

**MACHINE
AND
ASSEMBLY
LANGUAGE
PROGRAMMING
OF THE PDP-11**

MACHINE AND ASSEMBLY LANGUAGE PROGRAMMING OF THE PDP-11

ARTHUR GILL

*Department of Electrical Engineering and Computer Sciences
University of California, Berkeley*

Library of Congress Cataloging in Publication Data

GILL, ARTHUR, (date)

Machine and assembly language programming of PDP-11.

Includes index.

1. PDP-11 (Computer)—Programming. 2. Assembler language (Computer program language) I. Title.

QA76.8.P2G54 001.6'42 78-9690

ISBN 0-13-541870-4

© 1978 by Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632

All rights reserved. No part of this book
may be reproduced in any form or by any means
without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

To Velta

CONTENTS

PREFACE *xiii*

NUMBER SYSTEMS 1

1.1	Decimal-to-Binary Conversion	2
1.2	Decimal-to-Octal Conversion	3
1.3	Binary-to-Decimal Conversion	5
1.4	Octal-to-Decimal Conversion	5
1.5	Octal-to-Binary Conversion	6
1.6	Binary-to-Octal Conversion	7
1.7	Binary and Octal Addition	7
	Exercises	8

2**THE PDP-11 ORGANIZATION 9**

- 2.1 The Central Memory 10
- 2.2 The Central Processor 12
 - 2.3 The Teletype 12
 - 2.4 The Line Clock 13
- Exercises 14

3**REPRESENTATION OF NUMBERS AND CHARACTERS 15**

- 3.1 2's-Complement Representation 15
- 3.2 Addition and Subtraction 19
- 3.3 Character Representation 21
- 3.4 Floating-Point Representation 23
- Exercises 24

4**INSTRUCTIONS AND ADDRESSING MODES 26**

- 4.1 The Execution Cycle 27
- 4.2 Single-Operand and Double-Operand Instructions 27
 - 4.3 Addressing Modes 29
 - 4.4 Immediate Addressing 30
 - 4.5 Absolute Addressing 32
 - 4.6 Relative Addressing 33
- 4.7 Relative Deferred Addressing 34
- 4.8 Branch Instructions 35
- 4.9 No-operand Instructions 38
 - 4.10 Examples 39
- Exercises 42

5**ASSEMBLY LANGUAGE PROGRAMMING 45**

- 5.1 Assembly Language versus Machine Language 45
- 5.2 Assembly Language Directives 46
- 5.3 Assembly Language Program Format 50
- 5.4 Example: Multiple Echo 51
- 5.5 Coding Hints 55
- Exercises 57

6**STACKS AND SUBROUTINES 60**

- 6.1 Stacks 60
- 6.2 Example: Backward Echo 62
- 6.3 Subroutines 64
- 6.4 Subroutine Call and Return 65
- 6.5 Argument Transmission 67
- 6.6 Nested Subroutines 70
- 6.7 Recursive Subroutines 72
- 6.8 Example: Tower of Hanoi 74
- 6.9 Coroutines 79
- Exercises 80

7**ARITHMETIC OPERATIONS 85**

- 7.1 Carry and Overflow under Addition 85
- 7.2 Carry and Overflow under Subtraction 87
- 7.3 Double-Precision Arithmetic 88
- 7.4 The TST and CMP Instructions 91
- 7.5 More on Branch Instructions 93

7.6	Shift Instructions	97
7.7	Example: ASCII-to-Binary Conversion	99
	Exercises	106

8

TRAPS AND INTERRUPTS 111

8.1	Traps	111
8.2	Illegal Address and Illegal Instruction Traps	112
8.3	The Trap Bit and BPT Instruction	113
8.4	Interrupts	114
8.5	Why Use Interrupts?	116
8.6	Priority Interrupts	118
8.7	Example: Time Request	120
	Exercises	126

9

THE ASSEMBLER AND LINKAGE EDITOR 130

9.1	The Two-Pass Assembly Process	131
9.2	Example of Assembler Listing	134
9.3	Absolute and Relocatable Addresses	136
9.4	The Linkage Editor	138
9.5	Address Modification	140
9.6	Global Symbols	140
9.7	The Two-Pass Linkage Process	144
9.8	Position-Independent Code	147
	Exercises	148

10

ADVANCED ASSEMBLY LANGUAGE TECHNIQUES 151

10.1	Macros	151
10.2	Macro Definitions and Macro Calls	154

10.3 Local Symbols	159
10.4 Repeat Directives	161
10.5 Conditional Assembly	164
Exercises	167

APPENDIXES 171

A. PDP-11 Organization (Partial)	172
B. Seven-Bit ASCII Code (Partial)	173
C. PDP-11 Addressing Modes	174
D. PDP-11 Instructions (Partial List)	175
E. MACRO-11 Directives (Partial List)	177
F. Powers of 2	178
G. Notes on Programming Style	179

INDEX 183

PREFACE

This book evolved from notes written for a course in machine structures offered at the University of California at Berkeley. It is a second course in computer science (the first being a course in a high-level language programming) where students are exposed to the basic concept of computer operation (registers, instruction set, addressing modes, etc.) and learn assembly language programming techniques.

The course offers the students “hands-on” experience with a stand-alone computer, one which they can program and manipulate with no operating system standing between them and important machine features. The availability of a “bare” machine not only eliminates much of the “magic” associated with computers and computation, but also permits experimentation with I/O programming, interrupt handling, and other essential techniques not possible with simulated or “protected” machines. In addition to the stand-alone machine, students have access to a large time-shared computer where they can perform editing, assembling, linking, filing, and other operations which are essentially of a bookkeeping nature.

For these purposes we selected the Digital Equipment Corporation’s PDP-11 family of computers. (Specifically, we used the PDP-11/10 as the

stand-alone machine, and the PDP-11/70, equipped with the UNIX operating system, as the time-shared computer.) The PDP-11 machines were chosen because of their versatility, popularity, and the wide range of available software systems. We found the PDP-11's highly suitable as vehicles for presenting the fundamental concepts of the course, and at the same time exposing the students to the ever-widening world of minicomputers.

The objective of this book, then, is to familiarize the reader with the basic organizational and operational features of the PDP-11 and to present machine and assembly language techniques for this class of computers. It is not, per se, a general text on machine structures, and does not attempt to provide a comprehensive treatment of available computer organizations and assemblers. However, to the extent that the concepts and methods governing the operation and programming of the PDP-11 are used in many other minicomputers, the material in this book should serve as good preparation for the operation and programming of other machines.

Chapter 1 outlines algorithms for converting numbers from one system (binary, octal, decimal) to another. The algorithms are not provided with proof and are intended to serve only for reference. Chapter 2 describes the organizational structure of the PDP-11 (the central memory, central processor, and peripheral devices). Chapter 3 explains how numbers (integers and floating point), characters, and strings are represented in the PDP-11. Chapter 4 describes the PDP-11's instruction formats and addressing modes, and Chapter 5 introduces the reader to assembly language programming.

The first five chapters should provide the reader with sufficient background to write simple programs for the PDP-11. The remaining chapters delve deeper into operational details and describe further techniques. Chapter 6 introduces stacks and subroutines (including recursion). Chapter 7 looks closer at the PDP-11's arithmetic (including double-precision) and other operations, such as the test, comparison, branch, and shift operations. Chapter 8 explains the trap and interrupt mechanisms. Chapter 9 describes the workings of the assembler and linkage editor and the notion of relocation. (Although the MACRO-11 assembler and LINKR-11 linkage editor are used for illustration, the concepts discussed are quite general.) Chapter 10 introduces some advanced assembler facilities, such as macros, repeated assembly, and conditional assembly.

The book ends with a number of appendixes, which consist of reference lists and tables (character code, summary of addressing modes, list of operation codes, etc.). There is also an appendix on programming style, which should be carefully read by the beginner.

Each chapter concludes with a set of exercises that serve to illustrate and sometimes complement the material in the text. The reader is encouraged to

solve the problems and run the programs included in these exercises. True assimilation of the material in this book can come about only through practice — by the actual writing and execution of programs.

The only prerequisite to this book is some experience with high-level language programming. No particular language is assumed, but it is taken for granted that the reader is familiar with the notions of an algorithm, a flow-chart, and a stored program.

It is not intended that this book stand alone as a course text. Since it does not describe all the fine details of the PDP-11 instructions and assembler directives, students should be in possession of the PDP-11 processor handbook and the assembler manual appropriate to their particular installation, where these details can be found when needed. Little is said in the book regarding peripheral equipment (only the teletype and line clock are treated in any detail), and students doing I/O programming may wish to refer to the PDP-11 peripherals handbook and local manuals for assistance.

The author is indebted to Mr. R. S. Epstein of the University of California at Berkeley for reviewing the manuscript, for offering useful advice, and for contributing most of Section 5.5 as well as some exercises. Thanks are also due to Professors R. S. Fabry and M. R. Stonebraker (also of U.C., Berkeley) for helpful comments and suggestions.

ARTHUR GILL

NUMBER SYSTEMS

In working with the PDP-11, we shall make extensive use of the binary and octal number systems, as well as the decimal system. It is important that the student acquire, as soon as possible, the facility to convert from one system to another. In this chapter we shall outline, without proof, some algorithms for carrying out these conversions. Students familiar with these algorithms may proceed directly to Chapter 2.

A “number” in this chapter will mean a non-negative integer (0, 1, 2, . . .). The number N will be denoted by N_{10} , N_8 , or N_2 if it is in the decimal, octal, or binary system, respectively. However, the subscript may be dropped if it is understood from the context.

An m -digit number will be written symbolically as $D_{m-1} \cdots D_1 D_0$ [D_i being the $(i + 1)$ st digit from the right].

The flowcharts in Figures 1.1 and 1.2 describe algorithms for converting a decimal number N into its binary equivalent M .

Example (Subtraction-of-powers method)

$$\begin{aligned}
 N &= 217_{10} \\
 217 - 2^7 &= 217 - 128 = 89 & (D_7 = 1) \\
 89 - 2^6 &= 89 - 64 = 25 & (D_6 = 1) \\
 25 - 2^4 &= 25 - 16 = 9 & (D_4 = 1) \\
 9 - 2^3 &= 9 - 8 = 1 & (D_3 = 1) \\
 1 - 2^0 &= 1 - 1 = 0 & (D_0 = 1) \\
 M &= 11011001_2
 \end{aligned}$$

□

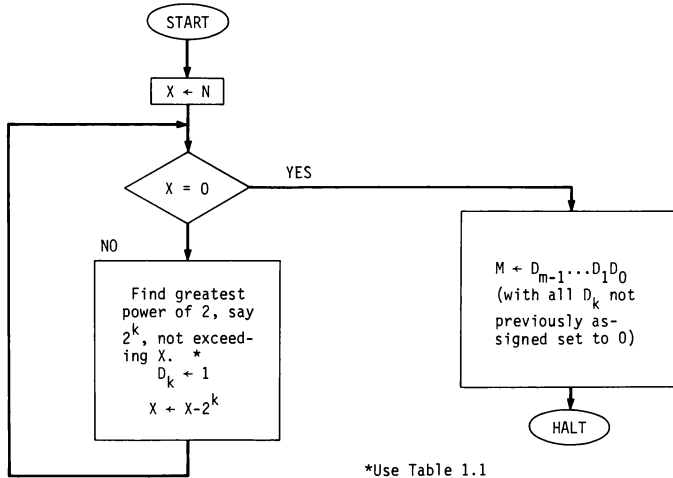


Figure 1.1 Decimal-to-binary conversion by subtraction of powers.

Example (Division method)

$$\begin{aligned}
 N &= 217_{10} \\
 217 &\text{ is odd} & (D_0 = 1) \\
 217/2 &= 108 \text{ is even} & (D_1 = 0) \\
 108/2 &= 54 \text{ is even} & (D_2 = 0) \\
 54/2 &= 27 \text{ is odd} & (D_3 = 1)
 \end{aligned}$$

$$\begin{aligned}
 27/2 &= 13 \text{ is odd} & (D_4 &= 1) \\
 13/2 &= 6 \text{ is even} & (D_5 &= 0) \\
 6/2 &= 3 \text{ is odd} & (D_6 &= 1) \\
 3/2 &= 1 \text{ is odd} & (D_7 &= 1) \\
 1/2 &= 0 & & \\
 \mathbf{M} &= \mathbf{11011001_2} & &
 \end{aligned}$$

□

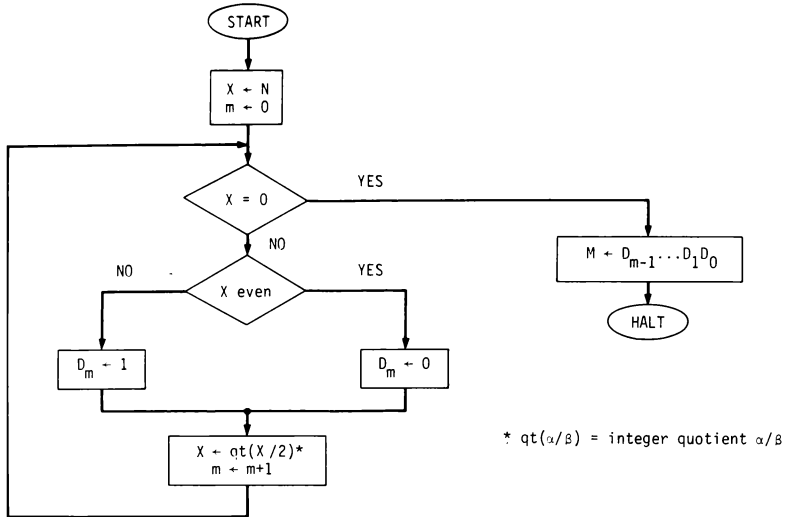


Figure 1.2 Decimal-to-binary conversion by division.

1.2 DECIMAL-TO-OCTAL CONVERSION

The flowcharts in Figures 1.3 and 1.4 describe algorithms for converting a decimal number N into its octal equivalent M.

Example (Subtraction-of-powers method)

$$\begin{aligned}
 N &= 2591_{10} \\
 2591 - 5 \cdot 8^3 &= 2591 - 2560 = 31 & (D_3 &= 5) \\
 31 - 3 \cdot 8^1 &= 31 - 24 = 7 & (D_1 &= 3) \\
 7 - 7 \cdot 8^0 &= 0 & (D_0 &= 7) \\
 \mathbf{M} &= \mathbf{5037_8} & &
 \end{aligned}$$

□

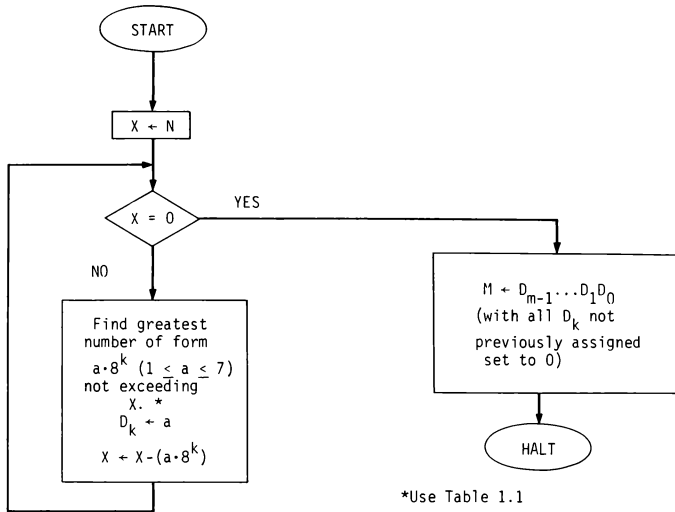


Figure 1.3 Decimal-to-octal conversion by subtraction of powers.

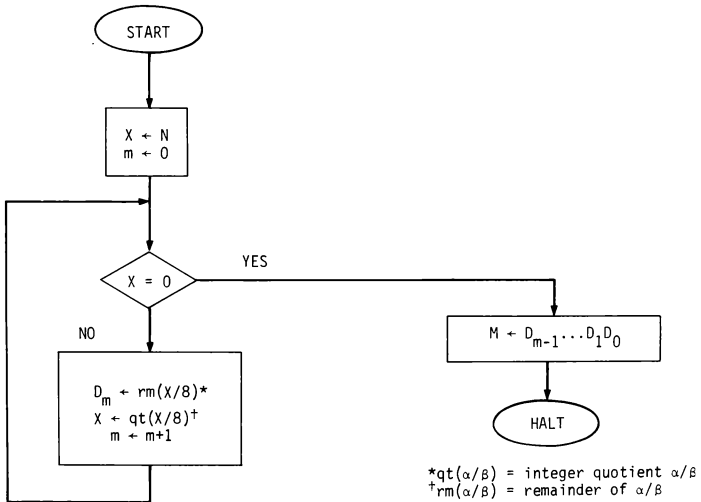


Figure 1.4 Decimal-to-octal conversion by division.

Example (Division method)

$$\begin{aligned}
 N &= 2591_{10} \\
 2591/8 &= 323 \text{ (remainder 7)} & (D_0 = 7) \\
 323/8 &= 40 \text{ (remainder 3)} & (D_1 = 3) \\
 40/8 &= 5 \text{ (remainder 0)} & (D_2 = 0) \\
 5/8 &= 0 \text{ (remainder 5)} & (D_3 = 5) \\
 M &= 5037_8
 \end{aligned}$$

□

(Another method consists of first converting N into binary as shown in Section 1.1, and then converting the result into octal as shown in Section 1.6.)

1.3 BINARY-TO-DECIMAL CONVERSION

If $N = D_{m-1} \cdots D_1 D_0$ is a binary number, then its decimal equivalent is

$$M = \sum_{i=0}^{m-1} D_i \cdot 2^i$$

(Powers of 2 are listed in Table 1.1.)

Example

$$\begin{aligned}
 N &= 1011100_2 \\
 M &= 2^2 + 2^3 + 2^4 + 2^6 = 4 + 8 + 16 + 64 = 92_{10}
 \end{aligned}$$

□

1.4 OCTAL-TO-DECIMAL CONVERSION

If $N = D_{m-1} \cdots D_1 D_0$ is an octal number, then its decimal equivalent is

$$M = \sum_{i=0}^{m-1} D_i \cdot 8^i$$

(Powers of 8 are listed in Table 1.1.)

TABLE 1.1 Powers of 2 and 8

$8^0 = 2^0 = 1$	$8^3 = 2^9 = 512$
$2^1 = 2$	$2^{10} = 1024$
$2^2 = 4$	$2^{11} = 2048$
$8^1 = 2^3 = 8$	$8^4 = 2^{12} = 4096$
$2^4 = 16$	$2^{13} = 8192$
$2^5 = 32$	$2^{14} = 16384$
$8^2 = 2^6 = 64$	$8^5 = 2^{15} = 32768$
$2^7 = 128$	$2^{16} = 65536$
$2^8 = 256$	$2^{17} = 131072$

Example

$$\begin{aligned}
 N &= 3107_8 \\
 M &= 7 \cdot 8^0 + 0 \cdot 8^1 + 1 \cdot 8^2 + 3 \cdot 8^3 \\
 &= 7 \cdot 1 + 0 \cdot 8 + 1 \cdot 64 + 3 \cdot 512 \\
 &= 7 + 64 + 1536 \\
 &= 1607_{10}
 \end{aligned}$$

□

1.5

OCTAL-TO-BINARY CONVERSION

The binary equivalent M of the octal number N is obtained by replacing each digit in N with a group of three binary digits as shown in Table 1.2. (Leading 0's in the leftmost group can be deleted.)

Example

$$\begin{aligned}
 N &= 160734_8 \\
 M &= 1\ 110\ 000\ 111\ 011\ 100_2
 \end{aligned}$$

□

TABLE 1.2 Octal-to-Binary Conversion

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

BINARY-TO-OCTAL CONVERSION

The octal equivalent M of the binary number N is obtained by partitioning N into three-digit groups from right to left and then replacing each group by an octal digit as shown in Table 1.2. (If the length of N is not divisible by 3, add enough leading 0's to make it so.)

Example

$$N = 11010101_2 = 011\ 010\ 101$$

$$M = 325_8$$

□

BINARY AND OCTAL ADDITION

Two binary numbers are added as if they were decimal, except for the following rules:

$$1 + 1 = 0 \quad (\text{carry } 1)$$

$$1 + 1 + 1 = 1 \quad (\text{carry } 1)$$

Example

$$\begin{array}{r} 101101_2 \\ + 100111_2 \\ \hline \end{array}$$

$$1010100_2$$

□

Two octal numbers are added as if they were decimal, except for the following rule: If two digits (added as if they were decimal) yield the sum $D \geq 8$, then replace D with $D - 8$ and carry 1 (e.g., $4 + 5 = 1$ with carry 1; $7 + 7 = 6$ with carry 1).

Example

$$\begin{array}{r} 67036_8 \\ + 52147_8 \\ \hline \end{array}$$

$$141205_8$$

□

- 1.1 Using both the subtraction-of-powers method and the division method, convert the following decimal numbers into their binary and octal equivalents.
- (a) 2337_{10} (c) 16383_{10}
(b) 10000_{10}
- 1.2 Convert the following binary numbers into their decimal and octal equivalents.
- (a) 1111111_2 (c) 1010011100101110_2
(b) 10000000_2
- 1.3 Convert the following octal numbers into their decimal and binary equivalents.
- (a) 377_8 (c) 123456_8
(b) 77777_8
- 1.4 Perform the following binary additions and check the results by converting the numbers into decimal.
- (a) $1011010_2 + 11010_2$ (b) $1111101_2 + 1110_2$
- 1.5 Perform the following octal additions and check the results by converting the numbers into decimal.
- (a) $1456_8 + 567_8$ (b) $1234_8 + 7473_8$
- 1.6 Compute the following binary product and check the result by converting the numbers into decimal.
- $$11010111_2 \times 100101_2$$
- 1.7 Show that the binary equivalent of $2^k - 1$ is $111 \dots 1$ (k times).

THE PDP-11 ORGANIZATION

2

The PDP-11 consists of a *central processor* (CP), where all computations take place; the *central memory* (CM), where the data and program are stored; and *peripheral devices*, such as the teletype, printer, paper tape punch, card reader, plotter, cathode-ray tube display, magnetic disk, magnetic tape, clock, and front-panel switches. In this text we shall assume a very rudimentary PDP-11 configuration with only a *teletype* (TTY) and a *line clock* as peripheral devices. Details on these and other devices can be found in the "PDP-11 Peripheral Handbook."

Figure 2.1 outlines the general structure of the PDP-11. In this chapter we shall describe the main features of the various components shown in this figure.

A symbol by itself, for example A, will represent a CM address or a register name. The notation (A) will stand for "the contents of A." Similarly, ((A)) will stand for the "contents of (A)" (i.e., the contents of the memory address found in A).*

*In the "PDP-11 Processor Handbook" these conventions are confined only to memory addresses. When A is a register name, it stands in the handbook for "the contents of register A"; (A) stands for "the contents of the CM address found in register A."

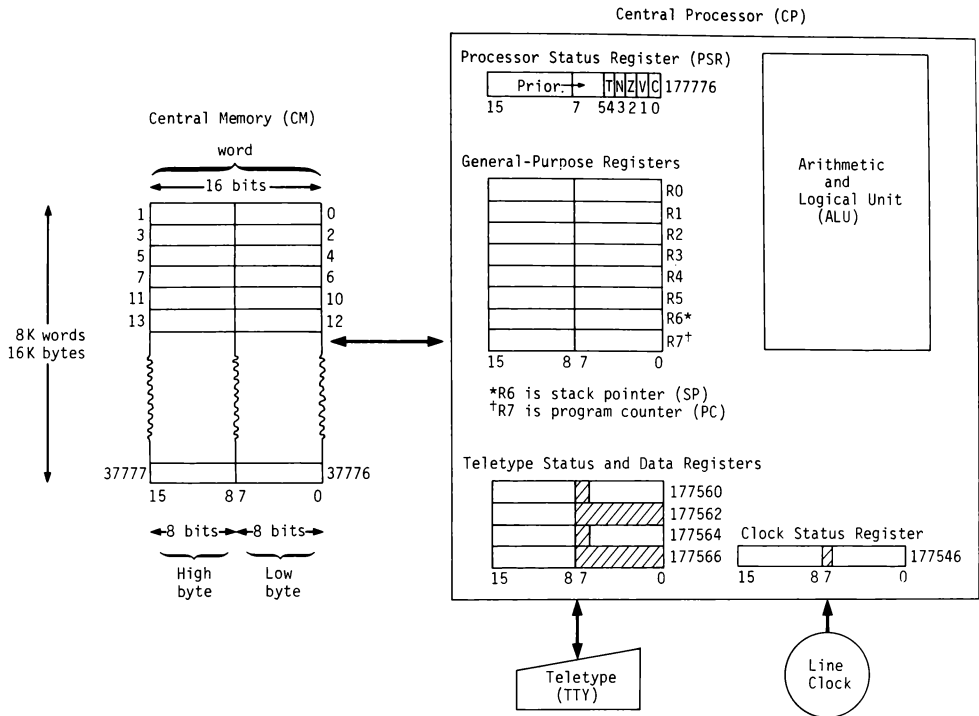


Figure 2.1 PDP-11 organization.

2.1 THE CENTRAL MEMORY

The basic memory element of a computer is the *bit*, which is capable of holding a single binary digit, either a 0 or a 1. The multitude of bits of which the central memory consists are grouped into *bytes*, which, in turn, are grouped into *words*.

In the PDP-11 each word consists of 16 bits, numbered (right to left)

from 0 through 15; bits 0 through 7 constitute the *low byte* and bits 8 through 15 the *high byte* of the word:

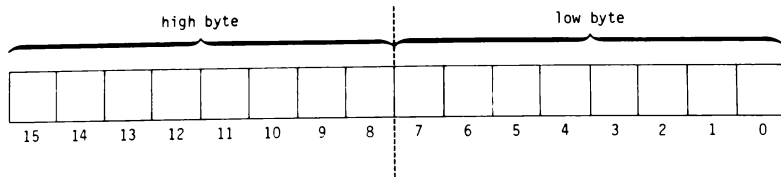


Figure 2.1 shows a PDP-11 memory consisting of $2^{13} = 8192_{10} = 8\text{K}$ words or $2^{14} = 16384_{10} = 16\text{K}$ bytes (where K stands for $2^{10} = 1024$). Larger memories are possible, the exact size (always a multiple of 4096_{10}) varying from one installation to another.

Each byte in the CM is identified by a unique number called the *address* (or the *location*) of the byte. If the CM has b bytes, the sequence of byte addresses is $0, 1, 2, \dots, b - 1$. The address of a word is, by convention, the address of its low byte. Thus, the sequence of word addresses is $0, 2, 4, 6, \dots, b - 2$. Note that a *word address is always even*. For example, in the 8K-word machine shown in Figure 2.1, the byte addresses are

$$0, 1, 2, 3, \dots, 37777_8 \quad (37777_8 = 16383_{10})$$

and the word addresses are

$$0, 2, 4, 6, \dots, 37776_8$$

We shall always write CM addresses *in octal*.

