0 through 7 and the value of the unsigned integer is in the range 0 to 255 decimal. A byte can be used to store one ASCII character.

cache memory A small, high-speed memory placed between slower main memory and the processor. A cache increases effective memory transfer rates and processor speed. It contains copies of data recently used by the processor and may fetch several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes.

call To transfer control to a specified routine.

calling sequence A specified set of instructions and necessary data requried to call a given routine.

character A symbol represented by an ASCII code. See also *alphanumeric character*.

character string A contiguous set of bytes. A character string is identified by two attributes: an address and a length. Its address is the address of the byte containing the first character of the string. Subsequent characters are stored in bytes of increasing addresses. The length is the number of characters in the string.

character string descriptor A data structure used for passing character data (strings). The first word contains the length of the character string. The second word contains the address of the string.

carry In performing binary addition, one bit of information often has to be carried from one digit of the addition to the next most significant digit. This operation is referred to as a "carry".

carry bit Indicates that an operation resulted in a carry from the most significant bit. During subtraction, indicates a borrow from bit 16.

central processor See "processor".

clear To erase the contents of a storage location by replacing the contents with zeros; to set register and/or flip-flops in a device to the required initial states.

clock A device that generates regular, periodic signals for synchronization.

coding To write instructions for a computer using symbols meaningful to the computer or to an assembler, compiler, etc.

command An instruction, generally an English word, typed by the user at a terminal or included in a command file, which requests the software monitoring a terminal or reading a command file to perform some well-defined activity. For example, typing the COPY command requests the system to copy the contents of one file into another file.

command procedure A file containing commands and data that the command interpreter can accept in lieu of the user's typing the commands individually on a terminal.

compiler A program that produces a binary-coded program from a source (symbolic) program.

complement The binary opposite of a number, variable, or function. See "one's complement" and "two's complement".

condition An exception condition detected and declared by software.

condition codes Four bits in the Processor Status Word (PSW) that indicate the results of previously executed instructions.

condition handler A procedure that a process wants the system to execute when an exception condition occurs. The operating system searches for a condition handler and, if it is found, initiates the handler immediately. The condition handler may perform some act to change the situation that caused the exception condition and continue execution for the process that incurred the exception condition. Condition handlers execute in the context of the process at the access mode of the code that incurred the exception condition.

conditional jump A jump that occurs only if specified criteria have been met.

console The manual control unit integrated into the central processor. The console may include a microprocessor and a serial line interface connected to a terminal. It enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal The terminal connected to the central processor console or the first serial-line unit.

context switching Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution. The operating system saves the interrupted process's hardware context, then loads another process's hardware context scheduling that process for execution.

control and status register A register, used with a peripheral, that contains information needed to communicate with the peripheral.

current access mode The processor access mode of the currently executing software. The Current Mode field of the Processor Status Word (PSW) indicates the access mode of the currently executing software.

data A general term used to denote any or all facts, numbers, letters, and symbols. It connotes basic elements of information which can be processed or produced by a computer.

data buffer register A register used with a peripheral to temporarily store data that is to be transferred into or out of the processor or other device.

data paths That portion of the processor where normal processing and computation accurs. All modifications and routing of data within the processor are performed by the data paths which consist primarily of the input gating and latches, adder, and output gating circuits.

data structure Any table, list, array, queue, or tree whose format and access conventions are well-defined for reference by one or more images.

data type In general, the way in which bits are grouped and interpreted. In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand. Operand data types include: byte, word, and longword integer; single-precision floating, and double-precision floating; character string; and packed decimal string.

debug To detect, locate, and remove mistakes from a program

dedicated controller A processor or computer system, usually with a read-only memory, that is designed and/or used to control only one specific process. For example, a computer designed to continually monitor, evaluate, and change a chemical process.

dedicated line A signal path used for only one purpose.

deferred address Indirectly addressed. The contents of the location is the address of the operand rather than the operand itself.

descriptor A data structure used in calling sequences for passing argument types, addresses and other optional information. See character string descriptor.

device Usually refers to an external device which is synonymous with the term "peripheral".

device flag A bit in either the interface logic or the device itself that is set to indicate a specific condition such as ready or busy.

device interrupt An interrupt received on interrupt priority levels 4 through 7. Device interrupts can be requested only by devices, controllers, and memories.

device name The field in a file specification that identifies the device unit on which a file is stored. Device names also include the mnemonics that identify an I/O peripheral device in a data transfer request. A device name consists of a mnemonic followed by a controller identification letter (if applicable), followed by a unit number (if applicable). A colon (:) separates it from following fields.

device register A location in device controller logic used to request device functions (such as I/O transfers) and/or to report status.