The contents of a word (or byte) can be, naturally, represented by a sequence of 16 (or 8) binary digits. When the sequence, viewed as a binary number, is converted into its six-digit (or three-digit) octal equivalent, the result is the *octal contents* of the word (or byte) — a concise and convenient representation which will be used frequently in this text. For example, the word whose actual (binary) contents is 1010011100101110 has the octal contents 123456; its low byte has the octal contents 056, and its high byte, 247.

The CP consists of the *arithmetic and logical unit* (ALU), where all numerical and logical operations take place, and a number of 16-bit (word-size) *registers* which constitute the CP's "private memory."

There are 8 *general-purpose registers* called R0, R1, . . . , R7. R6 is also called the *stack pointer* (SP), for reasons to be discussed later. R7 is also called the *program counter* (PC); it always contains the address of the next program instruction to be executed.

The *processor status register* (PSR) exhibits a number of 1-bit *condition codes* whose values depend on the result of the instruction last executed. For example:

$$\begin{aligned} \text{Z code (bit 2)} &= \begin{cases} 1 & \text{if result was 0} \\ 0 & \text{if result was } \neq 0 \end{cases} \\ \text{N code (bit 3)} &= \begin{cases} 1 & \text{if result was } < 0 \\ 0 & \text{if result was } \geq 0 \end{cases} \end{aligned}$$

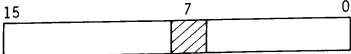
Other condition codes will be discussed in later chapters. It should be noted that different instructions affect the condition codes differently (and some do not affect them at all). The precise effect of each instruction on the condition codes is specified in the "PDP-11 Processor Handbook."

A number of registers associated with peripheral devices will be introduced in subsequent sections.

2.3 THE TELETYPE

By means of the TTY, information can be entered into the PDP-11 (by a human operator) via a *keyboard*, and printed by the PDP-11 via a *printer*. It is important to note that unlike an ordinary typewriter, the TTY consists of *two* devices, the keyboard and the printer, which are completely independent. (For example, pressing a character on the keyboard will not automatically result in a printout of this character.)

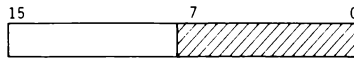
The TTY is associated with four registers:

Keyboard status register (addressed 177560): 

Bit 7 (the "ready" bit) is 1 when the character typed in is available in

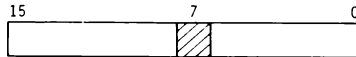
the keyboard data register. It becomes 0 only after an instruction is executed which refers to the keyboard data register.

Keyboard data register (addressed 177562):



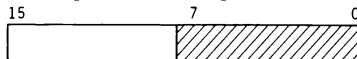
Bits 0-7 hold the encoded form of the character typed in. This register is “read only” and cannot be written into by a program.

Printer status register (addressed 177564):



Bit 7 (the “ready” bit) is 1 when the printer is available for printing. It becomes 0 as soon as a character is loaded into the printer data register.

Printer data register (addressed 177566):



Bits 0-7 hold the encoded form of the character to be printed. If the printer status register’s “ready” bit is 1, printing will take place immediately upon loading the printer data register. This register is “write only.” If read, it behaves as if it were all-zero.

We shall later see that 1 in bit 7 of a byte whose address is α implies that the number stored in byte α is negative. Thus, to test whether a new character has already been delivered from the keyboard to the keyboard data register (assuming that the previous character has already been processed), we simply test whether byte 177560 contains a negative number. Similarly, to test whether the teletype printer is available (for printing the contents of the printer data register), we simply test whether byte 177564 contains a negative number.

2.4 THE LINE CLOCK

For timing purposes, a line clock is linked to the PDP-11, which “ticks” 60 times per second. It is associated with a *clock status register* (addressed 177546), whose bit 7 (the “ready” bit) becomes 1 every 1/60 of a second. This bit does not normally revert to 0 after the “tick”; it should be cleared by the program immediately after it becomes 1 so that the next “tick” can be detected.*

As explained in the preceding section, the appearance of a “tick” in the line clock can be recognized by the fact that the byte 177546 becomes negative.

*However, when bit 6 (the “interrupt enable” bit) of the clock status register is 1, the clearing of the ready bit after each “tick” is accomplished *automatically*.

2.1 Write (in octal) the highest byte and word addresses in a w -word computer, where w (in decimal) is

(a) 28K (b) 2^{15}

2.2 How many words are in a CM whose highest byte address is 17777_8 ?

2.3 Determine the octal contents of the words that contain

(a) 0110110110110110 (b) 1001011111010101

What are the octal contents of the low and high bytes of each of these words?

REPRESENTATION OF NUMBERS AND CHARACTERS



In this chapter we shall see how numbers (integer and “real,” positive and negative), characters, and character strings are stored within the PDP-11.

Unless otherwise specified, a “number” will mean an integer. A “positive number” will mean a non-negative integer (0, 1, 2, . . .).

3.1

2'S-COMPLEMENT REPRESENTATION

Numbers are represented in the PDP-11 in the *n-bit 2's-complement* forms, where *n* is either 16 (for word representation) or 8 (for byte representation). This form is constructed by the following rules.

To store a positive number N:

1. Express *N* in binary form.
2. Store the (binary) *N* right-justified in the word (or byte), supplying

it with as many leading 0's as needed to make the total number of digits n .

Examples

1. $N = 1607_{10} = 3107_8$

16-bit 2's-complement representation:

$$0\ 000\ 011\ 001\ 000\ 111\ (003107_8)$$

2. $N = 10_{10} = 12_8$

8-bit 2's-complement representation:

$$00\ 001\ 010\ (012_8)$$

□

The largest positive number recognized in the n -bit 2's-complement form is

$$\underbrace{11 \dots 1}_n_2 = 2^{n-1} - 1$$

When $n = 16$, then

$$N \leq 0\ 111\ 111\ 111\ 111\ 111_2 = 077777_8 = 32767_{10}$$

When $n = 8$:

$$N \leq 01\ 111\ 111_2 = 177_8 = 127_{10}$$

Thus, the most-significant bit (MSB) in every positive 2's-complement number is 0.

To store a number $-N$:

1. Store N .
2. Obtain $\sim N$ (the 1's complement of N) from N by replacing (bit by bit) every 0 with 1 and every 1 with 0. (The octal form of $\sim N$ can be obtained from the octal form of N by replacing every digit D of N with $7 - D$.*)
3. Add 1 to $\sim N$ (ignoring carry from MSB, if any).

*If the largest possible value of the leftmost digit is 1 or 3 (rather than 7), replace it with $1 - D$ or $3 - D$, respectively (instead of $7 - D$).

Examples

1. $N = 1607_{10} = 3107_8$

16-bit 2's-complement representation:

$$\begin{array}{r}
 N: \quad 0\ 000\ 011\ 001\ 000\ 111 \\
 \sim N: \quad 1\ 111\ 100\ 110\ 111\ 000 \\
 +1: \quad \underline{\hspace{10em}} \quad 1 \\
 -N: \quad 1\ 111\ 100\ 110\ 111\ 001
 \end{array}$$

In octal:

$$\begin{array}{r}
 N: \quad 0\ 0\ 3\ 1\ 0\ 7 \\
 \sim N: \quad 1\ 7\ 4\ 6\ 7\ 0 \\
 +1: \quad \underline{\hspace{2em}} \quad 1 \\
 -N: \quad 1\ 7\ 4\ 6\ 7\ 1
 \end{array}$$

2. $N = -1607_{10}$

16-bit 2's-complement representation:

$$\begin{array}{r}
 N: \quad 1\ 111\ 100\ 110\ 111\ 001 \\
 \sim N: \quad 0\ 000\ 011\ 001\ 000\ 110 \\
 +1: \quad \underline{\hspace{10em}} \quad 1 \\
 -N: \quad 0\ 000\ 011\ 001\ 000\ 111
 \end{array}$$

In octal:

$$\begin{array}{r}
 N: \quad 1\ 7\ 4\ 6\ 7\ 1 \\
 \sim N: \quad 0\ 0\ 3\ 1\ 0\ 6 \\
 +1: \quad \underline{\hspace{2em}} \quad 1 \\
 -N: \quad 0\ 0\ 3\ 1\ 0\ 7
 \end{array}$$

3. $N = 10_{10} = 12_8$

8-bit 2's-complement representation:

$$\begin{array}{r}
 N: \quad 00\ 001\ 010 \\
 \sim N: \quad 11\ 110\ 101 \\
 +1: \quad \underline{\hspace{2em}} \quad 1 \\
 -N: \quad 11\ 110\ 110
 \end{array}$$

In octal:

$$\begin{array}{r}
 N: \quad 012 \\
 \sim N: \quad 365 \\
 +1: \quad \underline{\quad 1} \\
 -N: \quad 366
 \end{array}$$

□

Note that the MSB of every negative 2's-complement number is 1. Thus, the MSB indicates whether the number is negative or positive. (The MSB is therefore often referred to as the *sign bit*.)

If $N = 0$, the 2's-complement form of $-N$ is obtained as follows:

$$\begin{array}{r}
 N: \quad 00 \dots 00 \\
 \sim N: \quad 11 \dots 11 \\
 +1: \quad \underline{\quad \quad 1} \\
 -N: \quad 00 \dots 00
 \end{array}$$

Hence, the forms of $+0$ and -0 in 2's complement are identical.

The n -bit number $10 \dots 0$ is "its own negative," and not the negative of any positive n -bit 2's-complement number:

$$\begin{array}{r}
 N: \quad 100 \dots 00 \\
 \sim N: \quad 011 \dots 11 \\
 +1: \quad \underline{\quad \quad 1} \\
 -N: \quad 100 \dots 00
 \end{array}$$

This number is employed to represent -2^{n-1} . Thus, in n -bit 2's-complement representation,

$$-2^{n-1} \leq N \leq 2^{n-1} - 1$$

Table 3.1 shows the range of numbers representable in 16-bit and in 8-bit 2's-complement forms. Note that an 8-bit 2's-complement number (stored in low byte) can be converted into a 16-bit 2's-complement number simply by extending its sign bit (bit 7) to the entire high byte.

TABLE 3.1. Range of Numbers Representable in n-Bit 2's-Complement Form

	Decimal Value	Octal Representation
n = 16	32767	077777
	32766	077776
	.	.
	.	.
	.	.
	2	000002
	1	000001
	0	000000
	-1	177777
	-2	177776
	.	.
	.	.
	.	.
	-32766	100002
	-32767	100001
-32768	100000	
n = 8	127	177
	126	176
	.	.
	.	.
	.	.
	2	002
	1	001
	0	000
	-1	377
	-2	376
	.	.
	.	.
.	.	
-126	202	
-127	201	
-128	200	

3.2

ADDITION AND SUBTRACTION

The sum of two numbers, N_1 and N_2 , in n-bit 2's-complement form is computed by adding N_1 and N_2 as if they were unsigned (n-bit positive numbers), and ignoring the carry from the MSB (if any). The difference $N_1 - N_2$ is computed simply by adding N_1 to the 2's complement of N_2 .

Examples (n = 16)

Decimal	Binary	Octal
1. 11	0 000 000 000 001 011	000013
+ 21	+ 0 000 000 000 010 101	+ 000025
32	0 000 000 000 100 000	000040
2. 21	0 000 000 000 010 101	000025
- 11	+ 1 111 111 111 110 101	+ 177765
10	✓ 0 000 000 000 001 010	000012
3. 11	0 000 000 000 001 011	000013
- 21	+ 1 111 111 111 101 011	+ 177753
- 10	1 111 111 111 110 110	177766
4. - 11	1 111 111 111 110 101	177765
- 21	+ 1 111 111 111 101 011	+ 177753
- 32	✓ 1 111 111 111 100 000	177740

□

As mentioned in Chapter 2, arithmetic operations in the PDP-11 influence the Z and N condition codes in the PSR. They also influence the C (“carry”) and V (“overflow”) codes in the PSR. For example, after an addition operation:

$$C \text{ code (bit 0)} = \begin{cases} 1 & \text{if addition resulted in carry from MSB} \\ 0 & \text{otherwise} \end{cases}$$

$$V \text{ code (bit 1)} = \begin{cases} 1 & \text{if added numbers are of same sign and their} \\ & \text{sum is of the opposite sign (“overflow”)} \\ 0 & \text{otherwise} \end{cases}$$

More will be said about the C and V codes in a later chapter.

Characters consist of:

Letters: A, B, . . . , Z, a, b, . . . , z

Digits: 0, 1, . . . , 9

Special characters: +, *, /, ↑, \$, space, . . .

Nonprinting characters: bell, line feed (LF), carriage return (CR), . . .

A character is represented in the PDP-11 by an 8-bit binary number (occupying 1 byte). The rightmost 7 bits of this number correspond to the ASCII code of the character (see Table 3.2). Note that the ASCII code of the digit i is $60+i$ (octal).

The leftmost bit of a byte that contains a character is called the *parity bit* and is sometimes used for error detection. When an “even-parity” scheme is used, the parity bit is filled in such a manner so as to make the total number of 1-bits in the byte even. For example, the letter G (ASCII code $107_8 = 01000111_2$) is stored as $107_8 = 01000111_2$, since the number of 1’s in the ASCII code of G is even. However, the letter g (ASCII code $147_8 = 01100111_2$) is stored as $347_8 = 11100111_2$, since the number of 1’s in the ASCII code of g is odd.

Now, suppose that a character (coded according to the even-parity scheme) is transmitted from some peripheral device to the CM or CP. If something happened during transmission to alter one of the bits (from 0 to 1 or from 1 to 0), then the number of 1’s in the received byte will no longer be even. Thus, the erroneous transmission of some bit in a byte can be detected simply by checking whether the received byte has an even or an odd number of 1-bits.*

The “odd-parity” scheme for error detection follows the same principle, except that the parity bit is filled so as to make the total number of 1’s odd rather than even.

A character entered into the PDP-11 via the teletype may in some cases contain a nonzero parity bit, and hence may not correspond to its ASCII code (as given in Table 3.2). Thus, before comparing an entered character with an internally stored one (which conforms with Table 3.2), it is a good idea to zero in the entered character all bits except the rightmost seven.

*Note that any odd number of errors, but no even number of errors, can be detected by this scheme. Usually, the probability of having more than one erroneous bit per byte is too low to worry about.

TABLE 3.2 Some Characters and Their ASCII Codes (in octal)

Character	Code	Character	Code
Bell	007	N	116
Line feed	012	O	117
Carriage return	015	P	120
Space	040	Q	121
!	041	R	122
"	042	S	123
#	043	T	124
\$	044	U	125
%	045	V	126
&	046	W	127
'	047	X	130
(050	Y	131
)	051	Z	132
*	052	[133
+	053	\	134
,	054]	135
-	055	↑	136
.	056	←	137
/	057	`	140
0	060	a	141
1	061	b	142
2	062	c	143
3	063	d	144
4	064	e	145
5	065	f	146
6	066	g	147
7	067	h	150
8	070	i	151
9	071	j	152
:	072	k	153
;	073	l	154
<	074	m	155
=	075	n	156
>	076	o	157
?	077	p	160
@	100	q	161
A	101	r	162
B	102	s	163
C	103	t	164
D	104	u	165
E	105	v	166
F	106	w	167
G	107	x	170
H	110	y	171
I	111	z	172
J	112	{	173
K	113		174
L	114	}	175
M	115	~	176

The parity bit has no effect on the printer: a character will be properly printed out regardless of whether its parity bit is zero or not.

Character strings (i.e., sequences of characters) are stored in successive bytes in the CM. For example, the string PDP-11 is stored in three successive words as follows:

042120	(D=104, P=120)
026520	(- =055, P=120)
030461	(1=061, 1=061)

Consecutive character strings can be conveniently separated in the CM by the *null* character (ASCII code 000), a character that is rarely included in any “practical” string.

3.4 FLOATING-POINT REPRESENTATION

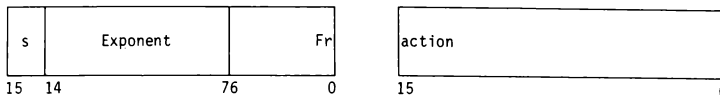
The basic PDP-11 is not equipped to handle “real” numbers (i.e., numbers with a fractional part). However, optional hardware is available for some models to carry out arithmetic operations on real numbers; alternatively, such operations can be implemented by software (i.e., they can be programmed).

The reader is probably familiar with the *scientific* or *floating-point representation* of real decimal numbers, where, for example, 12345000000_{10} is written as $.12345 \times 10^{11}$ and 0.00012345 is written as $.12345 \times 10^{-3}$. This notation, which represents a number by its *fraction* or *mantissa* (e.g., 12345) and by its *exponent* (e.g., 11 or -3), together with an assumed decimal point (e.g., immediately to the left of the fraction), is especially convenient for expressing very large and very small numbers.

In the binary equivalent of the floating-point representation, the notation is of the form $.f \times 2^e$, where both the fraction f and exponent e are binary. (The point preceding f is here referred to as the *binary point*.) For example, $-832_{10} = -1101000000_2$ is written as $-.1101 \times 2^{1010}$, and $0.0390625_{10} = 0.0000101_2$ is written as $.101 \times 2^{-100}$.

The representation of real numbers in the PDP-11 (as in most other computers) is based on the binary floating-point notation. A “single-precision” real number occupies two memory words (32 bits), using the following format*:

*A “double-precision” floating-point number occupies four memory words, with the fraction field consisting of 55 (instead of 23) bits.



Here s is a sign bit—0 if the number is positive and 1 if negative. The binary exponent is stored in excess-200₈ form; that is, it equals the true exponent plus 200₈ = 10000000₂. (The true exponents can thus range from -128₁₀ to +127₁₀.) The binary point is assumed to be immediately to the left of the fraction. The fraction is normalized; that is, it is shifted to the left (and the exponent incremented correspondingly) until its leftmost bit is 1. This leading 1 (which is always there after normalization, and hence redundant) is then deleted and the fraction is shifted one additional bit to the left. An exception is the real number 0, which is represented by an arbitrary fraction and an exponent field (bits 7 through 14) filled with 0's.

Examples

1. $N = -832_{10} = -.1101 \times 2^{+1010}$

The normalized fraction is 1101 (in which the leading 1 is deleted) and the excess-200₈ exponent is 10000000 + 1010 = 10001010. Thus, the floating-point representation of N is (in binary)

1 10001010 1010000 0000000000000000

2. $N = 0.0390625 = .101 \times 2^{-100}$

The normalized fraction is 101 (in which the leading 1 is deleted) and the excess-200₈ exponent is 10000000 - 100 = 1111100. Thus, the floating-point representation of N is (in binary)

0 01111100 0100000 0000000000000000 □

In this text we shall not refer any more to real numbers but will deal exclusively with integers.

EXERCISES

3.1 Show the octal contents of the words that contain:

- | | |
|-------------------------|------------------------|
| (a) -1 | (c) -1000 ₈ |
| (b) -1000 ₁₀ | (d) -7530 ₈ |

3.2 Find the octal number N if:

- (a) The 16-bit 2's-complement form of $-N$ is 176542.
 (b) The 8-bit 2's-complement form of $-N$ is 273.

3.3 Without performing any octal-to-binary or octal-to-decimal conversions, write the octal contents of the words that contain:

- (a) $500_8 - 311_8$ (c) $-1043_8 - 751_8$
 (b) $3721_8 - 6260_8$

3.4 Bytes 1000 through 1007 are filled with the following numbers and characters:

1000:	123_{10}	1001:	17_8
1002:	-32_8	1003:	ASCII "6"
1004:	ASCII "Z"	1005:	-100_{10}
1006:	ASCII "LF"	1007:	ASCII "CR"

Show the octal contents of words 1000, 1002, 1004, and 1006.

3.5 Determine the character string stored in the five consecutive words whose octal contents are

020061
 026117
 046103
 041517
 003513

3.6 Show the floating-point representations of

- (a) $-(1/16)_{10} = -0.0001_2$ (b) $(10.25)_{10} = 1010.01_2$

3.7 What decimal numbers have the following floating-point representations (given in octal):

- (a) 041377 000000 (b) 137000 000000

INSTRUCTIONS AND ADDRESSING MODES



In this chapter we shall describe the format of PDP-11 instructions and the various methods used for specifying operands' addresses ("addressing modes").

Instructions in the PDP-11 occupy one word (followed, possibly, by one or two additional words to specify operands), and will be written as six-digit octal numbers. These numbers correspond to the way instructions would appear in a *machine language* program, that is, a program actually residing in the CM and executable by the CP. We shall also write instructions in *mnemonic* or *symbolic* form. This is the form in which instructions would appear in an *assembly language* program, a "symbolic" program that requires translation (by an *assembler*) into a machine language program before execution.

Henceforth all numbers will be assumed to be octal unless otherwise specified.

As mentioned in Chapter 2, the PC at any time contains the address of the instruction to be executed next. After the instruction is fetched from that address, but *before execution*, the contents of PC is incremented by 2. Thus, if PC does not change in the course of execution (which may happen with “branch” instructions and others), the next instruction will be automatically fetched from the next word. This “execution cycle” continues until a HALT instruction is encountered, which causes the PDP-11 to halt (see Figure 4.1).

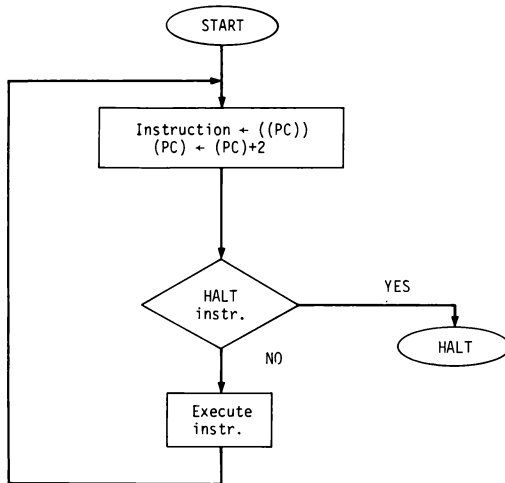


Figure 4.1 The PDP-11 execution cycle.

4.2
SINGLE-OPERAND AND
DOUBLE-OPERAND INSTRUCTIONS

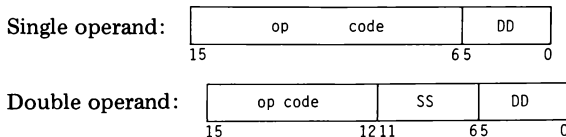
Single-operand instructions consist of an *operation code* (op code), and a *DD field*, which specifies the location of the *destination operand*. Double-operand instructions contain, in addition to the op code and DD

TABLE 4.1 Single-Operand and Double-Operand Instructions

Machine Language	Assembly Language	Name of Instruction	Resulting Action
<u>Single Operand</u>			
0050DD	CLR d	clear	$(d) \leftarrow 0$
0051DD	COM d	complement	$(d) \leftarrow \sim(d)$
0052DD	INC d	increment	$(d) \leftarrow (d) + 1$
0053DD	DEC d	decrement	$(d) \leftarrow (d) - 1$
0054DD	NEG d	negate	$(d) \leftarrow -(d)$
0057DD	TST d	test	$(d) \leftarrow (d)$
0060DD	ROR d	rotate right	$(d) \leftarrow (d)$ shifted right 1 bit
0061DD	ROL d	rotate left	$(d) \leftarrow (d)$ shifted left 1 bit
0062DD	ASR d	arith. shift right	$(d) \leftarrow (d)/2$
0063DD	ASL d	arith. shift left	$(d) \leftarrow 2 * (d)$
0003DD	SWAB d	swap bytes	$(d)_{\text{low}} \leftrightarrow (d)_{\text{high}}$
0055DD	ADC d	add carry	$(d) \leftarrow (d) + C$
0056DD	SBC d	subtract carry	$(d) \leftarrow (d) - C$
0001DD	JMP d	jump	$(PC) \leftarrow d$
<u>Double Operand</u>			
01SSDD	MOV s,d	move	$(d) \leftarrow (s)$
02SSDD	CMP s,d	compare	form $(s) - (d)$
06SSDD	ADD s,d	add	$(d) \leftarrow (s) + (d)$
16SSDD	SUB s,d	subtract	$(d) \leftarrow (d) - (s)$
03SSDD	BIT s,d	bit test	form $(s) \wedge (d)^*$
04SSDD	BIC s,d	bit clear	$(d) \leftarrow [\sim(s)] \wedge (d)^*$
05SSDD	BIS s,d	bit set	$(d) \leftarrow (s) \vee (d)^*$

* \vee is "OR," \wedge is "AND," \sim is "NOT."

field, an *SS field*, which specifies the location of the *source operand*. The formats of these instructions are:



If the instruction operates on a byte (a “byte instruction”) rather than a word, bit 15 is set to 1; otherwise, it is 0. (*Note:* The address of a “word instruction” must be even; otherwise, the program will fail.)

In assembly language, single-operand and double-operand instructions have the formats

OPR	d
OPR	s,d

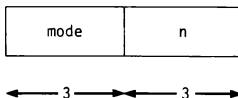
where OPR is the *mnemonic op code* and d and s are the symbolic forms of DD and SS, respectively. In byte instructions, OPR is appended with the letter B (i.e., CLR becomes CLR_B).

Table 4.1 lists the single- and double-operand instructions of the PDP-11. All instructions can be made into byte instructions, except ADD and SUB (which can operate only on 16-bit 2's-complement numbers), SWAB and JMP.

Most instructions listed in Table 4.1 are self-explanatory. A detailed description of each one of them can be found in the "PDP-11 Processor Handbook." Some of the instructions will be elaborated upon in later chapters.

4.3 ADDRESSING MODES

The DD and SS fields are each divided into a *register subfield* (a 3-bit number n), and a *mode subfield* (a 3-bit number *mode*):



The manner in which the operand address is derived from these subfields during execution is described in Table 4.2. Table 4.3 shows examples illustrating the various addressing modes.

TABLE 4.2 Addressing Modes

Mode	Name of Mode	Assembly Language	Operand's Location	Explanation
0	Register	Rn	Rn	Operand is in Rn.
1	Register deferred	(Rn)	(Rn)	Address of operand is in Rn.
2	Autoincrement	(Rn)+	(Rn)	Address of operand is in Rn; (Rn) \leftarrow (Rn)+2 after operand is fetched.*
3	Autoincrement deferred	@(Rn)+	((Rn))	Address of address of operand is in Rn; (Rn) \leftarrow (Rn)+2 after operand is fetched.
4	Autodecrement	-(Rn)	(Rn)	(Rn) \leftarrow (Rn)-2 before address is computed †; address of operand is in Rn.
5	Autodecrement deferred	@-(Rn)	((Rn))	(Rn) \leftarrow (Rn)-2 before address is computed; address of address of operand is in Rn.
6	Index	X(Rn)	X+(Rn)	Address of operand is X plus (Rn). Address of X is in PC; (PC) \leftarrow (PC)+2 after X is fetched.
7	Index deferred	@X(Rn)	(X+(Rn))	Address of address of operand is X plus (Rn). Address of X is in PC; (PC) \leftarrow (PC)+2 after X is fetched.

*But (Rn) \leftarrow (Rn)+1 if instruction is byte instruction and $n < 6$.
†But (Rn) \leftarrow (Rn)-1 if instruction is byte instruction and $n < 6$.

4.4

IMMEDIATE ADDRESSING

Suppose that we have

Address	Contents	
α	OPR (PC)+,d	(instruction with SS = 27)
$\alpha+2$	k	(constant)
$\alpha+4$...	

Just before the instruction is executed, the contents of PC are incremented by 2 (see Figure 4.1). Thus, the address of the operand of this instruction is

TABLE 4.3 Examples of Addressing Modes

Before each instruction, assume:

(R3) = 000124	(456) = 000174
(122) = 000701	(524) = 000304
(124) = 000456	(700) = 000613
(304) = 000537	

Machine Language*	Instruction Assembly Language	Resulting Action
005003	CLR R3	(R3)←0
005013	CLR (R3)	(124)←0
005023	CLR (R3)+	(124)←0, (R3)←000126
105023	CLRB (R3)+	(124)←000400, (R3)←000125
005033	CLR @(R3)+	(456)←0, (R3)←000126
005043	CLR -(R3)	(R3)←000122, (122)←0
105043	CLRB -(R3)	(R3)←000123, (122)←000301
105053	CLRB @-(R3)	(R3)←000122, (700)←000213
005063 } 000400 }	CLR 400(R3)	(524)←0
005073 } 000400 }	CLR @400(R3)	(304)←0
010300	MOV R3,R0	(R0)←000124
011300	MOV (R3),R0	(R0)←000456
012300	MOV (R3)+,R0	(R0)←000456, (R3)←000126
112300	MOVB (R3)+,R0	(R0)←000056, (R3)←000125†
013300	MOV @(R3)+,R0	(R0)←000174, (R3)←000126
014300	MOV -(R3),R0	(R3)←000122, (R0)←000701
114300	MOVB -(R3),R0	(R3)←000123, (R0)←000001
115300	MOVB @-(R3),R0	(R3)←000122, (R0)←000001
016300 } 000400 }	MOV 400(R3),R0	(R0)←000304
117303 } 000400 }	MOVB @400(R3),R3	(R3)←000137
016363 } 000400 } 000500 }	MOV 400(R3),500(R3)	(624)←000304
012363 } 000400 }	MOV (R3)+,400(R3)	(526)←000456, (R3)←000126

*SS and DD are underlined.

†MOVB s,Rn copies the byte s into the low byte of Rn; the high byte of Rn is then filled with the contents of bit 7 ("sign extension").

$\alpha+2$ and hence the source operand is k. After the operand is fetched, $(PC) \leftarrow (PC)+2 = \alpha+4$ (by virtue of the autoincrement mode). In this manner the operand can be placed immediately after the instruction without

being misinterpreted as the next instruction (hence the term *immediate addressing*).

The sequence

$$\text{OPR (PC)+,d}$$

$$k$$

can be abbreviated in assembly language into

$$\text{OPR \#k,d}$$

(Using immediate addressing for destination makes no sense.)

Examples

Machine Language	Assembly Language	Resulting Action
1. $\begin{array}{l} 012700 \\ 177776 \end{array} \}$	MOV #177776,R0	(R0) \leftarrow 177776
2. $\begin{array}{l} 062716 \\ 000005 \end{array} \}$	ADD #5,(SP)	((SP)) \leftarrow 5+((SP))

□

4.5 ABSOLUTE ADDRESSING

Suppose that we have

Address	Contents
α	OPR @(PC)+,d (instruction with SS = 37)
$\alpha+2$	A (address)
$\alpha+4$...

Just before the instruction is executed, the contents of PC is incremented by 2 (see Figure 4.1). Thus, the address of the address of the operand is $\alpha+2$, and hence the operand is in address A. After the operand is fetched, $(PC)\leftarrow (PC)+2 = \alpha+4$ (by virtue of the autoincrement mode). In this manner the address of the operand can be placed immediately after the instruction without being misinterpreted as the next instruction. (This method is referred to as *absolute addressing*.)

The sequence

OPR @(PC)+,d
A

can be abbreviated in assembly language into

OPR @#A,d

Similar comments apply when DD = 37.

Examples

Machine Language	Assembly Language	Resulting Action
1. 005037 } 000472 }	CLR @#472	(472)←0
2. 012737 } 000007 } 177566 }	MOV #7,@#177566	(177566)←000007 (ring bell) □

4.6
RELATIVE ADDRESSING

Suppose that we have

Address	Contents
α	OPR X(PC),d (instruction with SS = 67)
$\alpha+2$	X (relative address)
$\alpha+4$...

Just before the instruction is executed, the contents of PC is incremented by 2 (see Figure 4.1) and hence equals $\alpha+2$. Just after X is fetched from $\alpha+2$, PC is again incremented and equals $\alpha+4$. Thus, the address A of the source operand, namely $X+(PC)$, is given by

$$A = X+\alpha+4$$

It is seen that X is *the operand's address relative to the current value of PC* (hence the term *relative addressing*).

The instruction

OPR X(PC),d

can be abbreviated in assembly language into

OPR A,d ($A = X + \alpha + 4$)

Similar comments apply when $DD = 67$.

Examples

Machine Language	Assembly Language	Resulting Action
1. 500: 005067 } 502: 000200 }	CLR 704	$(704) \leftarrow 0$ $[704 - 504 = 200]$
<i>Note:</i> The instruction above is equivalent to		
500: 005037 } 502: 000704 }	CLR @#704	$(704) \leftarrow 0$
2. 500: 162767 502: 000015 504: 177600	SUB #15,306	$(306) \leftarrow (306) - 15$ $[306 - 506 = -200 = 177600]$

Note: In this case X is stored in $\alpha + 4$ (rather than $\alpha + 2$) and hence $A = x + \alpha + 6$. □

We see that the relative address X is the “distance” between the actual address and the current value of PC. Thus, if the entire program is shifted in the CM, the relative address need not be modified (since this “distance” remains the same). Relative addresses are the ones to use in “position-independent programming”—the writing of programs in such a way that they can be stored anywhere in the CM. More will be said about these in Chapter 9.

4.7

RELATIVE DEFERRED ADDRESSING

Suppose that we have

Address	Contents
α	OPR @X(PC),d (instruction with $SS = 77$)
$\alpha + 2$	X (relative address)
$\alpha + 4$	---

The address A of the address of the operand is $X+(PC)$; that is, the operand's address is in

$$A = X + \alpha + 4$$

(see Section 4.6). Thus, X is the operand's address's address relative to the current value of PC .

The instruction

$$\text{OPR @X(PC),d}$$

can be abbreviated in assembly language into

$$\text{OPR @A,d}$$

Similar comments apply when $DD = 77$.

Example

Machine Language	Assembly Language	Resulting Action
500: 005077 } 502: 000200 }	CLR @704	(123)←0
⋮		
704: 000123		□

4.8 BRANCH INSTRUCTIONS

Branch instructions are used for program branching when certain conditions are satisfied. They have the form

$$\text{base code} + \text{offset}$$

where the base code is a 16-bit number that is always 0 in the low byte, and the offset (denoted f) is an 8-bit 2's-complement number. Thus (see Table 3.1),

$$-128_{10} \leq f \leq 127_{10} \quad \text{or} \quad 200_8 \leq f \leq 177_8$$

During execution, if the branch condition is satisfied, then

$$(PC) \leftarrow (PC) + 2f$$

Hence, if the branch instruction is in address α , the next instruction will be taken from the address

$$\beta = \alpha + 2 + 2f$$

Conversely, if we wish to branch from address α to address β , we must have

$$f = \frac{1}{2}(\beta - \alpha - 2)$$

[Note that f is the distance, in words, between (PC) and the branch destination, and hence that branch instructions are position-independent.]

Table 4.4 lists the branch instructions for the PDP-11. The first five instructions are the most commonly used. Applications of the remaining instructions will be exposed in a later chapter.

TABLE 4.4 Branch Instructions

Base Code	Assembly Language	Branch to a if*
000400	BR a	(unconditionally)
001000	BNE a	not equal to 0 ($Z = 0$)
001400	BEQ a	equal to 0 ($Z = 1$)
100000	BPL a	plus ($N = 0$)
100400	BMI a	minus ($N = 1$)
102000	BVC a	overflow clear ($V = 0$)
102400	BVS a	overflow set ($V = 1$)
103000	BCC a	carry clear ($C = 0$)
103400	BCS a	carry set ($C = 1$)
002000	BGE a	greater than or equal to 0 ($N \vee V = 0$)
002400	BLT a	less than 0 ($N \vee V = 1$)
003000	BGT a	greater than 0 [$Z \vee (N \vee V) = 0$]
003400	BLE a	less than or equal to 0 [$Z \vee (N \vee V) = 1$]
101000	BHI a	higher ($C \vee Z = 0$)
101400	BLOS a	lower or same ($C \vee Z = 1$)
103000	BHIS a	higher or same ($C = 0$)
103400	BLO a	lower ($C = 1$)

*The rules for the \vee operation are: $0+0=0$, $0+1=1$, $1+1=0$; the rules for the \vee operation are: $0+0=0$, $0+1=1$, $1+1=1$.

If branching is required beyond the 127_{10} range, the JMP instruction can be used.

Examples

1. Find the machine language form of the instruction

554: BEQ 570

Answer:

$$f = \frac{1}{2}(570 - 554 - 2) = \frac{1}{2} \cdot 12 = 5$$

$$001400 + 005 = 001405 \leftarrow \text{instruction}$$

2. Find the machine language form of the instruction

624: BR 602

Answer:

$$f = \frac{1}{2}(602 - 624 - 2) = \frac{1}{2} \cdot (-24) = -12 = 366$$

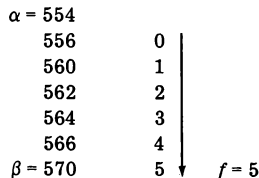
$$000400 + 366 = 000766 \leftarrow \text{instruction}$$

□

For “short” jumps from α to β , one can employ the following simplified procedure for computing the offset: If $\beta > \alpha$, attach to the words $\alpha+2$, $\alpha+4$, $\alpha+6$, . . . the numbers 0, 1, 2, . . . ; the number attached to β is then f . If $\beta \leq \alpha$, attach to the words α , $\alpha-2$, $\alpha-4$, . . . the numbers 377, 376, 375, . . . ; the number attached to β is then f .

Examples

1. To branch from 554 to 570:



2. To branch from 624 to 602:

$\beta = 602$	366	↑	$f = 366$
604	367		
606	370		
610	371		
612	372		
614	373		
616	374		
620	375		
622	376		
$\alpha = 624$	377		

□

4.9

NO-OPERAND INSTRUCTIONS

Table 4.5 lists some PDP-11 no-operand instructions. The HALT instruction is one that is used in almost every program. The others are condition code operations used to either clear (i.e., set to 0) or set (i.e., set to 1) condition code bits in the PSR.

TABLE 4.5 No-Operand Instructions

Machine Language	Assembly Language	Resulting Action
000000	HALT	halt
000241	CLC	clear C
000242	CLV	clear V
000244	CLZ	clear Z
000250	CLN	clear N
000257	CCC	clear C, V, Z, N
000261	SEC	set C
000262	SEV	set V
000264	SEZ	set Z
000270	SEN	set N
000277	SCC	set C, V, Z, N

Two or more clear/set instructions can be written in a single line, separated by !. For example, to clear the C and V bits, issue

```
<CLC!CLV>
```

The following are some program segments that illustrate most of the PDP-11 addressing modes as well as some of the PDP-11 instructions.

It will be noticed that some assembly language instructions make use of “symbolic” addresses (e.g., NEXT) rather than “absolute” (i.e., numerical) addresses. The place in the program corresponding to such an address is indicated by the corresponding symbol appended with a colon (e.g., NEXT:).

All program segments are assumed to start at 600.

1. Store in R1 the absolute value of the 16-bit 2's-complement number X stored in R0.

Machine Language	Assembly Language	Comments
600: 010001	MOV R0,R1	;MOVE X TO R1
602: 100001	BPL NEXT	;IF X \geq 0, DO NOTHING
604: 005401	NEG R1	;ELSE, NEGATE R1
606: ---	NEXT: ---	

Note: A MOV instruction automatically clears or sets the Z and N condition codes of the PSR in accordance with the value of the number moved. Thus, a branch to NEXT will be effected by the BPL instruction only if X is positive.

2. Accept a character from the teletype, move it to R5 and “echo” it (i.e., print it back).

Machine Language	Assembly Language	Comments
600: 105767	WAIT1: TSTB 177560	;IS CHARACTER IN?
602: 176754		
604: 100375	BPL WAIT1	;IF NOT, WAIT
606: 016705	MOV 177562,R5	;ELSE, PUT IT IN R5
610: 176750		
612: 105767	WAIT2: TSTB 177564	;IS PRINTER FREE?
614: 176746		
616: 100375	BPL WAIT2	;IF NOT, WAIT
620: 010567	MOV R5,177566	;ELSE PRINT (R5)
622: 176742		

Note: As mentioned in Section 2.3, one can test whether or not the “ready” bit (bit 7) of a status register is set by testing whether the low byte of the register (regarded as an 8-bit 2's-complement integer) is negative. This is the purpose of the TSTB/BPL pair of instructions.

Note also that all status and data registers are addressed here in the relative mode.

3. Store the letters A, B, C, . . . , Z into a block of bytes starting at 1200.

Machine Language	Assembly Language	Comments
600: 012700	MOV #101,R0	;INITIALIZE CHARACTER TO A
602: 000101		
604: 012701	MOV #1200,R1	;INITIALIZE BYTE ADDRESS TO 1200
606: 001200		
610: 110021	AGAIN: MOVB R0,(R1)+	;STORE CHARACTER, INCREMENT ADDRESS
612: 020027	CMP R0,#132	;IS CHARACTER Z?
614: 000132		
616: 001402	BEQ EXIT	;IF SO, ALL DONE
620: 005200	INC R0	;ELSE, FORM NEXT CHARACTER
622: 000772	BR AGAIN	;RETURN FOR ANOTHER
624: ---	EXIT: ---	

Note: Constants representing addresses or characters (such as 1200 or 101) are here moved using immediate addressing. The autoincrement mode used in AGAIN automatically updates the address after each insertion. The INC instruction forms the next letter by adding 1 to the ASCII code of the preceding one.

4. The address of a number Y is stored in location 1000. Compute the number C of 1-bits in Y and put the result in R4.

Machine Language	Assembly Language	Comments
600: 005004	CLR R4	;INITIALIZE C TO 0
602: 017705	MOV @1000,R5	;PUT Y IN R5
604: 000172		
606: 001404	REPT: BEQ OUT	;IF Y=0, GET OUT
610: 100001	BPL SHIFT	;IF MSB=0, C UNCHANGED
612: 005204	INC R4	;ELSE, C=C+1
614: 006305	SHIFT: ASL R5	;SHIFT Y ONE BIT LEFT
616: 000773	BR REPT	;REPEAT
620: ---	OUT: ---	

Note: Y is retrieved using the relative deferred addressing mode. The number of 1-bits in Y is counted by counting the number of times Y becomes negative (i.e., bit 15 becomes 1) as Y is shifted left (using the ASL instruction).

5. Location 160 contains a single-operand instruction. An eight-word array starting at 750 contains eight addresses. Check the addressing

mode of the instruction; if it is D, jump to a point in the program whose address is stored in the (D + 1)st array word (i.e., in 750+2*D).

Machine Language	Assembly Language	Comments
600: 013700	MOV @#160,R0	;PUT INSTRUCTION IN R0
602: 000160		
604: 042700	BIC #177707,R0	;PICK UP 8*D
606: 177707		
610: 006000	ROR R0	;COMPUTE
612: 006000	ROR R0	; (8*D)/4 = 2*D
614: 000170	JMP @750 (R0)	;GO TO 750+2*D
616: 000750		

Note: The instruction is placed in R0, using the absolute mode (although relative mode could have worked just as well). The BIC clears all bits in R0 except bits 3, 4, and 5 (i.e., it clears all bits corresponding to the 1-bits in 177707); the result is D shifted 3 bits left (i.e., 8*D). To compute $2*D = (8*D)/4$, R0 is shifted 2 bits right (ROR is used, assuming that C is initially 0).

6. Insert a new number in the correct position of a sorted array of 16-bit 2's-complement numbers, whose first and last words are addressed A and B, respectively. Assume that the numbers are all positive; that they are arranged in order of ascending magnitude; that the new number N is in R0, and that the addresses A and B are stored in R1 and R2, respectively.

Machine Language	Assembly Language	Comments
600: 012741	MOV #177777,-(R1)	;PUT -1 IN A-2
602: 177777		
604: 005722	TST (R2)+	;PUT B+2 IN R2
606: 024200	LOOP: CMP -(R2),R0	;IS NEXT NUMBER $K \leq N$?
610: 003403	BLE INSERT	;IF SO, INSERT
612: 011262	MOV (R2),2(R2)	;ELSE, MOVE K UP ONE WORD
614: 000002		
616: 000773	BR LOOP	;RETURN FOR ANOTHER K
620: 010062	INSERT: MOVR0,2(R2)	;PUT N AHEAD OF K
622: 000002		

Note: Successive numbers K, starting at B and proceeding back toward A, are compared against N and moved one word ahead until a K is found that is less than or equal to N. When such a K is found (and the -1 stored in A-2 guarantees that this will eventually happen),

N is placed just ahead of this K. The comparison is done in LOOP with the autodecrement mode, where address decrementation is always done *before* access; thus, the array scanning must initiate with R2 holding B+2 rather than B. This initialization is done with the TST (R2)+ instruction, whose sole purpose is to increment the contents of R2 (change B to B+2) rather than test anything. Note that the index-mode address 2(R2), unlike the autoincrement address (R2)+, leaves the contents of R2 intact.

EXERCISES

4.1 Each one of the following instructions is located in address 500. Before each is executed, (R0) = 100, (76) = 176, (100) = 200, (176) = 276, and (200) = 500. Translate each instruction into machine language and determine the contents of R0 and R1 after each is executed.

- | | |
|--------------------|---------------------|
| (a) MOV R0,R1 | (b) MOV (R0),R1 |
| (c) MOV (R0)+,R1 | (d) MOV @(R0)+,R1 |
| (e) MOV -(R0),R1 | (f) MOV @-(R0),R1 |
| (g) MOV 100(R0),R1 | (h) MOV @100(R0),R1 |
| (i) MOV #100,R1 | (j) MOV @#100,R1 |
| (k) MOV 100,R1 | (l) MOV @100,R1 |

4.2 What does the following instruction accomplish?

```
MOV -(PC),-(PC)
```

4.3 What is the octal contents of R0 when the following program halts?

```
MOV #1,R0
MOV (PC),-(PC)
DEC R0
HALT
```

4.4 Location 1000 contains the octal number 42356. Assuming that the following program starts at 500, find the octal contents of R0 after each instruction is executed.

```
MOVB @#1001,R0
BIC #770,R0
MOV 500(R0),R0
SUB #123456,R0
```

- 4.5 The following program starts at address 500. Translate it into machine language.

```

CMP (R3),#123
BLT 650
MOVB @(R4)+,701
BIC R5,-(R0)
CMP @100,-350(PC)
BEQ 400
SUB @22(SP),(R2)+
COMB @#755
JMP @-(R1)

```

- 4.6 The following machine language program starts at address 500. Translate it into assembly language.

```

022733
000456
003756
012146
126467
000160
000273
001104
043777
000666
177652
160750
067215
177773

```

- 4.7 The following machine language program starts at address 0. What does it do?

```

016700
000022
016701
000020
105710
100376
105711
100376
022021
111011
000765
177560
177564

```

- 4.8 The following machine language program starts at address 0. What is the octal contents of 0 when the program halts?

```

062737
040000
000000
100774
000000

```

- 4.9 The following machine language program starts at address 0. After 5 instructions are executed, what are the octal contents of the words addressed 0, 2, 4, . . . , 32 and of PC?

```
066737
000000
000004
062737
000004
000000
012767
000002
177770
163777
000006
177766
000137
000000
```

- 4.10 Write a machine language program (starting at 500) which accepts an octal digit n from the keyboard and then rings a bell n times.
- 4.11 Write a machine language program (starting at 500) which echoes keyboard characters in such a way that all lowercase letters are capitalized.
- 4.12 Bytes 100 through 355 are filled with 8-bit 2's-complement numbers. Write a machine language program (starting at 500) which puts in R0 the largest of these.
- 4.13 Words 2000 through 3776 are filled with 1000₈ 16-bit 2's-complement numbers. Write a machine language program (starting at 500) which puts in R0 the average of these numbers (with the fractional part truncated).

ASSEMBLY LANGUAGE PROGRAMMING



In this chapter we will explain what assembly language is and provide some rudimentary facts on the MACRO-11 assembler. An example of a complete assembly language program will be given and some coding hints be offered.

5.1 ASSEMBLY LANGUAGE VERSUS MACHINE LANGUAGE

For any but the most trivial tasks, programming in pure binary or octal code (i.e., in *machine language*) is an onerous and frustrating exercise. The major disadvantages of this form of programming are:

1. Instructions are difficult to encode and interpret. The programmer must memorize op codes and mode numbers before he or she can

attain any fluency in machine language. Branch instructions are especially messy to encode.

2. Program modification is very difficult. If a bug is discovered which requires the insertion or deletion of an instruction, the entire program may have to be reviewed and many instructions corrected before the modification can be safely carried out.

To circumvent these difficulties, most manufacturers permit the user to write programs in *assembly language*, a language essentially equivalent to machine language, except that:

1. Op codes are not written numerically but mnemonically.
2. Addresses need not be specified numerically, but can be written symbolically.

The program that translates the assembly language program (or *source program*) into a machine language program (or *object program*) is called an *assembler* (the translation process is called *assembly*). The PDP-11 has a number of different assemblers; the one that will be used in this text is called *MACRO-11*.

A good assembler must be flexible and resourceful enough to make the task of program writing simple and efficient. Here are some of the features included in MACRO-11 toward this end:

1. The user is permitted to specify constants in nonoctal form (e.g., in decimal or binary).
2. The user is permitted to intersperse the program with comments.
3. The assembler lists the program in both assembly and machine languages.
4. The assembler issues diagnostic messages when the program contains syntactic errors.
5. The assembler provides for conditional assembly, repeated assembly, and macros. (These facilities will be explained in a later chapter.)

5.2

ASSEMBLY LANGUAGE DIRECTIVES

Unless otherwise specified, all numbers appearing in an assembly language program are assumed by the assembler to be octal. However, using the following conventions, one can also include in a program decimal and binary numbers, as well as octal and 1's-complement numbers:

n. or †Dn	denotes a decimal n
†Bn	denotes a binary n
†On	denotes an octal n
†Cn	denotes the 1's complement of n

The ASCII code for a character p can be written as 'p and the ASCII code for the character pair p₁p₂ as "p₁p₂".

The operation of the assembler can be controlled by the user by means of *directives* interspersed among the assembly language instructions. Directives are instructions addressed to the assembler; they are obeyed during *assembly time* and not during *run time* (i.e., not during the actual execution of the program)!

Here are some often-used directives:

- | | |
|-------|---|
| .WORD | d ₁ ,d ₂ ,... ,d _r |
| .BYTE | d ₁ ,d ₂ ,... ,d _r |

cause the assembler to store the data (numbers or characters) d₁,d₂, . . . ,d_r in consecutive words or bytes, respectively. For example,

```
.WORD 35,18.,'90
.BYTE †B1011,'K
```

will result in storing, in four consecutive words, the octal numbers

```
000035
000022
030071
045413
```

- .ASCII /str/

causes the assembler to store the ASCII code of string "str" in consecutive bytes (where / represents any character not in "str," except < and =). A nonprinting character with ASCII code n can be specified as <n>. For example,

```
.ASCII /THE ANSWER IS/
.ASCII /THE BELL WILL NOW RING/<7>
.ASCII *COMPUTATION OF X/Y* <15> <12>
```

The directive

```
.ASCIZ /str/
```

is the same as .ASCII, but a zero byte is inserted after the last character of “str” as a terminator.

3. `.BLKW n`
`.BLKB n`

causes the assembler to reserve a storage block of n words or bytes, respectively.

4. `.RADIX r`

instructs the assembler to regard all subsequent numbers as base-r numbers (r = 2, 4, 8, or 10). For example,

```
.RADIX 10
```

will result in the interpretation of all subsequent numbers as decimal (with no need for prefix ↑D or suffix .), unless otherwise specified by means of a prefix ↑B, ↑O, or ↑C. (This directive can be canceled by another .RADIX.)

5. The “assignment directive”

```
sym = exp
```

causes the assembler to assign the symbol “sym” the value of expression “exp.” For example,

```
KBSTAT=177560
```

assigns the value 177560 to KBSTAT. Henceforth, every time the assembler encounters the symbol KBSTAT, it replaces it with 177560. A subsequent directive,

```
KBDATA=KBSTAT+2
```

assigns to the symbol KBDATA the value 177562.

A symbol can be reassigned any number of times in a program. For example, a directive

```
I=1000
```


can be followed later by

```
I=I+5
```

Between the two directives `I` will be taken as the constant 1000, but after the second one, as the constant 1005.

6. The symbol `.` (“dot”) refers to the assembler variable (called *location counter*), which, at any time during assembly, holds the address of the word into which the next instruction is to be assembled.* Its initial value is 0. For reasons to be explained later, all programs should start with

```
LC=.
```

Thereafter, if we wish a section of source program to be assembled starting at address α , we issue the directive

```
.=  $\alpha$ +LC
```

(which will force “dot” to assume the value α).

7.

```
.EVEN
.ODD
```

will force “dot” to become the next even address (if it is odd), or the next odd address (if it is even), respectively.

8. A program usually starts with the directive

```
.TITLE name
```

which causes the assembler to head the program listing with the title “name.”

9. The last statement of every program must be

```
.END sym
```

where “sym” is the symbolic address of the program’s starting point. With this directive, the “loader” (the program which loads the ma-

*“Dot” can also be used to specify addresses. For example, “jump 3 words ahead” can be written as `JMP .+6`; “move current address to SP” can be written as `MOV #.,SP`.

chine language program into the CM) will automatically start the program at address “sym.”

5.3

ASSEMBLY LANGUAGE PROGRAM FORMAT

A source program is composed of a sequence of lines, each containing a single *statement*. A statement may be composed of as many as four *fields*, named and formatted as follows:

Label: Operator Operand ;Comment

For example:

```
LOOP:  MOV R0,@#177566      ;PRINT CHARACTER
```

The operator and operand fields must be separated by at least one blank or horizontal tab, but otherwise the spacing is arbitrary. However, since the teletype’s horizontal tabs are usually set every eight columns (starting with column 1), it is convenient to adopt the following standard format:

Field	Begins at column
Label	1
Operator	9
Operand	17
Comment	33

The label, which assigns a symbolic address to the statement, needs to appear only when the statement is referred to symbolically. A symbol may be of any length, but only the first six characters (letters and/or digits*) are significant. Two or more labels having the same first six characters result in an error message.

The comment field is optional, but should be used generously if a well-documented program is to result. (A line consisting entirely of commentary may start with ; in any column.)

*\$ and . can also be used, but the beginner is advised to avoid them.