device selection code Part of an address that is used to specify that a particular device has been selected for use.

device unit One drive, and its controlling logic, of a mass storage device system. A mass storage system can have several drives connected to it.

diagnostic A program that tests logic and reports any faults it detects.

digit A character used to represent once of the non-negative integers smaller than the radix. For example, in binary notation (radix 2), a digit is either 1 or 0.

direct address An address that specifies the location of an instruction operand.

direct address mode Any PDP-11 address mode that is not deferred.

direct mapping cache A cache organization in which only one address comparison is needed to locate any data in the cache because any block of main memory data can be placed in only one possible position in the cache. Contrast with *fully associative cache.*

direct memory access Transfer of data into memory without supervision of the processor. Data is passed directly between the memory and another device through the bus. Transfers are usually accomplished with a nonprocessor request.

disk A mass-storage device. Its basic unit is a record-like platter on which data is magnetically recorded. Types of disks include rigid, flexible (floppy), Winchester, and cartridge.

displacement deferred indexed mode An indexed addressing mode in which the base operand specifier uses displacement deferred mode addressing.

double-precision floating datum Eight contiguous bytes starting on an addressable word boundary, which are interpreted as containing a floating point number. The bits are labeled from right to left, 0 to 63. A four-word floating point number is identified by the address of the byte contain bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess —128 binary exponent. Bits 63 through

16 and 6 through 0 contain a normalized 56-bit fraction with the redundant, most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from 6 through 0, 31 through 16, 47 through 32, then 63 through 48. Exponent values of 1 through 255 in the 8-bit exponent field represent true binary exponents of —128 to 127. An exponent value of 0 together with a sign bit of 0 represents a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return an undefined operand fault. The value of a double-precision floating datum is in the approximate range (+ or −) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{55}, or sixteen decimal digits.

drive The electromechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

effective address The address obtained after indirect or indexing modifications are calculated.

entry point A location that can be specified as the object of a call.

escape sequence An escape is a transition from the normal mode of operation to a mode outside the normal mode. The escape character is the code that indicates the transition from normal to escape mode. An escape sequence refers to the set of character combinations starting with an escape character that the terminal transmits without interpretation to the software set up to handle sequences.

event A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process's ability to execute. Events can be synchronous with the process's execution (e.g., a wait request), or they can be asynchronous (e.g., I/O completion). Some other events include: swapping, and wake request.

event flag A bit in an event flag cluster that can be set or cleared to indicate the occurrence of the event associated with that flag. Event flags are used to synchronize activities in a process or among many processes.

exception An event detected by the hardware (other than an interrupt, or Jump or Branch instruction) that changes the normal flow of instruction or set of instructions (whereas an interrupt is caused by an activity in the system independent of the current instruction). There are three types of hardware exceptions; traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace traps, breakpoint instruction execution, and arithmetic traps such as overflow, underflow, and divide by zero.

exception condition A hardware- or software-detected event other than an interrupt or Jump or Branch instruction that changes the normal flow of instruction execution.

field A set of contiguous bytes in a logical record.

floating (point) datum See *single-precision floating datum*.

fully associative cache A cache organization in which any block of data from main memory can be placed anywhere in the cache. Address comparision must take place against each block in the cache to find any particular block. Constrast with *direct mapping cache*.

general register Any of the eight 16-bit registers used as the primary operands of the instructions. The general registers include 6 general purpose registers which can be used as accumulators, as counters, and as pointers to locations in main memory, and the Stack Pointer (SP), and Program Counter (PC) registers.

giga Metric term used to represent the number 1 followed by nine 0s (10^9, though in the computer industry it is often used to mean 2^{30}, which is about 7.4% larger.)

hardware context The values contained in the following registers while a process is executing: the Program Counter (PC); the Processor Status Word (PSW); the 6 general registers (R0 through R5); the Stack Pointer (SP) for the current access mode in which the processor is executing; plus the contents to be loaded in the stack pointer for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing its hardware context must be stored in memory.

image An image consists of procedures and data that have been bound together by the linker. There are three types of images: executable, shareable, and system.

immediate mode Autoincrement mode addressing in which the PC is used as the register.

index register A register used to contain an address offset.

input stream The source of commands and data. One of either the user's terminal, the batch stream, or an indirect command file.

instruction buffer A buffer in the processor used to contain bytes of the instruction currently being decoded and to prefetch instructions in the instruction stream. The control logic continously fetches data from memory to keep the buffer full.

interleaving Assigning consecutive physical memory addresses alternately between two memory controllers.

interrecord gap A blank space deliberately placed between data records on the recording surface of a magnetic tape.