As a matter of convention, all source programs in this text will be organized as follows:

```

Col. 1          9      17          33
               .TITLE Program title
               ;
               ;
               ; (Description of program)
               ;
               LC=.
               . =4+LC
               .WORD 6,0,12,0          ;INITIALIZE ERROR VECTORS
               . =500+LC                ;ALLOW FOR STACK SPACE
START:         MOV   PC,SP
               TST  -(SP)              ;INITIALIZE SP TO START
               ;
               ;
               ; (Program)
               ;
               .END START

```

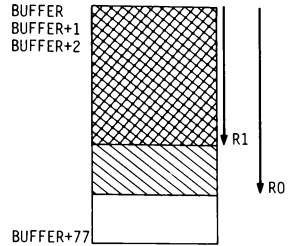
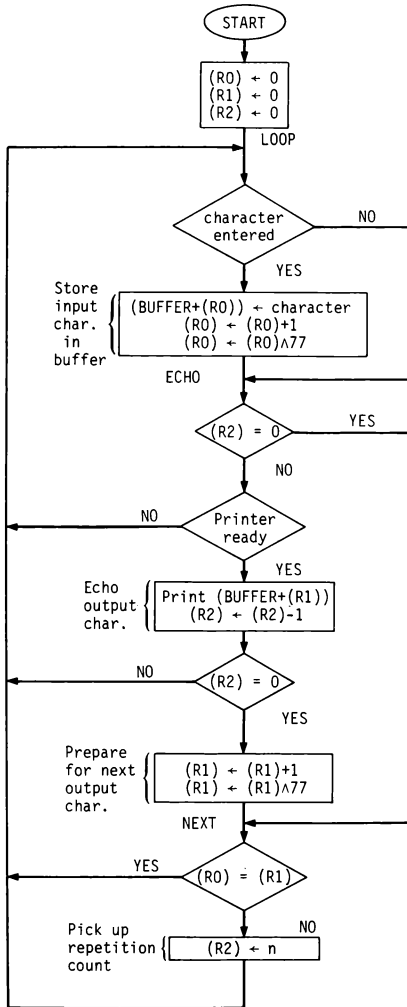
With this format, programs will start at 500 with the initialization of SP to 500. The reason for this and for the “error vectors” will be clarified in later chapters.

5.4 EXAMPLE: MULTIPLE ECHO

We wish to write a program which “echoes” (types back) every typed-in character n times. Since echoes of a character may not be completed before the next one is typed in, the program should set up a “buffer” (i.e., a temporary storage block) for storing characters awaiting printout. We shall assume that the backlog can never exceed 64_{10} characters.

The program employs a “buffer input pointer,” which, at any time, points to the next free buffer byte (i.e., the byte in which the next typed-in character is to be stored), and a “buffer output pointer,” which, at any time, points to the next buffer byte to be echoed. Typically, the output pointer lags behind the input pointer; when the former finally catches up with the latter, the buffer is “empty” (no character awaits printout).

For efficient operation, it is proposed to use a “circular buffer,” where the byte with the highest address is followed by the byte with the lowest address. In this manner, only 64_{10} bytes are required for storage, regardless of the number of characters typed in. An implementation of such a buffer



(R0) = buffer input pointer
 (R1) = buffer output pointer
 (R2) = repetition count n

Figure 5.1 Flowchart for multiple-echo program.

requires that when a buffer pointer reaches 63_{10} (77_8), its next value be reverted to 0. One way of accomplishing it is to mask out all but the right-most 6 bits of the pointer after every incrementation.

Figure 5.1 shows the flowchart and Figure 5.2 the assembler listing of the source and object codes of the program. (The symbols KBSTAT, KBDATA, etc., are defined merely to improve program readability.) In this program N was selected to have the value 5.

Note that every address which in the source program has the form

symbol \pm constant

(where either the symbol or the constant may be absent) is assembled by MACRO-11 into a relative address (mode 6). For example,

```
MOVB  KBDATA,BUFFER(R0)
```

(located at 520) is assembled as

```
116760
177036  (177562 - 524 = 177036)
000602
```

(See Section 4.6.)*

Remember that every symbolic address used by an instruction must be defined as a label, or by means of an assignment directive (sym = exp), or by means of a .GLOBL directive (to be discussed in a later chapter). A symbol not thus defined (or defined more than once) will result in a diagnostic message.

The masking operation is done in the program by means of the BIC (“bit clear”) instruction. BIC s,d clears to 0 all those bits in d which correspond to 1-bits in s, leaving all other bits intact. Thus, BIC #177700,R0 will clear all but the rightmost 6 bits of R0. (The “bit set” instruction BIS s,d sets to 1 all those bits in d which correspond to 1-bits in s, leaving all other bits intact.)

*The meaning of the apostrophe attached to some of the assembled addresses will be clarified in a later chapter.

```

1          .TITLE MULTTECHO
2          ; ECHO EACH TYPED-IN CHARACTER N TIMES. USE CIRCULAR BUFFER OF
3          ;64 (DECIMAL) BYTES FOR STORING AWAITING CHARACTERS.
4          000000'LC=.
5          000004'=.4+LC
6 000004 000006      .WORD  6,0,12,0      ;INITIALIZE ERROR VECTORS
          000000
          000012
          000000
7          000500'=.500+LC
8 000500 010706 START: MOV  PC,SP      ;ALLOW FOR STACK SPACE
9 000502 005746      TST  -(SP)      ;INITIALIZE SP TO START
10         ;
11         177560 KBSTAT=177560
12         177562 KBDATA=177562
13         177564 PRSTAT=177564
14         177566 PRDATA=177566
15         ;
16 00504 005000      CLR  R0      ;INITIALIZE BUFFER INPUT POINTER
17 00506 005001      CLR  R1      ;INITIALIZE BUFFER OUTPUT POINTER
18 00510 005002      CLR  R2      ;INITIALIZE REPETITION COUNT
19 00512 105767 LOOP: TSTB  KBSTAT    ;CHARACTER ENTERED?
          177042'
20 00516 100006      BPL  ECHO      ;IF NOT, KEEP ECHOING
21 00520 116760      MOVB  KBDATA,BUFFER(R0) ;IF S0, STORE CHAR IN BUFFER+(R0)
          177036'
          000602'
22 00526 005200      INC  R0      ;(R0)=(R0)+1
23 00530 042700      BIC  #177700,R0 ;ZERO R0 IF > 77
          177700
24 00534 005702 ECHO: TST  R2      ;IF MULTIPLE ECHO TERMINATES,
25 00536 001413      BEQ  NEXT     ;PREPARE FOR NEXT OUTPUT CHARACTER
26 00540 105767      TSTB  PRSTAT   ;OTHERWISE, IS PRINTER READY
          177020'
27 00544 100362      BPL  LOOP     ;IF NOT, ACCEPT NEXT CHARACTER
28 00546 116167      MOVB  BUFFER(R1),PRDATA ;IF S0, ECHO OUTPUT CHARACTER
          000602'
          177012'
29 00554 005302      DEC  R2      ;(R2)=(R2)-1
30 00556 001355      BNE  LOOP     ;IF (R2).NE.0, ACCEPT NEXT CHAR
31 00560 005201      INC  R1      ;ELSE, (R1)=(R1)+1
32 00562 042701      BIC  #177700,R1 ;ZERO R1 IF > 77
          177700
33 00566 020001 NEXT: CMP  R0,R1    ;IF (R0)=(R1) (BUFFER EMPTY),
34 00570 001750      BEQ  LOOP     ;ACCEPT NEXT CHARACTER
35 00572 016702      MOV  N,R2     ;(R2)=(N) (REPETITION COUNT)
          000002
36 00576 000745      BR   LOOP     ;ACCEPT NEXT CHARACTER
37         ;
38 00600 000005 N:   .WORD  5      ;REPETITION COUNT
39 00602 .BUFFER:  .BLKB  64.     ;BUFFER SPACE: 64 DECIMAL BYTES
40         000500' .END  START

```

Figure 5.2 Multiple-echo program.

It is useful to mention some common mistakes made by beginners while learning assembly language.

When uncertain whether something is correct, the programmer should ask himself or herself what the instruction would be in machine language. For example, consider the statement

```
MOV R1+4,R0
```

What was desired was to have 4 added to the contents of R1 and stored in R0; that is, $(R0) \leftarrow (R1) + 4$. What was written is perfectly legal but its interpretation is to move the contents of R5 to R0. The correct solution requires two instructions:

```
MOV R1,R0
ADD #4,R0
```

Suppose that it is desirable to use index mode and increment the register afterward. One may be tempted to issue

```
CLRB 1000(R0)+
```

Although this might be a very useful addressing mode to have, it does not exist! The solution again requires two instructions:

```
CLRB 1000(R0)
INC R0
```

There is a world of difference between

```
MOV #500,R0
```

and

```
MOV 500,R0
```

The first means

```
 $(R0) \leftarrow 500$ 
```

while the second means

(R0)←(500)

Suppose it is necessary to clear locations 1000, 1002, and 1004. The following is correct:

```
MOV #1000,R0
CLR (R0)+
CLR (R0)+
CLR (R0)
```

The first instruction puts the address 1000 into R0. The second instruction then clears location 1000 and adds 2 to R0. The third and fourth instructions clear locations 1002 and 1004, respectively.

Suppose that the contents of location 1000 is 500, and consider the following sequence of instructions:

```
MOV 1000,R0
CLR (R0)+
CLR (R0)+
CLR (R0)
```

The first instruction moves the contents of location 1000 to R0. R0 now contains the value 500. The next three instructions would clear locations 500, 502, and 504. Notice the difference between this and the previous example.

An additional word of caution: you will *never* use immediate addressing as a destination. For example,

```
CLR #500
MOV #500,#600
```

hardly makes sense. (Why?) If you see a # in the destination, you almost certainly are not writing what you want.

All symbolic addresses other than R0, R1, . . . , R7, SP (which may replace R6) and PC (which may replace R7) must be defined within the program. For example, CLR PSR is wrong unless preceded by PSR = 177776.

Note the difference between the two instructions,

SAM: .WORD 0

and

CLR SAM

The first is an assembler directive, instructing MACRO-11 to insert 0 in location SAM of the object program. The second is a PDP-11 CP instruction executed during program run. While .WORD 0 is obeyed only once (during assembly time), the CLR instruction can be placed as part of a loop to clear SAM each time the loop is entered during run time. (MOV #0,SAM is equivalent to CLR SAM but is more wasteful in both time and space.)

The compare (CMP) and subtract (SUB) instructions treat their operands in an opposite fashion. In SUB the source is subtracted from the destination, while in CMP the destination is subtracted from the source. For example, SUB R0,R1 forms (R1)-(R0) (and stores it in R1), while CMP R0,R1 forms (R0)-(R1) (with no register altered).

Beware of executing word instructions on operands which reside in odd addresses. For example, do not issue CLR (R0) unless you are sure that R0 will always contain an even address.

It is not enough to write a program that merely runs correctly. A program should be well organized and adequately documented. The reader is referred to Appendix G, "Notes on Programming Style," for further details.

EXERCISES

- 5.1 Modify the multiple-echo program of Figure 5.2 to accommodate a buffer size of 128_{10} bytes.
- 5.2 In as many different ways as you can think of, direct the assembler to store the octal constant 050520 in a memory word (e.g., .WORD 050520).
- 5.3 Translate the following (nonsensical) assembly language program into machine language.

```

LC=.
.=500+LC
JOE=100
SAM=JOE+20
START:  MOV     #SAM,JOE
        JMP     START
        .ASCIZ  $/XYZ/$ <130>
        .BYTE  +B1,19., 'A'
        .RADIX 10
        .BLKB  11
        .EVEN
        .WORD  "12,+013,+C+B11010,333
        .END   START

```

- 5.4 What are the octal contents of the eight general-purpose registers after the following program halts?

```

LC=.
.=500+LC
START: CLR R0
        MOVB SAM,R1
        MOV #'W,R2
        MOV PC,R3
        MOV SAM,R4
        BIC ANN,R4
        MOV #JOE,SP
        MOV ANN,(SP)+
JOE: CLR R5
SAM: .WORD 123456
      HALT
ANN: .WORD 012705
      .END START

```

- 5.5 What does the following program do? What are the octal contents of R0, R1, and R2 when it halts?

```

LC=.
.=500+LC
START: MOV #A3,R0
        CLR R2
A1: MOVB (R0)+,R1
      BIC #177600,R1
      CMP R1,#'0
      BLT A2
      CMP R1,#'7
      BGT A2
      SUB #'0,R1
      ASL R2
      ASL R2
      ASL R2
      ADD R1,R2
      BR A1
A2: HALT
A3: .ASCII '010706'
A4: .BYTE 15,12
      .END START

```

- 5.6 Write an assembly language program that divides an integer stored in A by an integer stored in B (with the fractional part truncated). The quotient is to be left in R0 and the remainder in R1.
- 5.7 Write an assembly language program that regards the word K as consisting of eight 2-bit “quarter-bytes” and counts the number of such quarter-bytes which have the value 3 (i.e., 11_2). The count is to be left in R0.
- 5.8 Write an assembly language program that prints out the octal contents of a word whose address is found in ADDR. (For example, if ADDR

- contains 123, and 123 contains 456, then the program should print out 000456.)
- 5.9 One hundred (decimal) 16-bit 2's-complement numbers (not necessarily different from each other) are stored in order of ascending magnitude in an array starting at address TABLE. Place in R0 the number of occurrences of the array number that occurs most often.
- 5.10 Fifty (decimal) numbers, each between 0 and 100_{10} , are stored in a 50-word array starting at address GRADE. The contents of $\text{GRADE}+i$ represents the grade of student number $i+1$. Write an assembly language program that sets up a 50-word array starting at address RANK, where the contents of $\text{RANK}+i$ is the rank of student number $i+1$ in the class of fifty. (The rank of a student equals one plus the number of students whose grades exceed that student's grade.)
- 5.11 Consider a chessboard whose rows and columns are numbered 0 through 7. Write an assembly language program that accepts an input of the form ij ($0 \leq i \leq 7$, $0 \leq j \leq 7$) and prints out all possible positions of a bishop starting at the intersection of row i and column j . For example, when the input is 42, the output should be

```

      0 1 2 3 4 5 6 7
0
1
2 *
3 * *
4 *
5 * *
6 * *
7 *
```

STACKS AND SUBROUTINES

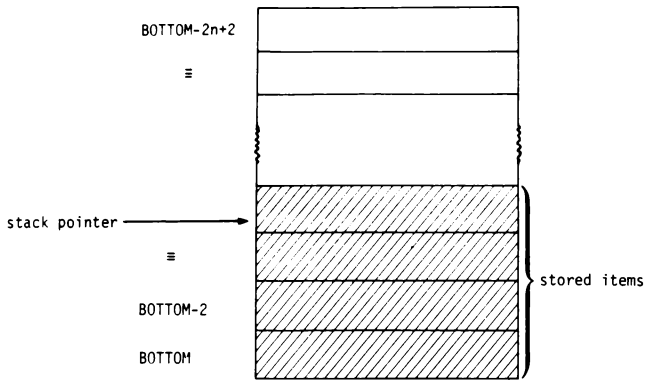


In this chapter we shall introduce the concept of a stack and see how stacks are implemented in the PDP-11. Our attention will then turn to the “system stack” and its most important application, the linking of subroutines. The chapter will close with a discussion of recursive subroutines.

6.1 STACKS

A *stack* is a data structure where the data items are retrieved in the reverse order in which they are stored (“last in, first out”). In this respect, a stack is analogous to a dish well in a cafeteria, where the only plate available is the one most recently added.

In the PDP-11, a stack is usually implemented as a block of n consecutive words, in the following manner:



Items are added starting at address *BOTTOM*, proceeding “upward” toward lower addresses. At any time, the *stack pointer* contains the address of the current “top” of the stack.

The three basic stack operations are:

Initialization:

$$(\text{stack pointer}) \leftarrow \text{BOTTOM}+2$$

“*Push*” operation (place contents of source on top of stack):

$$\begin{aligned} (\text{stack pointer}) &\leftarrow (\text{stack pointer}) - 2 \\ ((\text{stack pointer})) &\leftarrow (\text{source}) \end{aligned}$$

“*Pop*” operation (remove the top of the stack and copy its contents into destination):

$$\begin{aligned} (\text{destination}) &\leftarrow ((\text{stack pointer})) \\ (\text{stack pointer}) &\leftarrow (\text{stack pointer}) + 2 \end{aligned}$$

The autoincrement and autodecrement modes of addressing make the general registers (except PC) ideal as stack pointers. If *Ri* is chosen as a stack pointer, the basic stack operations can be coded very simply as follows:

```

Initialization:  MOV #BOTTOM+2,Ri
Push:           MOV source,-(Ri)  [denoted (↓(Ri)) ← (source)]
Pop:           MOV (Ri)+,destination  [denoted (destination) ← ((Ri)↑)]

```

(Byte stacks can be implemented in a similar manner, except that `MOVB` rather than `MOV` is used in the push and pop operations.)

In most of our programs we shall allocate the stack space just ahead of the program, so that `BOTTOM+2` is the program's starting address `START`. We shall also use `R6 (SP)` as the stack pointer. With these conventions we shall have:

```

Initialization:  START:  MOV #START,SP
                   or:   START:  MOV PC,SP
                               TST -(SP)
Push:           MOV source,-(SP)
Pop:           MOV (SP)+,destination

```

In normal use, data is pushed onto the stack when it is to be saved for future use, and popped when it is no longer needed. In this manner, memory space is allocated for data only as it is needed, and can be released for other purposes at other times.

The size n of the stack is an estimate of the greatest number of data items which the stack may be called upon to accommodate at any given time. An attempt to push an item onto the stack when it is full (i.e., after the stack pointer reaches `BOTTOM-2n+2`) results in *stack overflow*. At the other extreme, an attempt to pop an item from the stack when it is empty (i.e., when the stack pointer points to `BOTTOM+2`) results in *stack underflow*. Both stack overflow and underflow may have disastrous results and should be watched for by the program.

6.2

EXAMPLE: BACKWARD ECHO

We wish to write a program that accepts a line of characters from the teletype and then echoes it backward. For example, a type-in of

```
ABC...XYZ ↵
```

(where ↵ denotes carriage return) would result in a printout of

```
ZYX...CBA ↵
```

Since characters are printed out in the reverse order in which they are typed in, a stack is the most natural data structure for this program. The characters are pushed onto the stack as they arrive, until a carriage return is typed in, at which point the stack gets popped and printed until it is emptied. (In order to prepare the printer for another line, the two bottom words of the stack should be initialized with line-feed and carriage-return codes.)

Figure 6.1 shows the flowchart and Figure 6.2 the assembler listing of the program.

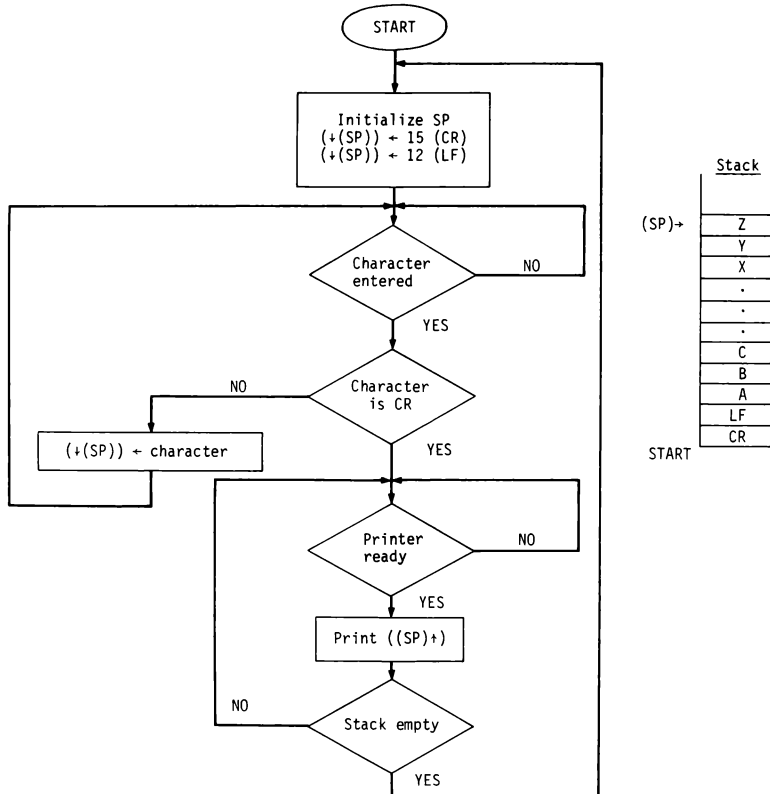


Figure 6.1 Flowchart for backward-echo program.

```

        .TITLE    BACKWARD
; ACCEPT A LINE OF CHARACTERS FROM TELETYPE AND ECHO IT BACKWARDS.
LC=.
.=4+LC
        .WORD    6,0,12,0        ;INITIALIZE ERROR VECTORS
.=500+LC
START:  MOV      PC,SP            ;ALLOW FOR STACK SPACE
        TST      -(SP)          ;INITIALIZE SP TO START
;
KBSTAT=177560
KBDATA=177562
PRSTAT=177564
PRDATA=177566
;
        MOV      #15,-(SP)      ;CR CODE TO STACK
        MOV      #12,-(SP)      ;LF CODE TO STACK
LOOP:   TSTB    KBSTAT          ;CHARACTER ENTERED?
        BPL     LOOP           ;NO: KEEP IDLING
        MOV     KBDATA,RO       ;(RO)=CHARACTER
        BIC     #177600,RO      ;CLEAR ALL BUT CODE BITS
        CMP     RO,#15         ;IS CHARACTER CR?
        BEQ     OUT            ;IF SO, GO TO OUTPUT
        MOV     RO,-(SP)        ;ELSE, PUSH CHARACTER ON STACK
        BR      LOOP           ;AND RETURN FOR NEXT CHARACTER
OUT:    TSTB    PRSTAT          ;IS PRINTER READY?
        BPL     OUT            ;IF NOT, KEEP IDLING
        MOV     (SP)+,PRDATA    ;IF SO, POP NEXT CHAR. AND PRINT
        CMP     SP,#START      ;IS STACK EMPTY?
        BEQ     START          ;IF SO, ACCEPT NEW LINE
        BR      OUT            ;IF NOT, GO ON PRINTING
        .END    START

```

Figure 6.2 Backward-echo program.

6.3 SUBROUTINES

It is often the case that similar segments of code are required at various points of a program, the only difference between them being the values assigned to some key variables. In this case a great deal of labor as well as program space are saved by writing all these segments as a single *subroutine*, with the key variables serving as the subroutine's *arguments* (or *parameters*). The subroutine is *called* by the *calling program* at the points at which it is needed, and *returns* to the calling program after it completes execution.

It should be pointed out that even when the saving of program space is of no importance, the generous use of subroutines is highly desirable. It helps the user write his or her program in a “top-down” (or “modular”) fashion, that is, in successive steps which proceed from the gross to the detailed. This approach to programming is not only intellectually simpler, but results in programs that are easier to comprehend, debug, and maintain.

To the extent that a subroutine may be used by different programs writ-

ten by different users, it should be carefully organized and amply documented and annotated. In addition, a subroutine should always be written as a *pure procedure* (or in *reentrant code*)—that is, the subroutine should in no way modify its own instructions during execution. This is imperative if the subroutine is loaded into the memory and then called successively by different independent programs (which expect to find it in its original form).

Subroutine calls entail two essential actions:

1. *Linkage*: Transmitting to the subroutine the address in the calling program to which it must return.
2. *Argument transmission*: Supplying the subroutine with the values for its arguments.

In the following sections we shall see how these actions can be implemented in the PDP-11.

6.4

SUBROUTINE CALL AND RETURN

A convenient linkage mechanism is provided in the PDP-11 by the JSR (“jump to subroutine”) instruction. To call subroutine SUB (i.e., the subroutine whose entry address is SUB), issue

```
JSR Ri,SUB
```

where Ri is any general register* (referred to as the *linkage register*). The effect of this instruction is the same as if the following sequence of instructions were executed in a single cycle:

```
MOV Ri,-(SP)
MOV PC,Ri
JMP SUB
```

Thus, JSR saves the contents of Ri by pushing its contents onto the stack whose pointer is SP; it then uses Ri for saving the contents of PC (i.e., the return address); finally, it jumps to SUB. (Because SP is automatically

*However, the use of R6 here should be avoided.

regarded by the system as a stack pointer, the stack pointed to by SP is called the *system stack*.)

Exit from a subroutine is done with RTS (“return from subroutine”) instruction. If the calling program used Ri as the linkage register when it called the subroutine, then to return from this subroutine back to the calling program, issue

```
RTS Ri
```

The effect of this instruction is the same as if the following sequence of instructions were executed in a single cycle:

```
MOV Ri,PC
MOV (SP)+,Ri
```

Thus, RTS puts in PC the contents of Ri (i.e., the return address) and then restores the original contents of Ri by popping the system stack.

Example

	Machine Language	Assembly Language	Resulting Action
Calling program	1000: 004567	JSR R5,SAM	(↓(SP))←(R5),(R5)←(PC)=1004
	1002: 000060		
	1004:	(Return point)	
	.	.	
Subroutine	.	.	
	1064:	SAM: (Entry point)	
		RTS R5	(PC)←(R5)=1004; (R5)←((SP)↑)

□

In many cases it is convenient to use R7 (PC) as a linkage register. In this case, the call becomes

```
JSR PC,SUB
```

which is equivalent to

```
MOV PC,-(SP)
MOV PC,PC
JMP SUB
```

Thus, JSR simply pushes the return address onto the system stack (without tampering with any of registers R1 through R5) and enters SUB. The return is done with

```
RTS PC
```

which is equivalent to

```
MOV PC,PC
MOV (SP)+,PC
```

Thus, RTS simply pops the return address from the system stack (at whose top, presumably, this address still resides) onto PC.

Example

	Machine Language	Assembly Language	Resulting Action
Calling program	1000: 004767	JSR PC,SAM	$(\downarrow(SP)) \leftarrow (PC) = 1004$
	1002: 000060		
	1004:	(Return point)	
	.		
Subroutine	.		
	.		
	1064:	SAM: (Entry point)	
	.		
		RTS PC	$(PC) \leftarrow ((SP)\uparrow) = 1004$
			□

6.5 ARGUMENT TRANSMISSION

A number of methods are available for transmitting arguments from a calling program to a subroutine, each with its own advantages and disadvantages. The particular method assumed by a subroutine should always appear in the commentary of that subroutine.

Figures 6.4 to 6.7 illustrate some possible methods for transmitting arguments to a subroutine MULT which executes $(C) \leftarrow (A) * (B)$ [where $(B) \geq 0$]. The flowchart of MULT is shown in Figure 6.3. (The multiplication algorithm used in MULT is quite inefficient, but will do for an example.)

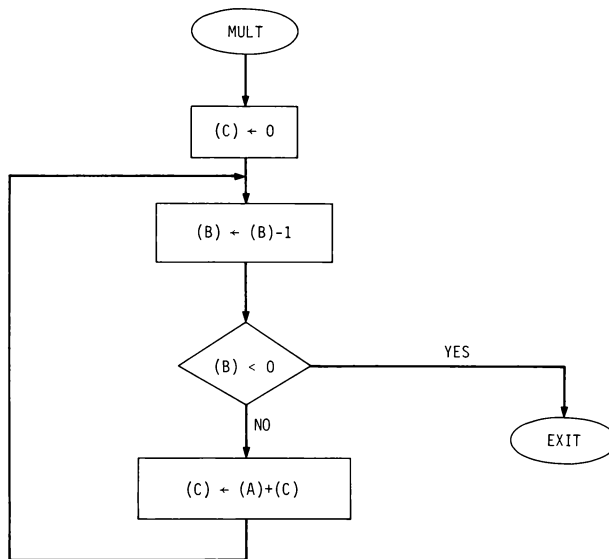


Figure 6.3 Flowchart for MULT subroutine.

```

MOV A,R1      ;(R1)=(A)
MOV B,R2      ;(R2)=(B)
JSR PC,MULT   ;CALL MULT
MOV R3,C      ;(C)=(R3)
.
.
A: .BLKW 1
.
.
B: .BLKW 1
.
.
C: .BLKW 1
.
MULT: CLR R3
LOOP: DEC R2
      BMI EXIT
      ADD R1,R3
      BR LOOP
EXIT:  RTS PC
      ;RETURN
      } (R3)+(R1)*(R2)
  
```

Figure 6.4 Argument transmission – method I.

```

.
.
.
A: .BLKW 1
B: .BLKW 1
C: .BLKW 1
.
.
MULT: MOV (R5)+,R1 ;(R1)=(A)
      MOV (R5)+,R2 ;(R2)=(B)
      CLR R3
LOOP: DEC R2
      BMI EXIT
      ADD R1,R3
      BR LOOP
EXIT:  MOV R3,(R5)+ ;(C)=(R3)
      RTS R5        ;RETURN
      } (R3)+(R1)*(R2)
  
```

Figure 6.5 Argument transmission – method II.

Here are some comments on the various methods.

Method I (see Figure 6.4). The arguments are moved to any of the registers R1 to R5 before the call. This is perhaps the simplest method, but is practical only for a small number of arguments.

Method II (see Figure 6.5). The arguments are placed immediately after the call. The subroutine moves the arguments (whose location is transmitted via the linkage register) to any registers or CM locations it finds convenient. The main disadvantage is that arguments must be stored after the call, in the middle of the program.

Method III (see Figure 6.6). Similar to method II, except that the addresses of the arguments are placed immediately after the call; the arguments themselves may appear anywhere in the CM.

Method IV (see Figure 6.7). Arguments are listed in an array whose base address is transmitted to the subroutine via a general-purpose register R1 to R5. Advantageous for a large list of arguments, since the subroutine does not have to allocate separate storage to them. However, as a result of the index mode of addressing, reference to arguments is rather slow.

```

                                JSR R5,MULT      ;CALL MULT
                                .WORD A,B,C      ;ADDRESSES A,B,C ARE STORED HERE
.
.
A:  .BLKW 1
.
.
B:  .BLKW 1
.
.
C:  .BLKW 1
.
.
MULT:  MOV @(R5)+,R1      ;(R1)=(A)
        MOV @(R5)+,R2      ;(R2)=(B)
        CLR R3
LOOP:  DEC R2
        BMI EXIT
        ADD R1,R3
        BR LOOP
EXIT:  MOV R3,@(R5)+      ;(C)=(R3)
        RTS R5            ;RETURN

```

} (R3)+(R1)*(R2)

Figure 6.6 Argument transmission – method III.

```

        MOV #ARG,R5      ;(R5)=BASE ADDRESS OF ARGUMENT ARRAY
        JSR PC,MULT      ;CALL MULT
        .
        .
        .
ARG:    .BLKW 3          ;STORAGE FOR MULTIPLICAND,MULTIPLIER,PRODUCT
        .
        .
        .
MULT:   MOV (R5),R1      ;(R1)=(ARG)
        MOV 2(R5),R2     ;(R2)=(ARG+2)
        CLR R3
LOOP:   DEC R2
        BMI EXIT        } (R3)+-(R1)*(R2)
        ADD R1,R3
        BR LOOP
EXIT:   MOV R3,4(R5)     ;(ARG+4)+-(R3)
        RTS PC          ;RETURN

```

Figure 6.7 Argument transmission – method IV.

6.6 NESTED SUBROUTINES

It is often the case in modular programs that a subroutine calls a subroutine, which itself calls a subroutine, which itself calls a subroutine, and so forth. Such subroutines are said to be “nested.”

Since a subroutine may tamper with some general-purpose registers holding information essential to its calling program, the calling program is frequently obliged to save register contents in the CM before it issues a call. In particular, if the calling program is itself a subroutine, it must save the linkage information transmitted to it by *its* calling program. All register information must be restored by the calling program as soon as it regains control.

As the nesting “depth” increases, the amount of information requiring temporary storage also increases, and careful bookkeeping is required to decide where and when to move each item. In the PDP-11 this bookkeeping is made exceedingly simple by the system stack and the automatic linkage mechanism.

The idea is that all essential registers and linkage information be saved in (and restored from) the system stack. In this manner, whenever a calling program is ready to restore register contents or whenever a subroutine is ready to use linkage information (in order to return), they can find the

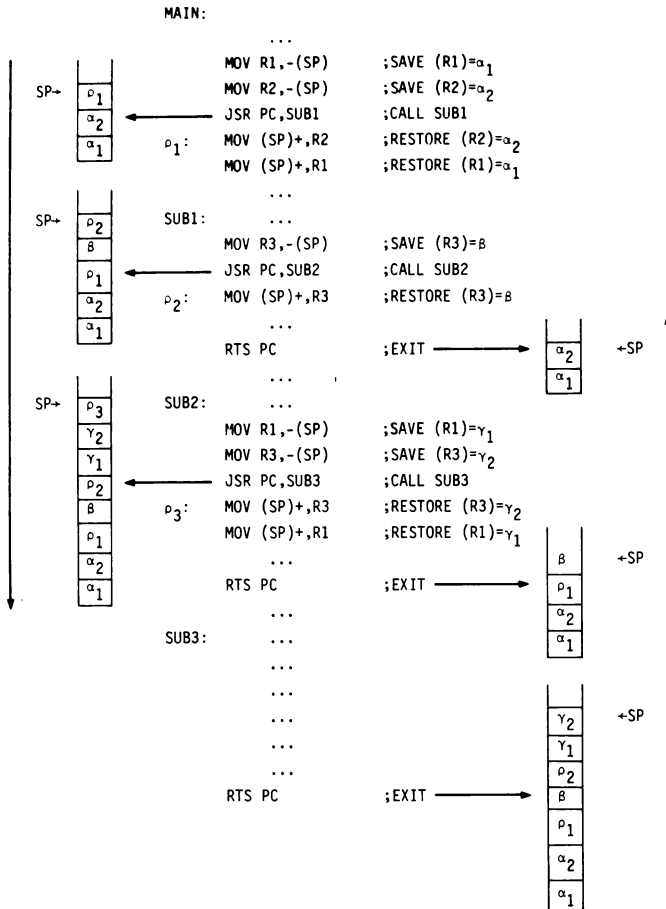


Figure 6.8 Nested subroutines.

necessary items right on top of the stack. Specifically, for every subroutine call a calling program contains the following successive steps:

1. Push contents of essential registers onto the system stack (using MOV Ri,-(SP) instructions).
2. Call the subroutine, employing PC as the linkage register (using the JSR PC,SUB instruction). This will automatically push the return address onto the system stack.
3. Restore the contents of essential registers by popping the system stack (using MOV (SP)+,Ri instructions).

Correspondingly, each subroutine should return with an RTS PC instruction (which pops the return address from the system stack).

Figure 6.8 illustrates this scheme. On the left, “snapshots” of the system stack are shown as MAIN calls SUB1, as SUB1 calls SUB2, and as SUB2 calls SUB3. On the right, the system stack is shown as SUB3 returns to SUB2, SUB2 to SUB1, and SUB1 to MAIN.

In the scheme above, the burden of protecting essential registers is placed on the calling program. Alternatively, the subroutine itself can assume this burden by saving, upon entry, all registers which are to be used by it and restoring them just before exit. Conveniently, the saving can be done by pushing the register contents onto the system stack (as in 1 above) and the restoring by popping the same contents from the system stack (as in 2 above).

6.7 RECURSIVE SUBROUTINES

In many cases it is simpler to define mathematical functions *recursively*, rather than directly. The *recursive definition* of a function $F(n)$, for all integers $n \geq n_0$, consists of:

1. *Basis*: specifying the values of $F(n_0)$, $F(n_0 + 1)$, . . . , $F(n_0 + k)$ explicitly.
2. *Induction step*: For all $n > n_0 + k$, specifying $F(n)$ in terms of any of the values $F(n_0)$, $F(n_0 + 1)$, . . . , $F(n - 1)$.

Examples

1. Defining the factorial function FACT(n) for all $n \geq 0$:

Basis: FACT(0) = 1

Induction step: FACT(n) = FACT($n - 1$) · n ($n > 0$)

For example:

$$\begin{aligned}\text{FACT}(4) &= \text{FACT}(3) \cdot 4 = \text{FACT}(2) \cdot 3 \cdot 4 = \text{FACT}(1) \cdot 2 \cdot 3 \cdot 4 \\ &= \text{FACT}(0) \cdot 1 \cdot 2 \cdot 3 \cdot 4 = 1 \cdot 1 \cdot 2 \cdot 3 \cdot 4 = 24\end{aligned}$$

2. Defining the Fibonacci number $\text{FIB}(n)$ for all $n \geq 1$:

Basis: $\text{FIB}(1) = 1, \text{FIB}(2) = 1$

Induction step: $\text{FIB}(n) = \text{FIB}(n-2) + \text{FIB}(n-1) \quad (n > 2)$

For example:

$$\begin{aligned}\text{FIB}(6) &= \text{FIB}(4) + \text{FIB}(5) = \text{FIB}(2) + \text{FIB}(3) + \text{FIB}(3) + \text{FIB}(4) \\ &= 1 + \text{FIB}(1) + \text{FIB}(2) + \text{FIB}(1) + \text{FIB}(2) + \text{FIB}(2) + \text{FIB}(3) \\ &= 1 + 1 + 1 + 1 + 1 + 1 + \text{FIB}(1) + \text{FIB}(2) = 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8\end{aligned}$$

3. Defining the set $\text{PERM}(n)$ of all permutations of the n -tuple $(a_1, a_2, \dots, a_n) \quad (n \geq 1)$:

Basis: $\text{PERM}(1) = \{(a_1)\}$

Induction step: $\text{PERM}(n) = a_n \circ (\text{PERM}(n-1)) =$ set of all n -tuples obtained by inserting a_n in all possible positions of all elements of $\text{PERM}(n-1) \quad (n > 1)$.

For example:

$$\begin{aligned}\text{PERM}(3) &= a_3 \circ (\text{PERM}(2)) = a_3 \circ (a_2 \circ (\text{PERM}(1))) \\ &= a_3 \circ (a_2 \circ (\{(a_1)\})) = a_3 \circ (\{(a_2, a_1), (a_1, a_2)\}) \\ &= \{(a_3, a_2, a_1), (a_2, a_3, a_1), (a_2, a_1, a_3), (a_3, a_1, a_2), (a_1, a_3, a_2), (a_1, a_2, a_3)\}\end{aligned}$$

□

A *recursive subroutine* is a subroutine that computes a recursive function. Correspondingly, it consists of a basis that produces $F(n)$ directly, and an induction step that consists of the subroutine *calling itself* with the argument n replaced with a “lower” argument (usually $n-1$). For example, the following is a self-explanatory recursive subroutine (written in the language PASCAL), which computes $\text{FACT}(n)$ *:

```

FUNCTION FACT(N: INTEGER): INTEGER;
  BEGIN
    IF N=0 THEN FACT := 1
    ELSE FACT := FACT(N-1)*N
  END;

```

*A PASCAL (or FORTRAN) subroutine that returns a single numerical value is called a “function.”

The way this subroutine operates is typical of all recursive subroutines: If the argument n is greater than 0, it keeps calling itself with successively decreasing values of n , until n is reduced to 0. At this point it keeps returning to itself (each time multiplying the returned value by n), with an ultimate return to the calling program.

Recursion is merely a special case of subroutine nesting and its implementation in the PDP-11 is as described in the preceding section.

6.8

EXAMPLE: TOWER OF HANOI

A classical example of a recursive subroutine is the one that solves the "Tower of Hanoi" puzzle. In this puzzle, a number of discs are stacked in decreasing size on a spindle A. They are to be moved to spindle C (stacked in the original order), using, if necessary, a spindle B for temporary storage. (See Figure 6.9.) In the moving process, the following two rules should be obeyed: (1) only one disc may be moved at a time (from any spindle to any other spindle); (2) at no time may a disc rest on top of a smaller one.

The subroutine for solving this puzzle is `HANOI(N,X,Y,Z)`, where the argument N is the number of discs and the arguments X , Y , and Z are the names of the spindles used as initial, temporary, and final spindles, respectively. Thus, for the situation depicted in Figure 6.9, a solution is obtained by calling `HANOI(5,A,B,C)`.

We shall name the N discs $1, 2, \dots, N$ (ordered from smallest to largest), and denote a movement of disc i from spindle u to spindle v by uiv .

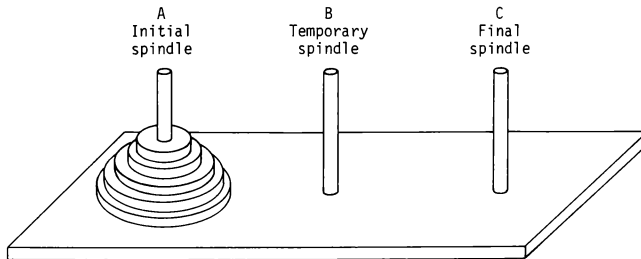


Figure 6.9 Tower of Hanoi puzzle.

HANOI(N,X,Y,Z), which prints out a sequence of moves denoted in this manner, can be described recursively as follows:

Basis: HANOI(1,X,Y,Z) prints out X1Z.

Induction step: HANOI(N,X,Y,Z) ($N > 1$) does the following:

1. Executes HANOI(N - 1,X,Z,Y).
2. Prints out XNZ.
3. Executes HANOI(N - 1,Y,X,Z).

As an example, Figure 6.10 describes schematically the execution of HANOI(3,A,B,C).

Figure 6.11 shows the listing of the program for solving the Tower of Hanoi puzzle. (For simplicity, we assumed that $N \leq 7$.) Besides the recursive subroutine HANOI described above, the program contains the subroutines INPUT, OUTPUT, PRINT, NULINE, SAVE, and RESTOR. Figure 6.12 shows the subroutine nesting structure of the program. This structure, which at first sight seems quite complex, can be easily implemented in the PDP-11, using the scheme described in Section 6.6.

Figure 6.13 shows “snapshots” of the system stack (bottom on the left and top on the right) as it appears in the course of executing HANOI

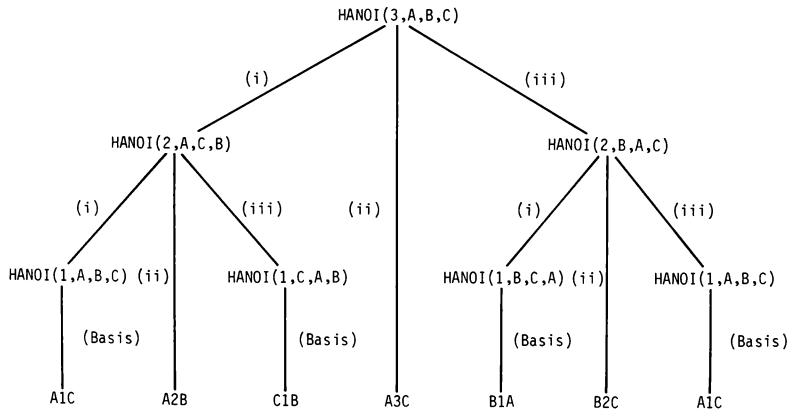


Figure 6.10 Execution of HANOI(3,A,B,C).

```

        .TITLE HANOI
; SOLVES TOWER OF HANOI PUZZLE. PRINTS OUT SEQUENCE OF MOVES OF N<7 DISCS
; FROM INITIAL SPINDLE A TO FINAL SPINDLE C, USING SPINDLE B FOR TEMPORARY STORAGE
LC=
.=4+LC
        .WORD 6,0,12,0 ;INITIALIZE ERROR VECTORS
.=500+LC ;ALLOW FOR STACK SPACE
START: MOV PC,SP
        TST -(SP) ;INITIALIZE SP TO START
;
KBSTAT=177560
KBDATA=177562
PRSTAT=177564
PRDATA=177566
;
; MAIN PROGRAM
        JSR PC,INPUT ;READ N INTO R5 (ASCII FORM) AND ECHO
        MOV R5,R0
        BIC #177770,R0 ;(R0)=N
        JSR PC,NULINE
        JSR PC,NULINE ;LEAVE DOUBLE SPACE
        TST R0
        BEQ OUT ;IF N=0, HALT
        MOV #'A,R1 ;X='A
        MOV #'B,R2 ;Y='B
        MOV #'C,R3 ;Z='C
        JSR PC,HANOI ;CALL HANOI(N,'A','B','C')
OUT: (a1) HALT
;
; HANOI(N,X,Y,Z)
; SOLVES TOWER OF HANOI PUZZLE. PRINTS OUT SEQUENCE OF MOVES OF N DISCS FROM
; INITIAL SPINDLE X TO FINAL SPINDLE Z, USING SPINDLE Y FOR TEMPORARY STORAGE.
; ARGUMENTS: (R0)=N, (R1)=X, (R2)=Y, (R3)=Z.
; USES R4, R5.
HANOI: CMP R0,#1
        BEQ BASIS ;IF N=1, EXECUTE BASIS
        JSR PC,SAVE ;SAVE N,X,Y,Z
        DEC R0 ;(R0)=N-1
        MOV R2,R4
        MOV R3,R2 ;(R2)=Z
        MOV R4,R3 ;(R3)=Y
        JSR PC,HANOI ;CALL HANOI(N-1,X,Z,Y)
        JSR PC,RESTOR ;RESTORE N,X,Y,Z
        JSR PC,OUTPUT ;PRINT XNZ
        JSR PC,SAVE ;SAVE N,X,Y,Z
        DEC R0 ;(R0)=N-1
        MOV R1,R4
        MOV R2,R1 ;(R1)=Y
        MOV R4,R2 ;(R2)=X
        JSR PC,HANOI ;CALL HANOI(N-1,Y,X,Z)
        JSR PC,RESTOR ;RESTORE N,X,Y,Z
        RTS PC ;EXIT
BASIS: JSR PC,OUTPUT ;PRINT X1Z
        RTS PC ;EXIT
;

```

Figure 6.11 Tower of Hanoi program.

```

;          OUTPUT
; PRINTS XNZ (NEXT DISC MOVEMENT). REGISTERS UNCHANGED.
OUTPUT: MOV  R1,R5
        JSR  PC,PRINT      ;PRINT X
        MOV  R0,R5
        ADD  #60,R5        ;CONVERT N TO ASCII
        JSR  PC,PRINT      ;PRINT N
        MOV  R3,R5
        JSR  PC,PRINT      ;PRINT Z
        JSR  PC,NULINE     ;SKIP TO NEXT LINE
        RTS  PC            ;EXIT

;
;          SAVE
; PUSHES (R0),(R1),(R2),(R3) ONTO SYSTEM STACK. REGISTERS UNCHANGED.
SAVE:   MOV  (SP)+,R4      ;SAVE RETURN ADDRESS
        MOV  R0,-(SP)     ;PUSH (R0)
        MOV  R1,-(SP)     ;PUSH (R1)
        MOV  R2,-(SP)     ;PUSH (R2)
        MOV  R3,-(SP)     ;PUSH (R3)
        MOV  R4,-(SP)     ;RESTORE RETURN ADDRESS
        RTS  PC            ;EXIT

;
;          RESTOR
; POPS (R3),(R2),(R1),(R0) FROM SYSTEM STACK. R4 AND R5 UNCHANGED.
RESTOR: MOV  (SP)+,R4      ;SAVE RETURN ADDRESS
        MOV  (SP)+,R3     ;POP (R3)
        MOV  (SP)+,R2     ;POP (R2)
        MOV  (SP)+,R1     ;POP (R1)
        MOV  (SP)+,R0     ;POP (R0)
        MOV  R4,-(SP)     ;RESTORE RETURN ADDRESS
        RTS  PC            ;EXIT

;
;          PRINT
; PRINTS CONTENTS OF R5. REGISTERS UNCHANGED.
PRINT:  TSTB PRSTAT      ;IS PRINTER READY?
        BPL  PRINT        ;IF NOT, WAIT
        MOV  R5,PRDATA   ;IF SO, PRINT (R5)
        RTS  PC            ;EXIT

;
;          NULINE
; SKIPS TO NEW LINE. USES R5.
NULINE: MOV  #12,R5
        JSR  PC,PRINT      ;PRINT LF
        MOV  #15,R5
        JSR  PC,PRINT      ;PRINT CR
        RTS  PC            ;EXIT

;
;          INPUT
; STORES TYPED-IN CHARACTER IN R5 AND ECHOES IT. OTHER REGISTERS UNCHANGED.
INPUT:  TSTB KBSTAT      ;IS CHARACTER IN?
        BPL  INPUT        ;IF NOT, WAIT
        MOV  KBDATA,R5   ;(R5)=CHARACTER
        JSR  PC,PRINT      ;PRINT CHARACTER
        RTS  PC            ;EXIT

;
        .END  START

```

Figure 6.11 (cont.)

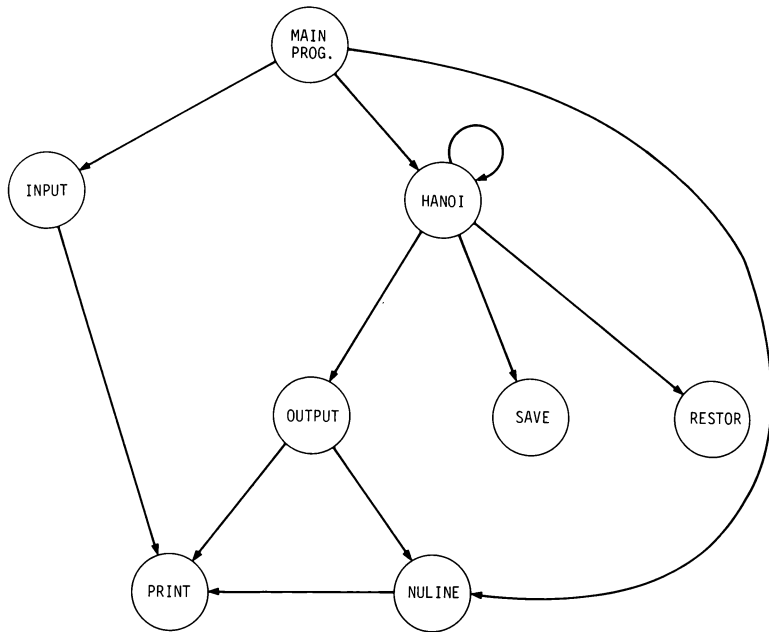


Figure 6.12 Subroutine nesting structure of Tower of Hanoi program.

```

α1
α1 3 A B C α2
α1 3 A B C α2 2 A C B α2
α1 3 A B C α2 2 A C B α2 1 A B C α2      (Print A1C)
α1 3 A B C α2 2 A C B α2                  (Print A2B)
α1 3 A B C α2 2 A C B α2 1 C A B α3      (Print C1B)
α1 3 A B C α2 2 A C B α2
α1 3 A B C α2                              (Print A3C)
α1 3 A B C α2 2 B A C α3
α1 3 A B C α2 2 B A C α3 1 B C A α2      (Print B1A)
α1 3 A B C α2 2 B A C α3                  (Print B2C)
α1 3 A B C α2                              (Print A1C)
α1
  
```

Figure 6.13 Stack snapshots for HANOI(3,A,B,C).

(3,A,B,C). (The perturbations of the stack due to OUTPUT are not shown.)
 (Compare Figure 6.13 with Figure 6.10!)

Note that the SAVE and RESTOR subroutines pop the return address from the system stack as they are entered and push it back onto the system stack just before they exit. These operations are necessary because the “saving” and “restoring” activities of SAVE and RESTOR might otherwise remove the return address from the top of the stack.

6.9 COROUTINES

It is sometimes the case that two subprograms alternately call each other in such a way that, each time, the jump is not made to the beginning of the other subprogram but to the point at which the other subprogram was last interrupted (see Figure 6.14). In this case (where each subprogram plays

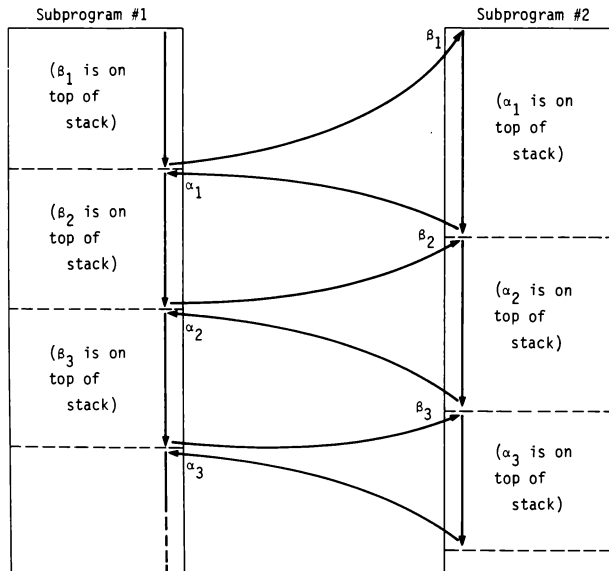


Figure 6.14 Coroutines.

the role of a calling program as well as a subroutine) the subprograms are referred to as *coroutines*.

Coroutines linkages can be implemented quite easily in the PDP-11. Suppose that β_1 is on top of the system stack when subprogram #1 is started. When #1 reaches $\alpha_1 - 2$ and is ready to transfer control to #2 (at β_1), it issues

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

which does the following:

1. Pops the top of the system stack (i.e., β_1) into temporary location.
2. Pushes PC (i.e., α_1) onto the system stack.
3. Copies the contents of the temporary location into PC.

Thus, α_1 is now on top of the system stack and #2 starts executing at β_1 . When #2 reaches $\beta_2 - 2$ and is ready to transfer control to #1 (at α_1), it issues the same JSR PC,@(SP)+ instruction. The return address β_2 will now replace α_1 at the top of the stack and execution will start at α_1 . And so forth.