interrupt An event other than a powerfail or abort that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also *device interrupt*, *software interrupt*, and *urgent interrupt*.

interrupt priority level (IPL) The interrupt level at which the processor executes when an interrupt is generated. There are 8 possible interrupt priority levels (IPL). IPL 0 is lowest, 7 highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the process is currently executing at an IPL greater than or equal to the one of the device's interrupt service routine.

interrupt service routine The routine executed when a device interrupt occurs.

interrupt vector See *vector*.

kernel mode The most privileged processor access mode (mode 0). The operating system's most privileged services, such as I/O drivers run in kernel mode.

main memory See *physical memory*.

mass storage device A device capable of reading and writing data on mass storage media such as a diskpack or a magnetic tape reel.

memory management The system functions that include the hardware's page mapping and protection.

nibble Half a byte—the low-order or high-order four bits of an 8-bit byte.

normalized fraction A numeric representation patterned on scientific notation, but in which the fraction part of the representation is greater than or equal to 0.5 and less than 1. As a binary form, such a fraction will always begin with a 1 in the leftmost (most significant) bit, unless the number is zero. Because of this, the lead 1 is not stored, and a bit-per-number saving is effected in storage.

numeric string A contiguous sequence of bytes representing up to 31 decimal digits (one per byte) and possily a sign. The numeric string is specified by its lowest addressed location, its length, and its sign representation.

offset A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

one's complement See *bit complement.*

opcode The pattern of bits within an instruction that specifies the operation to be performed.

packed decimal A method of representing a decimal number by storing a pair of decimal digits in one byte, taking advantage of the fact that only four bits are required to represent the numbers 0 through 9.

packed decimal string A contiguous sequence of up to 16 bytes interpreted as a string of nibbles. Each nibble represents a digit, except the low-order nibble of the highest addressed byte, which represents the sign. The packed decimal string is specified by its lowest addressed location and the number of digits.

page 1. A set of 8192 contiguous byte locations used as the unit of memory mapping and protection. 2. The data between the beginning of file and a page marker, between two markers, or between a marker and the end of a file.

physical address The address used by hardware to identify a location in physical memory or on directly-addressable secondary storage devices such as a disk. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

physical address space The set of all possible 22-bit physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical memory The memory modules connected to the processor that are used to store: 1) instructions that the processor can directly fetch and execute, and 2) any other data that a processor is instructed to manipulate. Also called *main memory.*

position dependent code Code that can execute properly only in the locations in virtual address space that were originally assigned to it by the linker or taskbuilder.

position independent code Code that can execute properly without modification wherever it is located in virtual address space, even if its location is changed after it has been linked. Generally, this code uses addressing modes that form an effective address relative to the PC.

privileged instructions In general, any instruction intended for use by the operating system or privileged system programs. In particular, instructions that the processor will not execute unless the current access mode is kernel mode (e.g., HALT and RESET).

procedure See *command procedure*.

process The basic entity scheduled by the system software, that provides the context in which an image executes. A process consists of an address space and both hardware and software contexts. It is loosely analogous to a job or task.

process address space See *process space*.

process context The hardware and software contexts of a process.

Processor Status Word (PSW) Processor status information includes: the condition codes (carry, overflow, zero, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

Program Counter (PC) General register 7(R7). At the beginning of an instruction's execution, the PC normally contains the address of a location in memory from which the processor will fetch the next instruction it will execute.

program locality A characteristic of a program that indicates how close or far apart the references to locations in virtual memory are over time. A program with a high degree of locality does not refer to many widely scattered virtual addresses in a short period of time. A cache memory is more effective if a program has locality.

queue 1. A linked list. 2. To make an entry in a list or table.

read access type An instruction or procedure operand attribute indicating that the specified operand is only read during instruction or procedure execution.

register A storage location in hardware logic other than main memory. See also *general register*, *processor register*, and *device register*.

register deferred mode In register deferred mode addressing, the contents of the specified register are used as the address of the actual instruction operand.

register mode In register mode addressing, the contents of the specified register are used as the actual instruction operand.

scatter/gather The ability to transfer in one I/O operation data from discontiguous pages in memory to contiguous blocks on disk, or data from contiguous blocks on disk to discontiguous pages in memory.

secondary storage Random access mass storage.

signal 1. An electrical impulse conveying information. 2. The software mechanism used to indicate that an exception condition was detected.

single-precision floating datum Four contiguous bytes starting on an addressable byte boundary. The bits are labeled from right to left 0 to 31. A two-word floating point number is identified by the address of the byte containing bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess-128 binary exponent. Bits 31 through 16 and 6 through 0 contain a normalized 24-bit fraction with the redundant, most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from bit 6 through 0, then 31 through 16. Exponent values of 1 through 255 in the 8-bit exponent field represent true binary exponents of −128 to 127. An exponent value of 0 together with a sign bit of 0 represents a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return a reserved operand fault. The value of a floating datum is in the approximate range $(+\text{ or }-)0.29 \times 10^{-38}$ to 1.7×10^{38}. The precision is approximately one part in 2^{23}, or seven decimal digits.

software interrupt An interrupt generated on interrupt priority levels through 7, which can be requested only by software.

stack An area of memory set aside for temporary storage, or for procedure and interrupt service linkages. A stack uses the last-in, first-out concept. As items are added to ("pushed on") the stack, the stack pointer decrements. As items are retrieved from ("popped off") the stack, the stack pointer increments.

Stack Pointer General register 6(R6). SP contains the address of the top (lowest address) of the processor-defined stack. Reference to SP will access one of the three possible stack pointers, kernel, supervisor, or user, depending on the value in the current mode and interrupt stack bits in the Processor Status Word (PSW).

status code A value that indicates the success or failure of a specific function. For example, system services often return a status code in the PSW's C-bit upon completion.

store through See *write through*.

string A connected sequence of entities such as characters in a command string.

subroutine A small routine, usually performing only one task, that is called frequently from various points of the main routine.

subroutine, closed A subroutine not stored in the main part of a program. Such a subroutine is entered by a jump or branch operation, and provision is made at the end of the subroutine to return control to the calling program.

subroutine, open A subroutine that must be inserted into a program at each place it is to be used.

supervisor mode The second most privileged processor access mode (mode 2).

symbolic address A set of characters used to specify a memory location within a program.

symbolic coding Writing instructions using mnemonic notation instead of actual machine language (binary) notation.

symbolic program A service program that translates symbolic programs into binary-coded programs. The programmer writes the symbolic program using symbols which are meaningful to him and the symbolic program translates the symbols into binary code which is meaningful to the computer.

synchronize To ensure that a level or pulse is presented to a system or component at the correct time.

synchronous All changes occurring simultaneously or in a definite, timed sequence.

system In the context "system, owner, group, world," *system* refers to the group numbers that are used by operating system and its controlling users, the system operators and system manager.

T bit A bit in the processor status word used in program debugging. This bit can be set or cleared under program control. If set, a processor trap occurs upon completion of the instruction.

table A collection of data in which each item is uniquely identified by its position relative to the other items, or by some other means.

terminal A device in a system through which data can either enter or leave.

time-out A specified amount of time (10 microseconds) that the system waits for a response from a referenced address. If there is no response within the specified time, an error occurs. Time-out errors are caused, in general, by attempts to reference nonexistent memory or nonexistent peripherals or words at odd addresses.

time sharing A method of allocating processor time and other computer services among multiple users so that the computer, in appearance, processes a number of programs simultaneously.

translate To convert from one language to another.

trap An unprogrammed jump to a known location, automatically activated by the hardware if certain predetermined conditions occur, such as illegal instructions, errors, etc.

two's complement A binary representation for integers in which a negative number is one greater than the bit complement of the positive number.

two-way associative cache A cache organization which has two groups of directly mapped blocks. Each group contains several blocks for each index position in the cache. A block of data from main memory can go into either group at its proper index position. A two-way associative cache is a compromise between the extremes of fully associative and direct mapping cache organizations, and it takes advantage of the features of both.

Unibus The single, high-speed bus structure shared by the processor, core memory, and all peripherals. It formed the basis for the smaller LSI-11 Bus.

unidirectional Capable of traveling in only one direction. Refers to the Unibus control transfer lines that carry signals to select the next bus master.

unit record device A device such as a card reader or lineprinter.

user mode The least privileged processor access mode. User processes and the Run Time Library procedures run in user mode.

user privileges The privileges granted a user by the system manager.

vector Two words, containing the value of the program counter and processor status word, respectively, that direct the processor to a new routine.

vector address The address of the location containing the vector words.

virtual address A 16-bit integer identifying a byte "location" in virtual address space. The memory management hardware translates a virtual address to a physical address. The term *virtual address* may also refer to the address used to identify a virtual block on a mass storage device.

virtual address space The set of all possible virtual addresses that an image executing in the context of a process can use to identify the location of an instruction of data. The virtual address space seen by the programmer is a linear array of 65,536(2^{16}) byte addresses.

wait loop A condition caused by the program WAIT instruction to allow the processor to wait for an interrupt. When the processor is in a wait loop, it does not compete for bus control by fetching instructions or operands from memory.