EXERCISES

6.1 What does the following program segment compute?

```

          CLR   R0
          JSR   PC,SUB
          HALT
SUB:      DEC   R2
          BMI   EXIT
          ADD   R1,R0
          JSR   PC,SUB
EXIT:    RTS   PC

```

6.2 Trace the execution of the following program, indicating after each instruction's execution the octal contents of R3, R4, SP, and the system stack.

```

LC=.
.=500+LC
START:  MOV   PC,SP
          TST  -(SP)
          MOV  #123,R3

```



```

                JSR   R3,GAME
                .WORD 5
                HALT
GAME:          MOV   (R3)+,R4
                DEC   R4
                BEQ   EXIT
                JSR   R3,GAME
                .WORD 1
EXIT:          RTS   R3
                .END

```

6.3 Consider the following program :

```

LC=.
.=500+LC
START:  MOV   #4567,R0
        JSR   PC,BTOA
        HALT
;
;          SUBROUTINE BTOA
; BTOA CONVERTS THE OCTAL CONTENTS OF R0 INTO ASCII AND PRINTS IT OUT.
BTOA:   CMP   #7,R0          ;N LESS THAN OR EQUAL TO 7?
        BHIS B1             ;IF SO, CONVERT AND PRINT
        MOV   RO,-(SP)       ;ELSE, SAVE N
        BIC   #177770,(SP)   ;REMOVE ALL BUT LAST 3 BITS FROM SAVED N
        CLC                   ;PREPARE TO SHIFT N RIGHT
        ROR   RO             ;SHIFT N
        ASR   RO             ;RIGHT
        ASR   RO             ;THREE BITS
        JSR   PC,BTOA        ;CONVERT WHAT'S LEFT
        MOV   (SP)+,R0       ;GET BACK SAVED N
B1:     ADD   #'0,R0         ;CONVERT TO ASCII
LOOP:   TSTB 177564         ;IS PRINTER READY?
        BPL  LOOP           ;IF NOT, IDLE
        MOV  RO,177566     ;ELSE, PRINT CHARACTER
        RTS   PC           ;AND RETURN
        .END  START

```

Show the contents of the system stack when it is at its fullest.

6.4 Consider the following program :

```

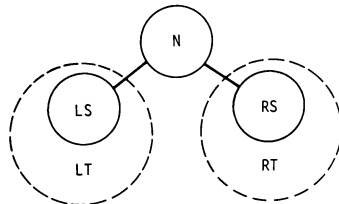
LC=.
.=500+LC
START:  MOV   PC,SP
        TST  -(SP)
        MOV  SIZE,R2
        MOV  #ARRAY,R1
        MOV  (R1)+,RO
        JSR  PC,COMPR
        HALT
SIZE:   .WORD 6
ARRAY:  .WORD 137,10,205,647,53,122
;
;          SUBROUTINE COMPR
COMPR:  DEC   R2
        BEQ  RETURN
        CMP  (R1)+,RO
        BLT  REPEAT
        MOV  -2(R1),RO
REPEAT: JSR  PC,COMPR
RETURN: RTS  PC
        .END  START

```

- (a) What is the final octal contents of R0?
 (b) Show the octal contents of the system stack when it is at its fullest.
 (c) Describe briefly what the program does (with an arbitrary collection of numbers in ARRAY).
- 6.5 Write a subroutine DUMPREG that prints out the octal contents of all the general-purpose registers and of the PSR as they appear just before the subroutine is entered. Before DUMPREG is exited, the original contents of all the registers must be restored. DUMPREG may call other subroutines, such as CONVERT (for converting octal contents into ASCII), and PRINT (for printout). (You can use SUMPREG for debugging.)
- 6.6 Write a subroutine whose parameters are the first address ADDR of an N-byte array, the number N, and a character CHAR. The subroutine puts 1 in R0 if CHAR is found in any of the N bytes, and 0 otherwise. Write a main program (using the above subroutine) which accepts a string of characters from the teletype and prints out (in octal) the number of these characters found in the array.
- 6.7 Write a recursive subroutine which adds up all the 16-bit 2's-complement numbers stored in an N-word array starting at location X. Assume that X is stored in R0, N in R1, and the sum in R2. (This can be easily done without recursion, but try it recursively anyway.)
- 6.8 Write a recursive subroutine that finds a positive number N in R0 and leaves FACT(N) in R1.
- 6.9 Write a recursive subroutine that finds an integer $N \geq 1$ in R0 and leaves FIB(N) in R1.
- 6.10 A *binary tree* T with *root* N is a graph that can be defined recursively as follows:

Basis: If T is a single node N, then T is a binary tree with root N.

Induction step: (See the figure on the right.) If T consists of a node N connected to nodes LS (*left successor*) and RS (*right successor*), and if LS and RS are the roots of binary trees (denoted LT and RT, respectively), then T is a binary tree with root N.



As an example, Figure 6.15(a) shows a binary tree whose root is labeled 0 and whose nodes are labeled (in octal) 0,1,2, . . . , 14.

In the PDP-11, a k-node tree can be represented by a k-word array with base address TREE. Node number N is represented in location TREE+2N. The high byte of TREE+2N contains the number LS of the left successor of N, and the low byte the number RS of the right successor of N.

Figure 6.15(b) shows the representation of the tree of Figure 6.15(a).

(a) The *inorder traversal* of a binary tree T is defined recursively as follows:

Basis: If T is a single node N, visit N.

Induction step: If T consists of a node N connected to binary trees LT and RT, then: (a) traverse LT in inorder; (b) visit N; (c) traverse RT in inorder.

Write a recursive subroutine which prints out the node numbers of a given binary tree as they appear in inorder traversal. (For example, for the tree of Figure 6.15 the printout should be 3,1,4,0,7,5,13,11, 14,10,12,2,6.)

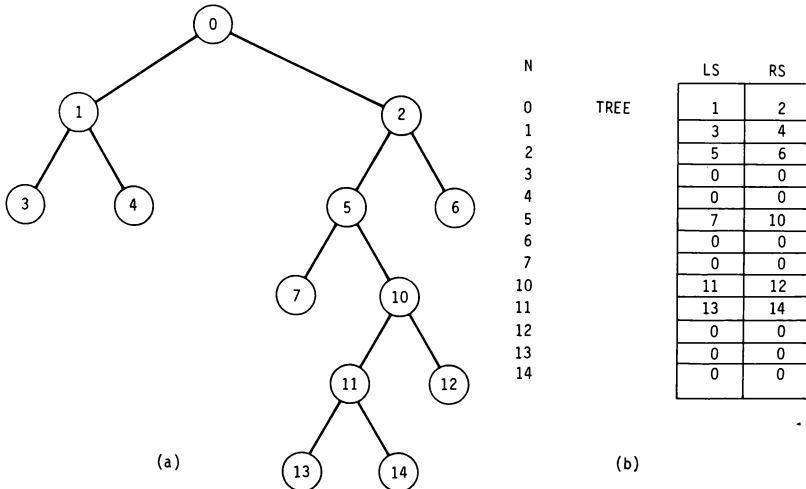


Figure 6.15 Binary tree and its representation.

(b) The *height* of a binary tree T , denoted $H(T)$, can be defined recursively as follows:

Basis: If T is a single node N , then $H(T) = 0$.

Induction step: If T consists of a node N connected to binary trees LT and RT , then

$$H(T) = 1 + \text{maximum}(H(LT), H(RT))$$

Write a recursive subroutine that prints out the height of a given binary tree. (For example, for the tree of Figure 6.15, the printout should be 5.)

ARITHMETIC OPERATIONS

7

In this chapter we describe in detail how the condition codes in the PSR are influenced by arithmetic operations. In particular, we examine how the C and V codes behave and how they can be used for overflow indication and in double-precision arithmetic. We also have a closer look at the TST and CMP instructions, at branch instructions, and at shift instructions. The chapter closes with an ASCII-to-binary conversion program which illustrates some of the points made previously.

7.1 CARRY AND OVERFLOW UNDER ADDITION

As already mentioned in Section 3.2, the C (*Carry*) and V (*oVerflow*) condition codes in the PSR are influenced by the result of an ADD instruction in the following manner:

$$C = \begin{cases} 1 & \text{if addition resulted in carry from MSB} \\ 0 & \text{otherwise} \end{cases}$$

$$V = \begin{cases} 1 & \text{if operands are of same sign and their sum is of the opposite sign} \\ 0 & \text{otherwise} \end{cases}$$

Examples

For simplicity, let us assume that the PDP-11 has 4-bit words, capable of accommodating integers from -8 to +7.

Example (Decimal)	4-bit 2's- Complement Addition	Final Value of		Binary Result Is
		C	V	
1 <u>+ (+2)</u>	0001 <u>+ 0010</u>	0	0	OK
3	0011			
5 <u>+ (+6)</u>	0101 <u>+ 0110</u>	0	1	wrong
11	1011			
-6 <u>+ (+7)</u>	1010 <u>+ 0111</u>	1	0	OK
1	↙0001 1			
-6 <u>+ (+3)</u>	1010 <u>+ 0011</u>	0	0	OK
-3	1101			
-5 <u>+ (-2)</u>	1011 <u>+ 1110</u>	1	0	OK
-7	↙1001 1			
-6 <u>+ (-6)</u>	1010 <u>+ 1010</u>	1	1	wrong
-12	↙0100 1			

□

Generally, the following can be readily deduced:

1. If both numbers are positive, C is always 0 (since both MSB's are 0).
The sum is incorrect only if it is negative ($V = 1$).
2. If both numbers are negative, C is always 1 (since both MSB's are 1).
The sum is incorrect only if it is positive ($V = 1$).

3. If numbers have opposite signs, V is always 0 and the sum is always correct. C can be either 0 or 1.

In conclusion, the result of adding two numbers is incorrect only if $V = 1$.

7.2 CARRY AND OVERFLOW UNDER SUBTRACTION

The way in which the C and V condition codes are influenced by the result of a subtraction operation (i.e., by SUB or CMP instruction) is the following:

$$C = \begin{cases} 1 & \text{if subtraction did not result in carry from MSB} \\ 0 & \text{otherwise} \end{cases}$$

$$V = \begin{cases} 1 & \text{if operands are of opposite signs and the result} \\ & \text{is of the same sign as the number subtracted*} \\ 0 & \text{otherwise} \end{cases}$$

Examples

Again, we shall use 4-bit words.

Example (Decimal)	4-bit 2's- Complement Subtraction	Final Value of		Binary Result Is
		C	V	
$\begin{array}{r} 6 \\ - (+3) \\ \hline 3 \end{array}$	$\begin{array}{r} 0110 \\ + 1101 \\ \hline \overset{\curvearrowright}{0}011 \\ 1 \end{array}$	0	0	OK
$\begin{array}{r} 4 \\ - (-2) \\ \hline 6 \end{array}$	$\begin{array}{r} 0100 \\ + 0010 \\ \hline 0110 \end{array}$	1	0	OK
$\begin{array}{r} 4 \\ - (-5) \\ \hline 9 \end{array}$	$\begin{array}{r} 0100 \\ + 0101 \\ \hline 1001 \end{array}$	1	1	wrong

*In SUB the number subtracted is the *source* operand; in CMP it is the *destination* operand.

Example (Decimal)	4-bit 2's- Complement Subtraction	Final Value of		Binary Result Is
		C	V	
5 <u>- (+7)</u> -2	0101 <u>+ 1001</u> 1110	1	0	OK
-3 <u>- (+2)</u> -5	1101 <u>+ 1110</u> ↙1011 1	0	0	OK
-2 <u>- (-5)</u> 3	1110 <u>+ 0101</u> ↙0011 1	0	0	OK
-4 <u>- (+6)</u> -10	1100 <u>+ 1010</u> ↙0110 1	0	1	wrong

□

Inspecting these examples we can conclude (as we did with addition) that the result of subtraction is incorrect only if $V = 1$.

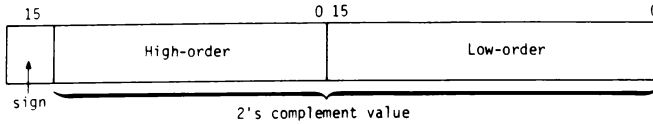
The negation operation (NEG), when applied to a nonzero number, results in $C = 1$; when it is applied to 0, it results in $C = 0$. Notice that the negation of a number (which consists of 1's-complementing it and then adding 1) produces a carry only when the number is 0. Thus, a negation operation results in $C = 1$ if there is *no* carry, and in $C = 0$ if there *is* carry.

7.3

DOUBLE-PRECISION ARITHMETIC

In many applications the range of numbers accommodated by a single computer word is not sufficient, and two consecutive words are employed to store a number. In the PDP-11 a two-word, or *double-precision* number is stored in a 32-bit 2's-complement form. The *low-order* word holds the least significant bits of the number, while the *high-order* word holds the

most significant bits of the number. The MSB of the high-order word is the sign bit of the entire number:



With double precision, the PDP-11 can handle numbers up to $2,147,483,647_{10}$, as compared to $32,767_{10}$ with single precision!

How do we add two double-precision numbers, say α and β ? Suppose that α is stored in AL and AH, β in BL and BH, and $\alpha + \beta$ is to be stored in BL and BH (where the suffix L denotes low-order words and H high-order words). To compute $\alpha + \beta$ it is not sufficient to issue ADD AL,BL followed by ADD AH,BH, since this will ignore a possible carry from the low-order words. This carry, which is the value of C after the low-order addition is executed, must be added to AH (or to BH) just before the high-order addition is performed. This operation is facilitated by the instruction ADC d, which performs $(d) \leftarrow (d) + C$. Thus, to compute $\alpha + \beta$ we issue

```
ADD AL,BL
ADC BH
ADD AH,BH
```

Examples

For the remainder of this section, all examples will assume 4-bit words.

		Example 1		Example 2
Initially	{ AH AL	0001 1001	$\leftarrow \alpha \rightarrow$	0001 1001
	{ BH BL	0011 0101	$\leftarrow \beta \rightarrow$	0011 0111
After	{ AH AL	0001 1001		0001 1001
ADD AL,BL	{ BH BL	0011 1110 (C = 0)		0011 0000 (C = 1)
After	{ AH AL	0001 1001		0010 1001
ADC AH	{ BH BL	0011 1110		0011 0000
After	{ AH AL	0001 1001		0010 1001
ADD AH,BH	{ BH BL	0100 1110	$\leftarrow \alpha + \beta \rightarrow$	0101 0000

□

To negate a double-precision number α stored in AL and AH, we 1's-complement AH, 1's-complement AL, add 1 to AL, and add the resulting carry to AH. The 1's complementation of AH can be done by negating it and subtracting 1; the 1's complementation and incrementation by 1 of AL can be done simply by negating it. If the negation of AL does not produce a carry (and hence results in $C = 1$), the negation of α is complete. If the negation of AL produces a carry (and hence results in $C = 0$), then the addition of this carry to AH can be effected by neglecting to subtract 1 from AH after it is first negated. In conclusion, α can be negated by negating AH, negating AL, and finally subtracting C from AH. The latter operation is facilitated by the SBC d instruction, which performs $(d) \leftarrow (d) - C$. Thus, to compute $-\alpha$, we issue

```
NEG AH
NEG AL
SBC AH
```

Examples

		Example 1		Example 2
(Initially)	AH AL	0101 1011	$\leftarrow \alpha \rightarrow$	0101 0000
(After NEG AH)	AH AL	1011 1011		1011 0000
(After NEG AL)	AH AL	1011 0101	($C = 1$)	1011 0000 ($C = 0$)
(After SBC AH)	AH AL	1010 0101	$\leftarrow -\alpha \rightarrow$	1011 0000

Suppose that α and β are stored as before and we wish to compute $\alpha - \beta$ and store it in AL and AH. We can do this by first negating β and then adding the result to α (using the double-precision negation and addition proposed above). These operations are equivalent to subtracting BL from AL, adding the carry to AH, subtracting BH from AH, and subtracting 1 from the result (since the second subtraction is tantamount to adding to AH the negative, rather than the 1's complement, or BH). Adding the carry to AH and then subtracting 1 from the final result can be combined (as was done with double-precision negation) by subtracting from AH the value of C produced by the low-order subtraction. Thus, to compute $\alpha - \beta$, we issue

← α →
 SBC AH
 SUB BH,AH

Examples

		Example 1		Example 2
Initially	{ AH AL BH BL	0101 1001	← α →	0101 0111
		0011 0101	← β →	0011 1010
After	{ AH AL SUB BL,AL	0101 0100 (C = 0)		0101 1101 (C = 1)
	{ BH BL	0011 0101		0011 1010
After	{ AH AL SBC AH	0101 0100		0100 1101
	{ BH BL	0011 0101		0011 1010
After	{ AH AL SUB BH,AH	0010 0100	← $\alpha - \beta$ →	0001 1101
	{ BH BL	0011 0101		0011 1010

□

From Sections 7.1 and 7.2 we know that the result of addition or subtraction is incorrect only if it produces overflow (i.e., $V = 1$). In the double-precision case this means that the result of the *high-order* addition or subtraction produces $V = 1$. Overflow produced by the low-order operation is irrelevant.

7.4 THE TST AND CMP INSTRUCTIONS

The instruction `CMP s,d` (or `CMPB s,d`) forms the difference $(s) - (d)$ inside the ALU (not in `d!`). It is most often used to set the Z, N, C, and V condition codes so as to facilitate the comparison of (s) and (d) . It is important to note that the C and V codes resulting from `CMP` are set according to the rules of *subtraction* (stated in Section 7.2). We shall elaborate on the applications of `CMP` in Section 7.5.

The instruction `TST d` (or `TSTB d`) executes $(d) \leftarrow (d)$. Its most common purpose is to set the Z and N condition codes so as to facilitate the determination whether (d) is zero or nonzero, positive or negative.

Examples

Contents of CM Word Addressed	Results of Instruction					
	TST 1000		TSTB 1000		TSTB 1001	
	Z	N	Z	N	Z	N
000000	1	0	1	0	1	0
000001	0	0	0	0	1	0
000400	0	0	1	0	0	0
000401	0	0	0	0	0	0
000200	0	0	0	1	1	0
100000	0	1	1	0	0	1
100001	0	1	0	0	0	1
177500	0	1	0	0	0	1
100200	0	1	0	1	0	1
000600	0	0	0	1	0	0

□

Although CMP and TST are most often used for comparing and testing numbers, they are sometimes used as a “clever” way for incrementation or decrementation of registers. For example:

```

TST -(SP)           ;(SP)-(SP)-2
TST (R5)+           ;(R5)-(R5)+2
CMP (R0)+,(R0)+    ;(R0)-(R0)+4

```

Any time that a TST or CMP instruction is not followed by a conditional branch instruction, it is most likely being used for autoincrementing or autodecrementing a register.

Note that TST (R5)+ is illegal when R5 holds an odd address (since TST is a word instruction). Thus, great caution should be exercised by the programmer in using TST and CMP for incrementation and decrementation.

While TST (or TSTB) is intended for testing entire words (or bytes), the BIT (or BITB) instruction can be used to test selected portions of words (or bytes). Specifically the instruction BIT d (or BITB d) forms the AND product of (s) and (d) inside the ALU (not in d!), enabling us to test only those bits in d which correspond to 1-bits in s. For example, to branch to L if either bit 7 or bit 15 of R0 (or both) are 1, we can issue

```

BIT #100200,R0
BNE L

```

In Section 4.8 we listed 16 conditional branch instructions. The first eight,

BEQ, BNE, BPL, BMI, BVC, BVS, BCC, BCS

are best suited to test whether the result of an arithmetic operation is zero or nonzero, positive or negative, did or did not cause overflow, and did or did not produce a carry.

To compare two numbers, say α in A and β in B, it is most convenient to issue

CMP A,B

(which forms $\alpha - \beta$) and then any of the following instructions:

	Mnemonic	Branch to q if*	
Signed conditional branches	BGE q	$N \vee V = 0$	$(\alpha \geq \beta)$
	BLT q	$N \vee V = 1$	$(\alpha < \beta)$
	BGT q	$Z \vee (N \vee V) = 0$	$(\alpha > \beta)$
	BLE q	$Z \vee (N \vee V) = 1$	$(\alpha \leq \beta)$
Unsigned conditional branches	BHI q	$C \vee Z = 0$	$(\alpha > \beta)$
	BLOS q	$C \vee Z = 1$	$(\alpha \leq \beta)$
	BHIS q	$C = 0$	$(\alpha \geq \beta)$
	BLO q	$C = 1$	$(\alpha < \beta)$

The “signed” conditional branches regard α and β as 16-bit 2’s-complement numbers and compare them accordingly. The “unsigned” conditional branches regard α and β as 16-bit positive numbers (with the MSB no longer regarded as a sign bit, but as the coefficient of 2^{15} in the binary representation), and compare them accordingly.

As a general rule, BGE, BLT, BGT, BLE, BEQ, and BNE should be used to compare program data. BHI, BLOS, BHIS, BLO, BEQ, and BNE should be used to compare CM addresses.

For example, consider the following code which clears a section of CM starting at A up to and including B (where B is a higher address):

*The rules for the \vee operation are: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 1 = 0$. The rules of the \wedge operation are: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 1 = 1$.

```

A=1000
B=2000
      MOV #A,R0
      MOV #B,R1
LOOP: CLR (R0)+
      CMP R1,R0
      BGE LOOP

```

BGE treats the operands as 2's-complement numbers. The loop will terminate when $(R0) = 2002$, which is the correct action. Suppose, however, that the first two assignments are

```

A = 070000
B = 170002

```

(which may be legal addresses in some PDP-11 models). After `CLR (R0)+` is first executed, $(R1) = 170002$ and $(R0) = 070002$. Hence $(R1) - (R0) = 100000$, which is a negative number, causing the loop to terminate prematurely.

The correct solution is to use a branch instruction that treats the operands as *unsigned* numbers. The following code is correct:

```

LOOP: CLR (R0)+
      CMP R1,R0
      BHIS LOOP

```

BHIS will work correctly regardless of the values of A and B.

Table 7.1 lists a number of examples (using 4-bit words) that illustrate how the signed and unsigned conditional branches are influenced by various combinations of condition codes. In each example the first row lists α , the second row lists β (preceded by a minus sign), and the third row lists $\alpha - \beta$ (the number formed by `CMP A,B`). The checkmarks indicate the conditional branch instructions which, when appearing after `CMP A,B`, cause a branch.

It is always advisable to use BGE and BLT rather than BPL and BMI when comparing two signed numbers. For example, suppose that we want the program to branch to L1 if $(A) < (B)$ and to L2 otherwise. We may write

```

      CMP A,B
      BMI L1
L2: ---

```

If A happens to contain 177772 (i.e., -6) and B happens to contain 077777, then `CMP A,B` forms $(A) - (B) = 077773$, with $Z = 0$, $N = 0$, $C = 0$, $V = 1$.

TABLE 7.1 Branch Examples

Example	Resultant Condition Codes										Signed Conditionals				Unsigned Conditionals									
	Decimal	Binary	Z	N	C	V	N	V	Z	V	(N	V)	C	V	Z	BGE	BLT	BGT	BLE	BHI	BLOS	BHIS	BLO	
7	0111																							
-(+3)	-0011																							
4	0100																							
5	0101																							
-(+5)	-0101																							
0	0000																							
3	0011																							
-(+6)	-0110																							
-3	1101																							
-2	1110																							
-(+2)	-0010																							
-4	1100																							
-4	1100																							
-(+5)	-0101																							
-9	0111																							
-3	1101																							
-(+2)	-1110																							
-1	1111																							
-6	1010																							
-(+6)	-1010																							
0	0000																							
-2	1110																							
-(+5)	-1011																							
3	0011																							

The program will thus branch to L2, although $(A) < (B)$. However, if we write

```

        CMP A,B
        BLT L1
L2: ---

```

the program will branch to L1. (What happens when `CMP A,B` is followed by `BL0 L1`?)

Unsigned conditional branches are often used in the comparison of double-precision numbers. For example, suppose that α and β are positive double-precision numbers stored as described in Section 7.3. Then $\alpha > \beta$ if $(AH) > (BH)$, or if: (1) $(AH) = (BH)$, and (2) $(AL) > (BL)$, where (AL) and (BL) are unsigned! Thus, to jump to X if $\alpha > \beta$ and to Y otherwise, we write

```

        CMP AH,BH
        BGT X
        BLT Y
        CMP AL,BL
        BHI X
Y:     ---
      .
      .
      .
X:     ---

```

Another use of the unsigned branch instructions is in testing for two conditions simultaneously. Suppose that we want to know if $(R0)$ is less than 0 or greater than 7 and if so, to branch to location X. We can write

```

        CMP R0,#7
        BGT X
        TST R0
        BLT X

```

An alternative solution is

```

        CMP R0,#7
        BHI X

```

This works because a negative number, when treated as an unsigned number, is always “higher” than any positive number. It is worth mentioning that another solution would be

```

        BIT #177770,R0
        BNE X

```


Almost every single-operand and double-operand instruction influences the Z and N condition codes and many of these instructions influence the C and V codes as well.* This means that a compare or a test instruction which one may be tempted to insert after an operation and before a branch instruction are actually redundant. For example, in

```
DEC   R4
CMP   R4,#0
BNE   LOOP
```

the CMP instruction is superfluous; in

```
SUB   A,B
TST   B
BEQ   NEXT
```

the TST instruction is superfluous.

7.6 SHIFT INSTRUCTIONS

The PDP-11 has a number of instructions that shift the contents of a word or a byte 1 bit to the left or 1 bit to the right (see Figure 7.1):

```
ROL,  ROLB    (rotate left)
ROR,  RORB    (rotate right)
ASL,  ASLB    (arithmetic shift left)
ASR,  ASRB    (arithmetic shift right)
```

To understand the operation of ROL and ROR (or ROLB and RORB), we take the destination word (or byte) and form a “ring” by joining its ends via a (fictitious) 1-bit register which holds the condition code C. The effect of ROL is to rotate the contents of this ring 1 bit clockwise; the effect of ROR is to rotate the contents counterclockwise.

The effect of ASL (or ASLB) is to shift the contents of the destination word (or byte) 1 bit left, with the rightmost bit replaced by 0 and the leftmost bit replacing the old value of C. This operation is equivalent to multiplying the contents of the destination (regarded as a 2’s-complement

*For details, see the “PDP-11 Processor Handbook.”