word 16-bits of data in the PDP-11 that is stored in two successive locations. The word address is always an even address.

word boundary The division between even numbered addresses. Since each word occupies two storage locations, words can be addressed only on even boundaries; bytes can be addressed on either even or odd boundaries.

write To transfer information from internal storage to an output device or external storage.

write access type The specified operand of an instruction or procedure written during that instruction's execution.

write allocate A cache management technique in which cache is allocated on a write miss as well as on the usual read miss.

write back A cache management techinque in which data from a write operation to cache is copied into main memory only when the data in cache must be overwritten. This results in temporary inconsistencies between cache and main memory. Contrast with *write through*.

write through A cache management technique in which data from a write operation is copied in both cache and main memory. Cache and main memory data are always consistent. Contrast with *write back*.

INDEX

NOTES

PDP-11 ARCHITECTURE HANDBOOK 1983–84

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 CARD

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 Architecture
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, Modesto, Monrovia, Oakland, Pasadena, Sacramento, San Diego, San Francisco, Santa Barbara, Santa Clara, Santa Monica, Sherman Oaks, Sunnyvale, Torrance COLORADO, Colorado Springs, Denver CONNECTICUT, Fairfield, Meriden DELAWARE, Newark, Wilmington FLORIDA, Jacksonville, Melbourne, Miami, Orlando, Pensacola, Tampa GEORGIA, Atlanta HAWAII, Honolulu IDAHO, Boise ILLINOIS, Chicago, Peoria INDIANA, Indianapolis IOWA, Bettendorf KENTUCKY, Louisville LOUISIANA, Baton Rouge, New Orleans MAINE, Portland MARYLAND, Baltimore, Odenton MASSACHUSETTS, Boston, Burlington, Springfield, Waltham MICHIGAN, Detroit, Kalamazoo MINNESOTA, Minneapolis MISSOURI, Kansas City, St. Louis NEBRASKA, Omaha NEVADA, Las Vegas, Reno 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, Eugene, Portland PENNSYLVANIA, Allentown, Harrisburg, Philadelphia, Pittsburgh RHODE ISLAND, Providence SOUTH CAROLINA, Columbia, Greenville TENNESSEE, Knoxville, Memphis, Nashville TEXAS, Austin, Dallas, El Paso, Houston, San Antonio UTAH, Salt Lake City VERMONT, Burlington VIRGINIA, Arlington, Lynchburg, Norfolk, Richmond WASHINGTON, Seattle, Spokane WASHINGTON D.C. WEST VIRGINIA, Charleston WISCONSIN, Madison, 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 ARGENTINA, Buenos Aires AUSTRALIA, Adelaide, Brisbane, Canberra, Darwin, Hobart, Melbourne, Newcastle, Perth, Sydney, Townsville, Victoria AUSTRIA, Vienna BELGIUM, Brussels BRAZIL, Rio de Janeiro, Sao Paulo CANADA, Calgary, Edmonton, Hamilton, Halifax, Kingston, London, Montreal, Ottawa, Quebec City, Regina, Toronto, Vancouver, Victoria, Winnipeg CHILE, Santiago COLOMBIA, Bogota DENMARK, Copenhagen EGYPT, Cairo ENGLAND, Basingstoke, Birmingham, Bristol, Ealing, Epsom, Leeds, Leicester, London, Manchester, Newmarket, Reading, Welwyn FINLAND, Helsinki FRANCE, Bordeaux, Lille, Lyon, Marseille, Nantes, Paris, Puteaux, Strasbourg HONG KONG INDIA, Bangalore, Bombay, Calcutta, Hyderabad, New Delhi IRELAND, Dublin ISRAEL, Tel Aviv ITALY, Milan, Padova, Rome, Turin JAPAN, Fukuoka, Nagoya, Osaka, Tokyo, Yokohama KOREA, Seoul KUWAIT, Safat MEXICO, Mexico City, Monterrey NETHERLANDS, Amsterdam, The Hague, Utrecht NEW ZEALAND, Auckland, Christchurch, Wellington NIGERIA, Lagos NORTHERN IRELAND, Belfast NORWAY, Oslo, PERU, Lima PUERTO RICO, San Juan SAUDI ARABIA, Jeddah SCOTLAND, Edinburgh REPUBLIC OF SINGAPORE, SPAIN, Barcelona, Madrid SWEDEN, Gothenburg, Malmoe, Stockholm SWITZERLAND, Geneva, Zurich TAIWAN, Taipei TRINIDAD, Port of Spain VENEZUELA, Caracas WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nuremberg, Stuttgart YUGOSLAVIA, Belgrade, Ljubljana, Zagreb

ORDER CODE: EB-23657-18