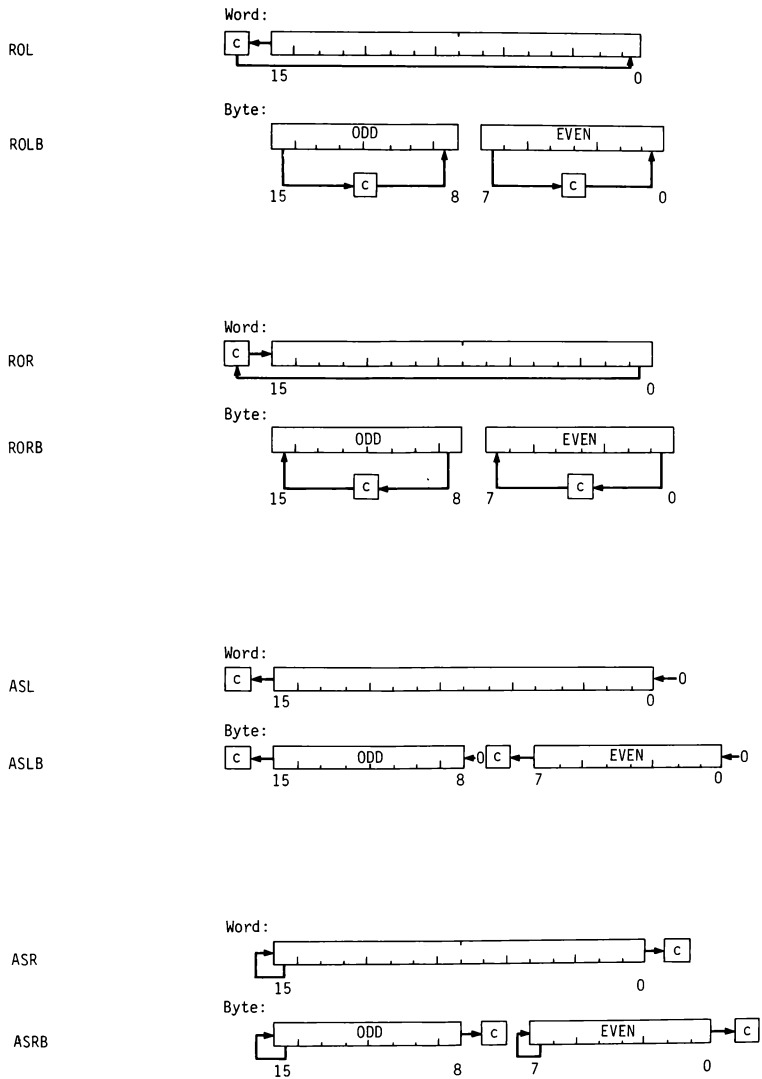


Figure 7.1 Shift instructions.

signed number) by 2. The V code is set to 1 if the result of this multiplication is out of bounds.

The effect of ASR (or ASRB) is to shift the contents of the destination word (or byte) 1 bit right, with the leftmost bit remaining unchanged and the rightmost bit replacing the old value of C. This operation is equivalent to dividing the contents of the destination by 2 (and subtracting $1/2$ if the contents is odd). An exception: -1 remains -1.

Examples

(A)	After ASL A	After ASR A
000032	000064 (= 32×2)	000015 (= $32/2$)
177746 (= -32)	177714 (= -64)	177763 (= -15)

□

7.7

EXAMPLE: ASCII-TO-BINARY CONVERSION

To illustrate the use of some of the instructions and features introduced in the preceding sections, we shall present a program which accepts a decimal number N from the teletype and converts it into a 16-bit 2's-complement form. N may be prefixed with + or - and is always followed by a carriage return. It should not exceed 32767_{10} in magnitude.

The program (flowcharted in Figure 7.2 and listed in Figure 7.3) stores the input characters in a byte array whose base address is STRING. The binary equivalent of N is left in R2. (If N is out of bounds, R2 is left with 100000_8 .) The program uses the following subroutines:

INPUT (stores the input characters in array STRING; uses PRINT)
 PRINT (prints out contents of R5)
 ATOB (computes the binary equivalent of N and puts it in R2;
 uses MUL)
 MUL [computes $(R0) \times (R1)$ and puts the result in R2]

The algorithm used by the multiplying subroutines MUL is much more efficient than that used by MULT in Section 6.5. It proceeds as follows:

1. Set (R2) to 0.
2. If bit 0 of (R1) is 1, add (R0) to (R2). Otherwise, skip this step.

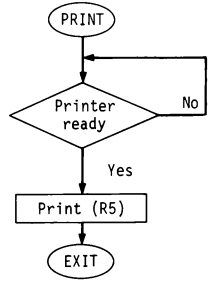
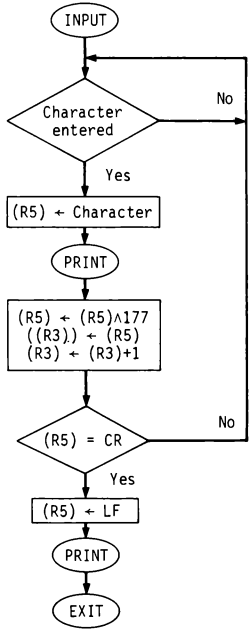
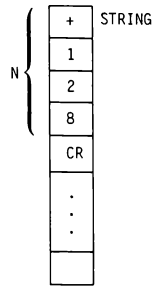
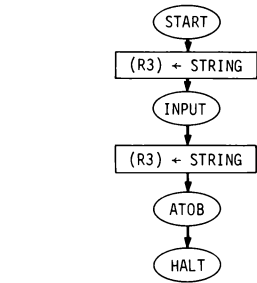


Figure 7.2 Flowcharts for ASCII-to-binary program.

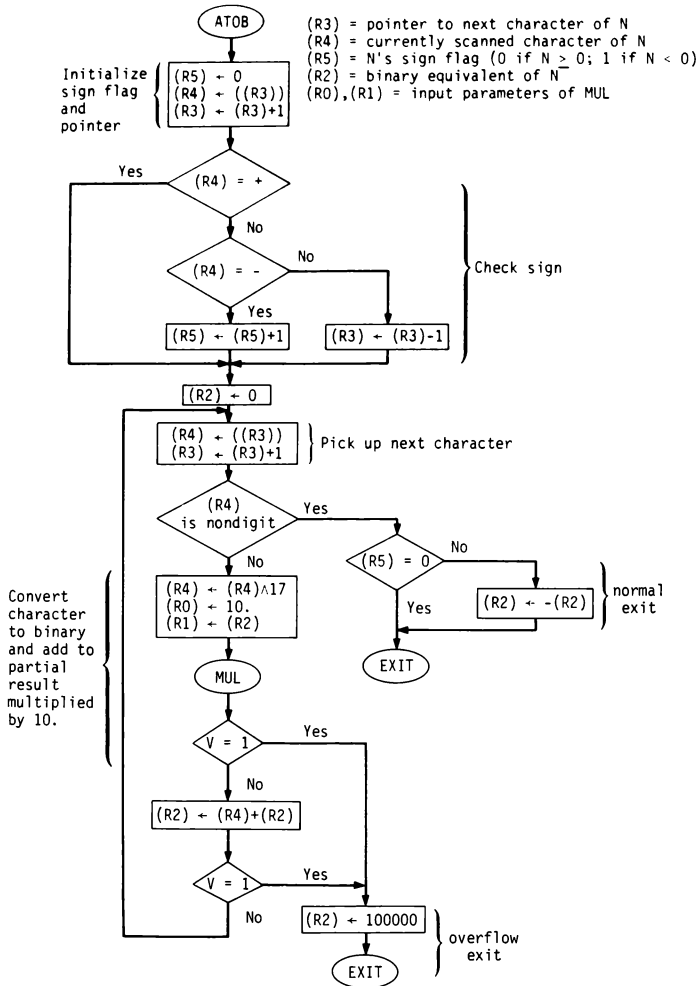


Figure 7.2 (cont.)

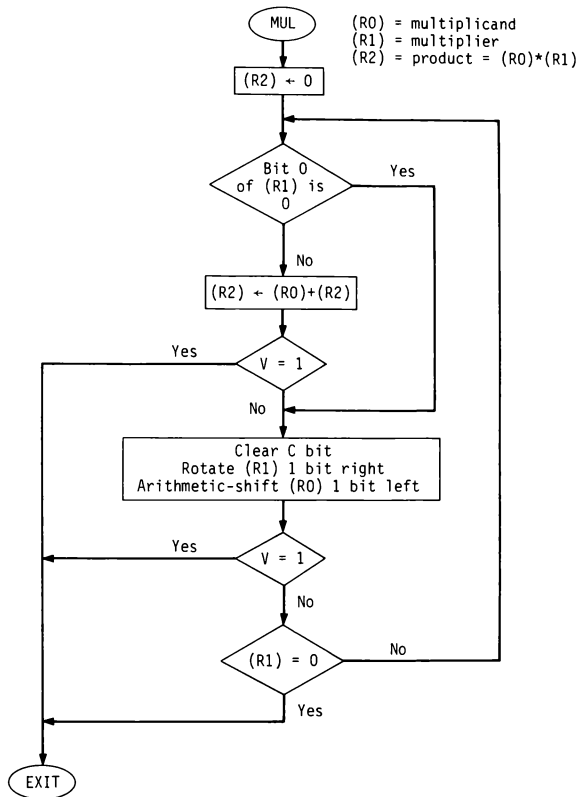


Figure 7.2 (cont.)

3. Shift (R0) left one bit and (R1) right one bit.
4. If (R1) = 0, R2 contains (R0)*(R1); exit. Otherwise, return to step 2.

That this algorithm works for positive numbers is clear from the following example (where all numbers are binary):

```

0 ... 00011010 ← Initial (R0)
0 ... 01001011 ← Initial (R1)
      11010
      11010
      11010
11010
      } shifted (R0)
11110011110 ← Final (R2) [sum of shifted (R0)'s]

```

To assure that the algorithm works for negative numbers also, the left shifting of (R0) must be done arithmetically (with ASL rather than ROL) and the right shifting of (R1) must be done circularly (with ROR rather than ASR), with C cleared before rotation.

If the product (R0)*(R1) is out of bounds, the addition in step 2 or the left shifting in step 3 will produce $V = 1$, which can be used by the calling program as an error indicator.

The subroutine ATOB starts by setting a “flag” (stored in R5) to 0 if N is positive and to 1 otherwise. It then converts $|N|$ to binary by the following algorithm:

1. Set (R2) to 0.
2. Pick up the next character. If it is a nondigit, R2 contains the binary equivalent of $|N|$; if (R5) = 1, negate (R2) and exit. If the character is a digit:
3. Multiply (R2) by 10_{10} (or 12_8) and add to it the binary equivalent of the digit picked up in step 2. Return to step 2.

For example, if $N = 2591$, R2 will successively contain, in decimal:

```

0
(0×10)+2 = 2
(2×10)+5 = 25
(25×10)+9 = 259
(259×10)+1 = 2591

```

or, in octal:

```

0
(0×12)+2 = 2
(2×12)+5 = 31
(31×12)+11 = 403
(403×12)+1 = 5037

```

```

        .TITLE  ASCTOBIN
; CONVERTS A TYPED-IN DECIMAL NUMBER N INTO ITS BINARY EQUIVALENT.
; N MAY BE PREFIXED WITH + OR - AND MUST BE FOLLOWED BY A CARRIAGE RETURN.
; THE BINARY EQUIVALENT OF N IS LEFT IN R2. IF N'S MAGNITUDE EXCEEDS
; 32767 DECIMAL, R2 WILL BE LEFT WITH 100000 OCTAL.
LC=
.=4+LC
        .WORD  6,0,12,0      ;INITIALIZE ERROR VECTORS
        .=500+LC           ;ALLOW FOR STACK SPACE
START:  MOV    PC,SP
        TST   -(SP)         ;INITIALIZE SP TO START
;
KBSTAT=177560
KBDATA=177562
PRSTAT=177564
PRDATA=177566
LF=12
CR=15
;
;
;           MAIN PROGRAM
        MOV   #STRING,R3    ;(R3)=STRING
        JSR  PC,INPUT       ;STORE INPUT STRING IN ARRAY
        MOV  #STRING,R3     ;(R3)=STRING
        JSR  PC,ATOB        ;CONVERT STRING INTO BINARY
        HALT
;
STRING: .BLKB 20.          ;STORAGE FOR TYPED-IN STRING
;
;           INPUT
; ECHOES TYPED-IN CHARACTERS AND STORES THEM IN BYTE ARRAY WHOSE BASE
; ADDRESS IS IN R3. EXITS AFTER CR IS TYPED. CHANGES R3, R5.
INPUT:  TSTB  KBSTAT        ;IS CHARACTER IN?
        BPL  INPUT         ;IF NOT, WAIT
        MOV  KBDATA,R5     ;(R5)=CHARACTER
        JSR  PC,PRINT      ;PRINT CHARACTER
        BIC  #177600,R5    ;REMOVE CHECK BIT
        MOV  R5,(R3)+      ;STORE CHAR. IN ARRAY, UPDATE INDEX
        CMP  #CR,R5       ;IS CHARACTER CR?
        BNE  INPUT        ;IF NOT, ACCEPT NEXT CHARACTER
        MOV  #LF,R5       ;ELSE,
        JSR  PC,PRINT      ; MOVE TO NEXT LINE
        RTS  PC           ;EXIT
;
;           PRINT
; PRINTS CONTENTS OF R5. REGISTERS UNCHANGED.
PRINT:  TSTB  PRSTAT       ;IS PRINTER READY?
        BPL  PRINT        ;IF NOT, WAIT
        MOV  R5,PRDATA    ;IF SO, PRINT (R5)
        RTS  PC           ;EXIT
;
;           ATOB
; CONVERTS INTO BINARY A DECIMAL NUMBER N STORED IN ASCII IN BYTE ARRAY
; WHOSE BASE ADDRESS IS IN R3. N MAY BE PREFIXED WITH + OR - AND MUST BE
; FOLLOWED BY A NON-DIGIT. THE BINARY EQUIVALENT OF N IS LEFT IN R2. IF
; N'S MAGNITUDE EXCEEDS 32767 DECIMAL, R2 IS LEFT WITH 100000 OCTAL.
; REGISTER ALLOCATION:
; (R0), (R1) ARE USED FOR INPUTTING MUL PARAMETERS
; (R2) = CONVERTED NUMBER
; (R3) = POINTER TO NEXT CHARACTER
; (R4) = SCANNED CHARACTER
; (R5) = SIGN FLAG (0 IF N IS POSITIVE, 1 OTHERWISE)
ATOB:  CLR   R5            ;ZERO SIGN FLAG (ASSUME N POSITIVE)

```

Figure 7.3 ASCII-to-binary program.


```

MOV B (R3)+,R4 ;(R4)=SCANNED CHARAC. , UPDATE INDEX
CMPB #'+',R4 ;IS CHARACTER '+'?
BEQ ATOB2 ;IF SO, START CONVERTING
CMPB #'-',R4 ;IS CHARACTER '-'?
BNE ATOB1 ;IF NOT, N IS UNSIGNED
INC R5 ;IF SO, SET SIGN FLAG FOR NEGATIVE N
BR ATOB2 ;START CONVERTING
ATO B1: DEC R3 ;CHARACTER IS DIGIT. BACKTRACK
ATO B2: CLR R2 ;INITIALIZE RESULT TO 0
ATO B3: MOV B (R3)+,R4 ;(R4)=SCANNED CHARACTER. UPDATE INDEX
CMPB #'0',R4 ;IF '0'-(R4) (NONDIGIT)
BHI ATOB4 ; PREPARE FOR EXIT
CMPB #'9',R4 ;IF '9'-(R4) (NONDIGIT)
BLO ATOB4 ; PREPARE FOR EXIT
BIC #177760,R4 ;CONVERT DIGIT TO BINARY
MOV #10.,R0 ;(R0)=10. (MUL PARAMETER)
MOV R2,R1 ;(R2)=(R1) (MUL PARAMETER)
JSR PC,MUL ;(R2)=(R0)*(R1)=10.*(R2)
BVS ATOB6 ;IF OVERFLOW, PREPARE FOR EXIT
ADD R4,R2 ;(R2)=(R4)+(R2)
BVS ATOB6 ;IF OVERFLOW, PREPARE FOR EXIT
BR ATOB3 ;SCAN NEXT CHARACTER
;NORMAL EXIT
ATO B4: TST R5 ;TEST SIGN FLAG
BEQ ATOB5 ;IF NUMBER IS POSITIVE, EXIT
NEG R2 ;ELSE, (R2)=- (R2)
ATO B5: RTS PC ;EXIT
;OVERFLOW EXIT
ATO B6: MOV #100000,R2 ;(R2)=100000
RTS PC ;EXIT
;
; MUL
; COMPUTES (R0)*(R1) AND STORES RESULT IN R2. IF RESULT'S MAGNITUDE
; EXCEEDS 32767 DECIMAL, V BIT IS SET TO 0. R3, R4, R5 NOT USED.
;
MUL: CLR R2 ;(R2)=0
MUL1: BIT #1,R1 ;TEST BIT 0 OF R1
BEQ MUL2 ;IF 0, DON'T ADD
ADD R0,R2 ;ELSE, (R0)=(R0)+(R2)
BVS MUL3 ;EXIT IF OVERFLOW
MUL2: CLC ;CLEAR C BIT
ROR R1 ;ROTATE R1 1 BIT RIGHT
ASL R0 ;ARITH.-SHIFT (R0) 1 BIT LEFT
BVS MUL3 ;EXIT IF OVERFLOW
TST R1 ;TEST (R1)
BNE MUL1 ;IF NOT 0, KEEP MULTIPLYING
MUL3: RTS PC ;EXIT
;
.END START

```

Figure 7.3 (cont.)

If the multiplication or the addition in step 3 produces an overflow, 100000₈ is stored in R2 as an error indicator.

Note that an ASCII digit can be converted to binary simply by picking up its rightmost 4 bits (or “ANDing” it with 17₈). For example, 5 in ASCII is 065, and 000065 \wedge 17 = 000005; 9 in ASCII is 071, and 000071 \wedge 17 = 000011.

- 7.1 What are the octal contents of R0 and the PSR after each instruction in the following program section is executed? (You may need the "PDP-11 Processor Handbook" for assistance.)

```

CLR      R0
DEC      R0
ADD      #77777,R0
ADD      #2,R0
COM      R0
SUB      #177777,R0
ADD      #100000,R0
SUB      #1,R0
BIC      #54321,R0
ASL      R0
ASR      R0
NEG      R0
ROR      R0

```

- 7.2 Consider the following program segment:

```

ADD      A1,B1
BVC      L1
BCC      L2
SUB      A2,B2
BCC      L3
BVC      L4
BR       L5

```

After the segment is executed, where does the program go to if the contents of A1, B1, A2, and B2 are as follows:

	(A1) and (A2)	(B1) and (B2)
(a)	012306	100334
(b)	132550	052267
(c)	041316	060215
(d)	150043	117720
(e)	117720	150043

- 7.3 Given: $(1000) = 177465$, $(1002) = 000313$.
Determine the values of the Z, N, C, and V codes after each one of the following instructions is executed. (Entries marked with an X need not be filled.)

Instruction	Z	N	C	V
TST 1000		X	X	
TSTB 1000		X	X	
TSTB 1001		X	X	
TST 1002		X	X	
TSTB 1002		X	X	
TSTB 1003		X	X	
CMP 1000,1002				
ADD 1000,1002				

- 7.4 The instruction `CMP A,B` is followed by a branch instruction of the form `B . . .L1`, where `. . .` is any of `GE`, `LT`, `GT`, `LE`, `HI`, `LOS`, `HIS`, or `LO`. Which of these eight instructions will result in a branch to location `L1` if the contents of `A` and `B` are as follows?

	(A)	(B)
(a)	017522	017522
(b)	104311	104311
(c)	177602	000176
(d)	054272	001016
(e)	177705	177613
(f)	004640	017227
(g)	105352	176051
(h)	151547	031246

- 7.5 The following program segment is supposed to clear the words addressed 000000 through 100000. Complete the fourth instruction.

```

LOOP    CLR      RO
        CLR      (RO)+
        CMP      #100000,RO
        B...     LOOP
        HALT

```

- 7.6 Assume that `X` and `X+2` contain a double-precision number `p`, and `Y` and `Y+2` contain the double-precision number `q`. (`X` and `Y` are the low-order words.) Describe what the following program segment does.

```

ADD     X+2,X+2
ADD     X,X
ADC     X+2
CMP     X+2,Y+2
BLT     L2
BGT     L1
CMP     X,Y

```

```

                BLOS      L2
L1:             MOV      #1,R0
                HALT
L2:             MOV      #2,R0
                HALT

```

7.7 Consider the following program:

```

START          MOV      X,R0
               CLR      R1
LOOP:          TST      RO
               BEQ      EXIT
               BPL      HERE
               ASL      R1
               INC      R1
HERE:          ASL      RO
               BR       LOOP
EXIT:          MOV      R1,Y
               HALT
X:             .WORD    123456
Y:             .BLKW    1

```

- (a) What is the final contents of Y?
 - (b) What does the program do in general [with an arbitrary (X)]?
- 7.8 Write an assembly language program that computes the double-precision product of two single-precision numbers. Assume that the multiplier and multiplicand are in addresses A and B; the result is left in C and C+2.
- 7.9 Write a subroutine that puts in R1 the number of 1-bits found in R0. [For example, if (R0) = 123456, then (R1) = 000011.]
- 7.10 Write an assembly language program which accepts a six-digit octal number from the teletype and prints out its binary equivalent.
- 7.11 Write and run a program that simulates a four-function stack calculator. The inputs to the calculator are signed or unsigned decimal numbers not exceeding 32767_{10} , and the operators + (add), - (subtract), * (multiply), / (integer-divide), S (change sign), X (clear top), and C (clear stack). Numbers and operators are entered through the teletype, each followed by a carriage return. Numbers are stored by the program in a stack (other than the system stack!). An operator +, -, *, or / results in the corresponding operation carried out on the two top stack elements, which are then replaced in the stack by the result. The operator S results in the replacement of the top stack element with its negative. X can be used to pop the stack top in case of a typing error. C initializes the stack and is to be used each time a

new calculation is to commence. Table 7.2 describes more precisely the action of the program. Table 7.3 shows an illustrative example.

Write the program in a highly modular form, using such subroutines as ATOB (which converts a decimal ASCII number to binary), BTOAS (which converts a binary number to a decimal ASCII number), MUL (a multiplying routine), DIV (a dividing routine), and PRINT (for printout).

TABLE 7.2 Operation of Stack Calculator

Input*	Stack Action	Printout	
Decimal number not exceeding 32767_{10} . May be preceded by either + or -.	Number is converted into binary and pushed onto stack.	Input is echoed.	
+ - * /	$\left. \begin{array}{l} \text{Stack top is popped into A;} \\ \text{stack top is popped into B;} \\ \text{the following is pushed} \\ \text{onto stack:} \end{array} \right\} \left\{ \begin{array}{l} (B)+(A) \\ (B)-(A) \\ (B)*(A) \\ (B)/(A) \end{array} \right\}$	Input is echoed. New stack top is printed out as a five-digit decimal number (possibly preceded by -), followed by CR and LF.	
S			Sign of stack top is changed.
X			Stack top is popped and ignored.
C			Stack is cleared (initialized).

*Always followed by CR.

TABLE 7.3 Example of Calculation of Stack Calculator

Expression to be evaluated:

$$-((-5)*((6+13)*(37-(125/7)))+(143/(-15))))$$

Input	Stack*	Printout
C		C
-5	-5	-5
6	6,-5	6
13	13,6,-5	13
+	19,-5	+
		00019
37	37,19,-5	37
125	125,37,19,-5	125
/	17,125,37,19,-5	7
		00017
-	20,19,-5	-
		00020
*	380,-5	*
		00380
143	143,380,-5	143
-15	-15,143,380,-5	-15
/	-9,380,-5	/
		-00009
+	371,-5	+
		00371
*	-1855	*
		-01855
S	1855	S
		01855

*Top to bottom is shown left to right.

TRAPS AND INTERRUPTS

8

In this chapter we study the mechanics and applications of traps and interrupts. We discuss illegal address and illegal instruction traps, the trap bit, the BPT instruction, and priority interrupts. The chapter concludes with a program illustrating the nesting of subroutines and interrupts and the manipulation of interrupt priorities.

8.1 TRAPS

To protect the user against various disasters, certain common program errors in the PDP-11 result in *processor traps* (i.e., in the *automatic* branch to fixed locations in the CM). Specifically, each such error is associated with some CM location α (which is fixed by the manufacturer and cannot be altered); when the error occurs, the following takes place automatically (in a single cycle):

```

MOV  PSR,-(SP)      ;Push processor status register onto system
                    ;stack
MOV  PC,-(SP)       ;Push program counter onto system stack
MOV  @# $\alpha$ ,PC      ;(PC) $\leftarrow$ ( $\alpha$ )
MOV  @# $\alpha+2$ ,PSR   ;(PSR) $\leftarrow$ ( $\alpha+2$ )

```

Thus, the PSR and PC are saved in the system stack, a new PSR is fetched from $\alpha + 2$, and a jump is executed to whatever address is encountered in α . If α contains the address β and $\alpha + 2$ contains the constant γ , then the CP proceeds to execute the program which starts at β , using γ as its PSR (see Figure 8.1). It is the programmer's responsibility to fill α and $\alpha + 2$ with β and γ (jointly referred to as a *trap vector*) and write a program (called a *trap routine*) which starts at β and which does whatever needs to be done when the error occurs (e.g., issues an error message and halts).

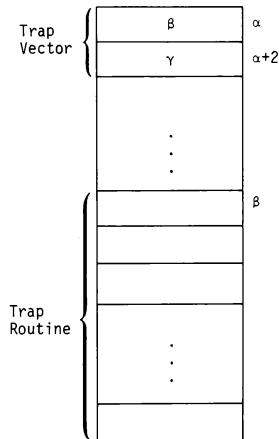


Figure 8.1 Trap vector and routine.

8.2 ILLEGAL ADDRESS AND ILLEGAL INSTRUCTION TRAPS

One of the most common programming errors is a reference to an illegal address, an attempt to execute a word instruction on an odd address or an attempt to address a nonexistent memory location. This error results in a trap through location 4, that is, in the following series of actions:


```
MOV PSR,-(SP)
MOV PC,-(SP)
MOV @#4,PC
MOV @#6,PSR
```

Another common error is an attempt to execute an illegal instruction (e.g., nonexistent op code). This results in a trap through location 10, that is, in the following series of actions:

```
MOV PSR,-(SP)
MOV PC,-(SP)
MOV @#10,PC
MOV @#12,PSR
```

The simplest thing to do in both of these cases is to cause the computer to halt. This can be accomplished (as proposed in Section 5.3) by initiating the program with

```
LC=.
.=4+LC
.WORD 6,0,12,0
```

which fills the locations 4, 6, 10, and 12 with the constants 6, 0, 12, and 0, respectively. When an illegal address is referenced, the CP traps through location 4 into location 6, which contains the instruction 000000 (i.e., a HALT instruction). Similarly, when an illegal instruction is attempted, the CP traps through location 10 into location 12, which also contains 000000.

Note that after the CP is halted by any of these two trap routines, the top of the system stack contains the value of PC at the time the error occurred. Also, PC contains either 10 or 14, depending on whether the trap routine executed was for illegal address or illegal instruction. Thus, by inspecting the contents of (SP) and of PC, one can determine the location and the type of error committed.

8.3 THE TRAP BIT AND BPT INSTRUCTION

From Figure 2.1 we see that the PSR contains, besides the condition codes N, Z, V, and C, a *trap bit* T in bit 4:



Like the other bits, the trap bit can be set and cleared under program control. When set, a processor trap will occur through location 14.

The trap bit is especially useful in the implementation of monitoring programs (such as tracing or debugging programs) whose function is to execute a user's program one instruction at a time, assuming control after each execution. The monitoring program can accomplish it by setting T to 1 before transferring control to the user's program. After the first of the user's instructions is executed, the CP traps through location 14 back to the monitoring program, which does whatever it is supposed to do. Using the information saved in the system stack when the user's program was trapped, the monitoring program can now yield control back to the user's program, returning to it the PC and PSR values it possessed when it was last interrupted. The second user's instruction is now executed, traps, and the process repeats.

A trap can be generated by program (rather than automatically by hardware) by issuing the instruction BPT ("breakpoint"). When this instruction is executed, a trap occurs through location 14.

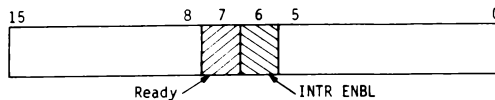
8.4 INTERRUPTS

An interrupt is another form of processor trap. While a processor trap, as we have seen, can be triggered by a programming error, an interrupt is triggered by a certain condition attained by a peripheral device. When this condition occurs, the program automatically branches to an address β stored in some fixed location α in the CM, while replacing the contents of the PSR with the contents γ of $\alpha + 2$. The contents of α and $\alpha + 2$ are referred to as an *interrupt vector* (analogous to a trap vector), and the program that starts at β is called an *interrupt routine* or an *interrupt handler* (analogous to a trap routine).

A basic difference between a system trap and an interrupt is that traps *always* take place when certain errors occur—they are beyond the programmer's control. On the other hand, the programmer does have control over interrupts; interrupts can be "disabled" either temporarily or permanently, if the programmer so desires.

We shall illustrate the interrupt mechanism with three peripheral devices: the teletype keyboard, the teletype printer, and the line clock. As we know from Sections 2.3 and 2.4, each one of these devices is associated with a status register whose bit 7 is the "ready" bit (indicating that a character

has been typed on the keyboard, or that the printer is free, or that the clock has “ticked”). We now introduce another important bit in the status register — bit 6, which is called the “interrupt enable” (or INTR ENBL) bit:



Each one of the three devices causes an interrupt when the “ready” bit in its status register changes from 0 to 1, provided that its INTR ENBL bit is 1. An interrupt will also occur if the “ready” bit is 1 and an instruction causes the INTR ENBL bit to change from 0 to 1. Thus, for an interrupt to occur, one of the two bits must be a 1 and the other must have just changed from a 0 to a 1.

The interrupt vectors for the teletype and line clock are located as follows:

Device	Interrupt Vector Location
TTY keyboard	60
TTY printer	64
Line clock	100

It is the programmer’s responsibility to initialize the INTR ENBL bit to 1 if interruption by the corresponding device is indeed contemplated (this bit is automatically cleared when the “start” switch is pressed in the PDP-11 or when a RESET instruction is issued). Of course, as in traps, it is also the programmer’s responsibility to supply the interrupt vector and the interrupt routine for each device.

While a trap routine might end with a HALT instruction, an interrupt routine almost always ends by returning control to the interrupted program. This returning of control is facilitated by the instruction RTI (“return from interrupt”), which executes the following (in a single cycle):

```
MOV (SP)+,PC      ;Pop system stack into program counter
MOV (SP)+,PSR     ;Pop system stack into processor status register
```

Thus, RTI simply undoes the actions taken by the interrupt mechanism just before the interrupt routine took over (see Section 8.1).

The way that interrupt routines assume and relinquish control is re-

miniscent of the way subroutines assume and relinquish control (using the JSR and RTS instructions). Both interrupt routines and subroutines use the system stack for their linkage information. However, in addition to a return address, an interrupt mechanism uses the stack to save the PSR of the interrupted program.

When more than one “interrupt-driven” device is used in a program, there is always the possibility of one interrupt routine interrupting another. The “nesting” of interrupt routines in this case is very similar to the nesting of subroutines. In fact, one can nest an arbitrary mixture of interrupt routines and subroutines without risking any confusion.

8.5

WHY USE INTERRUPTS?

The main advantage of the interrupt facility is that it saves the program the necessity of repeatedly testing (or “polling”) a device in order to determine whether or not it attained a certain condition. If the device is interrupt-driven, the occurrence of this condition will reveal itself automatically! Since the testing of a peripheral device is usually a relatively lengthy process, the interrupt facility may be an important time saver for the program. For example, in past programs we issued

```
LOOP: TSTB @#177564
      BPL LOOP
```

in order to ascertain that the printer is free. If the printer is interrupt-driven, this loop becomes superfluous, and the time the printer requires for completion (in the order of 1/10 second) can be used for the execution of tens of thousands of useful CP instructions.

The following are more detailed examples. In these examples, the reasons behind the components γ of the interrupt vector (200 in Example 1 and 300 in Example 2) will be explained in the next section.

Examples

1. We wish to use the TTY as an ordinary typewriter (i.e., have every type-in character echoed) while the main program is running. Figure 8.2 shows how this can be done. The main program runs until a character is typed in, at which point it is interrupted by the interrupt-driven keyboard and transfers control to INTHND. The interrupt routine, which starts at

```

        TITLE  INTYPE
; INTERRUPT DRIVEN ECHO PROGRAM
LC=.
.=4+LC
        .WORD  6,0,12,0           ;INITIALIZE ERROR VECTORS
.=60+LC
        .WORD  INTHND,200        ;INITIALIZE INTERRUPT VECTOR
.=500+LC
        ;ALLOW FOR STACK SPACE
START:  MOV    PC,SP
        TST   -(SP)              ;INITIALIZE SP TO START
        MOV   *100,@=177560      ;SET INTR ENBL BIT TO 1

        MAIN PROGRAM
        }
        } Main program

;
;
INTHND: MOV   @=177562,@=177566   ;PRINT INPUT CHARACTER
        RTI    ;RETURN FROM INTERRUPT
;
        .END   START

```

Figure 8.2 INTYPE program.

INTHND, simply echoes the character. (We assume a slow typist, and hence no need for buffering.)

Note that the main program resumes operation with the same PSR that it possessed when it was last interrupted. This is important, since the interruption might have occurred right after a TST or CMP instruction, which makes the preservation of the condition codes (and hence the PSR) essential.

2. We wish the bell to ring every 10 seconds (decimal) while a main program is running. Figure 8.3 shows how this can be done. The main program is interrupted by the line clock every 1/60 second. The interrupt routine decrements the “tick” count by 1 each time it is entered*; when this count becomes 0, it is reset to 600 decimal (600/60 = 10) and the bell rings. Thus, the bell rings immediately when the program is started (since the tick count is initially set to 1), and every 10 seconds thereafter. □

Although the preceding examples are rather contrived, they do illustrate the fact that, by means of interrupts, two programs can be executed by the computer “in parallel.”

*When the clock’s INTR ENBL bit is 1, the ready bit in the clock status register is cleared *automatically* after every tick.

```

        .TITLE  INBELL
; A BELL RINGS EVERY 10 SECONDS WHILE MAIN PROGRAM IS RUNNING.
LC=.
.=4+LC
        .WORD  6,0,12,0          ;INITIALIZE ERROR VECTORS
.=100+LC
        .WORD  INTHND,300        ;INITIALIZE INTERRUPT VECTOR
.=500+LC
START:  MOV    PC,SP              ;INITIALIZE SP TO START
        TST   -(SP)              ;SET INTR ENBL BIT TO 1
        MOV   #100,@#177546      ;INITIALIZE TICK COUNT
        MOV   #1,COUNT           ;INITIALIZE TICK COUNT
;
;           MAIN PROGRAM
;
;
;
;
;           INTERRUPT HANDLER
INTHND: DEC   COUNT              ;(COUNT)=(COUNT)-1
        BEQ   RING               ;IF (COUNT)=0, RING BELL
        RTI                       ;RETURN FROM INTERRUPT
RING:    MOV   #7,@#177566        ;RING BELL (ASCII CODE 007)
        MOV   #600,COUNT         ;SET COUNT TO 10 SECONDS
        RTI                       ;RETURN FROM INTERRUPT
;
COUNT  .BLKW 1                  ;TICK COUNT
;
        .END   START

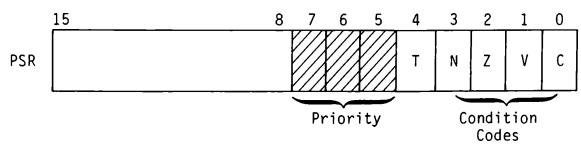
```

} Main program

Figure 8.3 INBELL program.

8.6 PRIORITY INTERRUPTS

To further understand the interrupt mechanism, we have to be familiar with the *priority field* of the PSR, which consists of bits 5, 6, and 7:



The contents of the PSR's priority field is referred to as the *CP priority*. It can be set by the programmer to any level between 0 and 7. For example,

to set the CP priority to 7, we issue

```
MOV #340,@#177776
```

(We shall assume that, when the program is started, the CP priority is set automatically to 0.)

In addition to the CP priority, one talks about the priority of a peripheral device. For example, the priorities of the TTY keyboard and printer are both 4, and the priority of the line clock is 6 (all of which are fixed by the manufacturer). The fundamental rule for interrupts is the following: *A device can interrupt a program only if the device's priority is strictly greater than the CP priority.* For example, the TTY and clock can interrupt any program that runs with CP priority 0; programs that run with CP priority 7 cannot be interrupted by any device!

Suppose that two interrupt-driven devices, D1 and D2, whose priorities are p_1 and p_2 , respectively, are used with a program that runs with CP priority p_0 . Assume that $p_0 < p_1 < p_2$. Sometime after the program is started, D1 attempts to interrupt. Since $p_1 > p_0$, the interrupt is "acknowledged" and D1's interrupt routine takes over. The CP priority at which this routine runs is determined by the new contents of the PSR—that is, by the second component (which we called γ) of D1's interrupt vector. Let's denote by p'_1 the new CP priority, coming from bits 5, 6, and 7 of γ . Suppose now that, while D1's interrupt is still being "serviced," D2 is attempting an interrupt. Whether or not this interrupt will be acknowledged depends on the relative magnitudes of p_2 and p'_1 . If $p_2 > p'_1$, then D2's interrupt routine will take over and be executed before control is returned (with RTI) to D1's interrupt routine. If $p_2 \leq p'_1$, D2's interrupt will have to remain "pending" until D1's interrupt routine is completed, control is returned (with RTI) to the main program and the priority is lowered back to p_0 .

Thus, by assigning appropriate γ 's to the interrupt vectors, the programmer can dictate which interrupt routines are interruptable by which other interrupt routines. Of course, the interrupt vectors are not the only means by which one can change the CP priority; one can always change it simply by issuing a `MOV #CONST,@#177776` instruction, where bits 5, 6, and 7 of `CONST` are the desired priority.

In most cases the CP priority at which we wish to run the interrupt routine of a device is identical with the device's priority. For example, suppose that we wish to write a program which, while executing a main program, acts as a typewriter and also rings a bell every 10 seconds (the combination

of Examples 1 and 2 of Section 8.5). We certainly want the main program to have lower priority than either the TTY keyboard or the clock. We also want the clock to have higher priority than the keyboard, since a clock interrupt every 1/60 second is essential if all “ticks” are to be kept track of. Thus, an appropriate choice of priorities could be:

Program	CP Priority
Main program	0
Keyboard-interrupt routine	4
Clock-interrupt routine	6

The main program’s CP priority of 0 is presumably in force when the program starts; the CP priority 4 of the keyboard interrupt routine can be enforced by storing 200 in address 62; the CP priority 6 of the clock interrupt routine can be enforced by storing 300 in address 102. Figure 8.4 illustrates the manner in which the CP is shared among the main program, the keyboard, and the clock.

8.7

EXAMPLE: TIME REQUEST

We shall close this chapter with a program that illustrates interrupts by two devices, the manipulation of CP priorities, and the nesting of subroutines (including a recursive subroutine) and interrupt routines.

The program (see Figure 8.5) opens with a query to the user: WHAT TIME IS IT?, to which the user is to respond with the correct time, in the form XXYY (e.g., 1143 if the correct time is 11:43). Thereafter, whenever the user types a character, the program prints out the message AT THE BELL THE TIME WILL BE:, followed by the correct time, in the form HH:MM:SS (hours:minutes:seconds), and a bell.

The main program (operating at CP priority 0) consists of the initial query and the echoing and conversion to binary of the user’s response XXYY. Subsequently, it enters an infinite loop (LOOP: BR LOOP) which actually can be replaced by any useful program of the user’s choosing (provided that it runs at CP priority 0). A clock-interrupt routine CLINT interrupts the main program every 1/60 second to update the internal

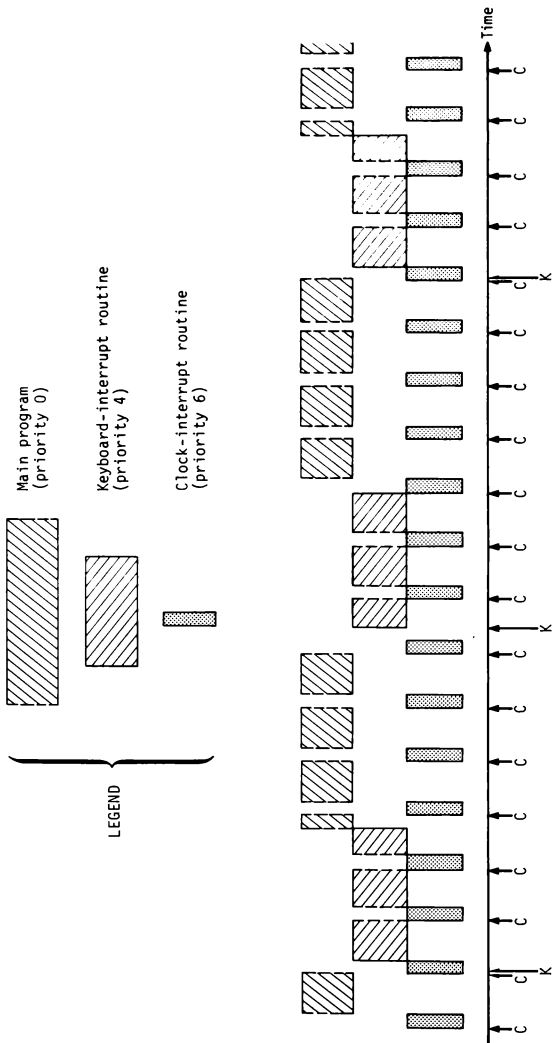


Figure 8.4 Priority interrupts.

clock stored at HOUR, MIN, SEC, and TICK. CLINT raises the CP priority from 0 to 6 (the γ component of the clock-interrupt vector) and hence cannot be interrupted by the keyboard (whose priority is 4). When a character is typed in, a keyboard-interrupt routine KBINT assumes control either immediately (if the main program is running) or right after CLINT is completed. The function of KBINT is to print the message AT THE BELL, etc.; convert to ASCII and print out the contents of HOUR, MIN, and SEC; and ring a bell—all while CLINT is keeping track of the time by updating HOUR, MIN, SEC, and TICK every 1/60 second. Since HOUR, MIN, and SEC might be altered while KBINT is still processing their contents, the first thing KBINT must do is copy them into temporary location (array TEMP). This must be done without CLINT's interruption and hence with CP priority of at least 6 (7 is selected, and is assigned to the γ component of the keyboard interrupt vector). As soon as the copying into TEMP is completed, KBINT lowers the CP priority to 0 (with CLR @#177776), ready to be interrupted by CLINT every 1/60 second as required.

The heart of CLINT is a recursive subroutine UPDATE, whose argument is a CM address Z containing a positive number less than 60_{10} . UPDATE is called with $Z = \text{TICK, SEC, MIN, and HOUR}$, in that order. It increments (Z) by 1 and checks whether the incremented (Z) equals 60; if not, UPDATE exits; otherwise, UPDATE clears (Z) to 0 and calls itself with the next address Z. For example:

	HOUR	MIN	SEC	TICK
Initial values	12	59	59	59
After UPDATE's 1st call	12	59	59	00
After UPDATE's 2nd call	12	59	00	00
After UPDATE's 3rd call	12	00	00	00
After UPDATE's 4th call	13	00	00	00
After correction by CLINT	01	00	00	00

The main program uses a subroutine INCON to convert the inputs XX and YY from ASCII to binary. For two-digit numbers, this subroutine is just as efficient as the one implemented in ASCTOBIN (Section 7.7).

Note that, before returning, KBINT must clear the “ready” bit in KBSTAT so that the next keyboard interrupt can be acknowledged. This can be done by any reference to KBDATA (e.g., TST KBDATA).

Remember that .WORD and .BLKW directives (such as those appearing at the end of the program) must initiate at *even* addresses. Thus, whenever in doubt (for example, after .BYTE, .BLKB, or .ASCII directives), preface them with an .EVEN directive.

```

        TITLE TIME
; IN RESPONSE TO THE PROGRAM'S QUERY "WHAT TIME IS IT?". THE USER
; INITIALIZES THE INTERNAL CLOCK BY TYPING THE TIME AS A 4-DIGIT NUMBER
; XXYX. THEREAFTER, WHENEVER A CHARACTER IS TYPED IN, THE PROGRAM PRINTS
; OUT THE MESSAGE "AT THE BELL THE TIME WILL BE:" FOLLOWED BY THE TIME IN
; THE FORMAT HH:MM:SS AND A BELL.
LC=.
.=4+LC
        .WORD    6,0,12,0           ;INITIALIZE ERROR VECTORS
.=60+LC
        .WORD    KBINT,340         ;INITIALIZE KEYBOARD INT. VEC. (PRIOR. 7)
.=100+LC
        .WORD    CLINT,300        ;INITIALIZE CLOCK INT. VEC. (PRIOR. 6)
.=500+LC
        .WORD    PC,SP            ;ALLOW FOR STACK SPACE
START:  MOV     PC,SP             ;INITIALIZE SP TO START
        TST    -(SP)
;
KBSTAT=177560
KBDATA=177562
PRSTAT=177564
PRDATA=177566
CLSTAT=177566
;
; PRINT QUERY
        MOV     #QUERY,R0        ;SET PARAMETERS
        MOV     #ENDQ,R1         ;   FOR PRINT SUBROUTINE
        JSR    PC,PRINT         ;PRINT LF, CR, QUERY TEXT
; ACCEPT AND ECHO INITIAL TIME XXYX
        MOV     #4,R2            ;(R2)=DIGIT COUNT
        MOV     #ITIME,R0       ;SET PARAMETERS
NEXTD:  MOV     R0,R1            ;   FOR PRINT SUBROUTINE
        TSTB   KBSTAT          ;CHARACTER ENTERED?
        BPL    .-4              ;IF NOT, KEEP TESTING
        MOVB   KBDATA,(R0)     ;ELSE, STORE DIGIT IN ITIME ARRAY
        BICB   #200,(R0)       ;REMOVE CHECK BIT FROM DIGIT
        JSR    PC,PRINT         ;PRINT DIGIT
        INC    R0               ;(R0)=(R0)+1
        DEC    R2               ;(R2)=(R2)-1
        BNE   NEXTD            ;IF (R2) NOT 0, ACCEPT NEXT DIGIT
; CONVERT INITIAL HOURS (XX) TO BINARY
        MOVB   ITIME+1,R0       ;SET PARAMETERS
        MOVB   ITIME,R1         ;   FOR INCON SUBROUTINE
        JSR    PC,INCON         ;CONVERT XX TO BINARY
        MOV    R2,HOUR          ;   AND STORE IN HOUR
; CONVERT INITIAL MINUTES (YY) TO BINARY
        MOVB   ITIME+3,R0       ;SET PARAMETERS
        MOVB   ITIME+2,R1       ;   FOR INCON SUBROUTINE
        JSR    PC,INCON         ;CONVERT YY TO BINARY
        MOV    R2,MIN           ;   AND STORE IN MIN
; SET INTERRUPT ENABLE BITS TO 1 AND WAIT
        MOV    #100,KBSTAT      ;SET KEYBOARD INTR ENBLE BIT TO 1
        MOV    #100,CLSTAT      ;SET CLOCK INTR ENBLE BIT TO 1
LOOP:   BR    LOOP             ;WAIT FOR INTERRUPTS
;

```

Figure 8.5 Time-request program.

```

;          CLOCK INTERRUPT HANDLER
; UPDATES TIME EVERY 1/60 SECOND
CLINT:  MOV    #TICK,R4          ;SET PARAMETER FOR UPDATE S.R.
        JSR    PC,UPDATE        ;UPDATE CLOCK COUNT
        CMP    HOUR,#12.        ;IS (HOUR)=12. OR LESS?
        BLE    EXIT3           ;IF SO, TIME UPDATE IS COMPLETE
        SUB    #12.,HOUR        ;ELSE, CORRECT FOR 12-HOUR CLOCK
EXIT3:  RTI                    ;RETURN FROM INTERRUPT
;
;          UPDATE (RECURSIVE SUBROUTINE)
; UPDATES TICK, SEC, MIN AND HOUR. ADDRESS OF UPDATED FIELD IS IN R4.
UPDATE: INC    (R4)             ;((R4))=((R4))+1
        CMP    (R4),#60.        ;((R4))=60.?
        BNE    EXIT4           ;IF NOT, UPDATING IS COMPLETE
        CLR    (R4)             ;ELSE, ((R4))=0 (RESET COUNT)
        TST    -(R4)           ;(R4)=(R4)-2 (GO TO NEXT FIELD)
        JSR    PC,UPDATE        ;UPDATE NEXT FIELD
EXIT4:  RTS                    ;EXIT
;
;          KEYBOARD INTERRUPT HANDLER
; PRINTS OUT TIME WHENEVER A CHARACTER IS TYPED IN.
KBINT:  MOV    #TEMP,R0         ;SAVE LATEST
        MOV    HOUR,(R0)+       ; HOUR, MIN AND SEC
        MOV    MIN,(R0)+        ; IN TEMP ARRAY TO
        MOV    SEC,(R0)         ; PROTECT FROM CLINT
        CLR    @#177776        ;LOWER PRIORITY TO ACCEPT CLINT
; PRINT MESSAGE
        MOV    #MESSG,R0        ;SET PARAMETERS
        MOV    #ENDM,R1         ; FOR PRINT SUBROUTINE
        JSR    PC,PRINT        ;PRINT LF, CR, MESSAGE TEXT
; CONVERT HOUR, MIN AND SEC TO ASCII
        MOV    #TEMP,R2        ;SET PARAMETERS
        MOV    #OUTPUT,R3       ; FOR OUTCON SUBROUTINE
        JSR    PC,OUTCON        ;CONVERT HOUR TO ASCII (HH)
        JSR    PC,OUTCON        ;CONVERT MIN TO ASCII (MM)
        JSR    PC,OUTCON        ;CONVERT SEC TO ASCII (SS)
; PRINT OUT HH:MM:SS AND RING BELL
        MOV    #OUTPUT,R0       ;SET PARAMETERS
        MOV    #END0,R1         ; FOR PRINT SUBROUTINE
        JSR    PC,PRINT        ;PRINT OUTPUT ARRAY
        TST    KBDATA          ;CLEAR READY BIT IN KBSTAT
        RTI                    ;RETURN FROM INTERRUPT
;
;          PRINT
; PRINTS STRING OF CHARACTERS STARTING AT (R0) AND ENDING AT (R1).
;CHANGES R5 ONLY.
PRINT:  MOV    R0,R5            ;(R5)=CHARACTER ARRAY INDEX
AGAIN:  CMP    R5,R1            ;HAS STRING ENDED?
        BHI    EXIT1           ;IF SO, EXIT
        TSTB  PRSTAT           ;IS PRINTER READY?
        BPL  -.4                ;IF NOT, KEEP TESTING
        MOVB  (R5)+,PRDATA      ;ELSE, PRINT ((R5)). (R5)=(R5)+1
        BR   AGAIN             ;PICK UP NEXT CHARACTER
EXIT1:  RTS                    ;EXIT
;

```

Figure 8.5 (cont.)

```

;          INCON
; CONVERTS A 2-DIGIT DECIMAL NUMBER STORED IN ASCII IN R0 (UNITS) AND
; R1 (TENS) INTO BINARY. THE RESULT IS PLACED IN R2, R3, R4, R5 UNCHANGED.
INCON:  BIC    #177760,R0      ;CONVERT (R0) INTO BINARY
        MOV    R0,R2          ; AND STORE IN R2
TENS:   CMPB   R1,#'0         ;(R1)='0'? (ANY TENS LEFT?)
        BEQ   EXIT2          ;IF NOT, EXIT
        ADD   #10.,R2        ;ELSE, (R2)=(R2)+10 DECIMAL
        DEC   R1             ;(R1)=(R1)-1 (1 TEN LESS)
        BR    TENS          ;CHECK FOR TENS AGAIN
EXIT2:  RTS    PC            ;EXIT
;
;          OUTCON
; CONVERTS A BINARY NUMBER N (FROM 0 TO 60 DECIMAL) INTO A 2-DIGIT
; ASCII NUMBER PQ. ADDRESS OF N IS (R2). ADDRESSES OF P AND Q ARE (R3)
; AND (R3)+1. BEFORE EXIT THE CONTENTS OF R1 IS INCREMENTED BY 2 AND OF
; R3 BY 3. R4 AND R5 ARE UNCHANGED.
OUTCON: MOV    (R2)+,R0      ;(R0)=BINARY NUMBER (HOUR, MIN, SEC)
        CLR    R1           ;INITIALIZE TENS
MORE:   CMP    R0,#10.      ;ANY TENS LEFT IN R0?
        BLT   UNITS        ;IF NONE, PROCESS UNITS
        INC   R1           ;ELSE, (R1)=(R1)+1 (ONE MORE TEN)
        SUB   #10.,R0      ;(R0)=(R0)-10 DECIMAL
        BR    MORE         ;CHECK FOR MORE TENS
UNITS:  ADD   #'0',R1       ;CONVERT TENS TO ASCII
        ADD   #'0',R0      ;CONVERT UNITS TO ASCII
        MOVB  R1,(R3)+     ;STORE TENS IN OUTPUT ARRAY
        MOVB  R0,(R3)+     ;STORE UNITS IN OUTPUT ARRAY
        INC   R3           ;SKIP COLON BYTE
        RTS   PC           ;EXIT
;
;          STORAGE FOR CONSTANTS AND TEMPORARIES
;
QUERY:  .BYTE  15,12        ;CR, LF
        .ASCII /WHAT TIME IS IT?/ ;QUERY TEXT
ENDQ:   .ASCII  / /        ;END OF QUERY (SPACE)
;
MESSG:  .BYTE  15,12        ;CR, LF
        .ASCII /AT THE BELL THE TIME WILL BE:/ ;MESSAGE TEXT
ENDM:   .ASCII  / /        ;END OF MESSAGE (SPACE)
;
OUTPUT: .ASCII  /HH:MM:SS/  ;STORATE FOR HH:MM:SS
ENDO:   .BYTE  7           ;END OF OUTPUT (BELL)
;
ITIME:  .BLKB  4           ;STORAGE FOR INITIAL TIME (XXYY)
;
        .EVEN             ;ADJUST WORD BOUNDARY
HOUR:   .BLKW  1           ;STORAGE FOR HOURS (BINARY)
MIN:    .BLKW  1           ;STORAGE FOR MINUTES (BINARY)
SEC:    .WORD  0           ;STORAGE FOR SECONDS (BINARY)
TICK:   .WORD  0           ;STORAGE FOR TICK COUNT (BINARY)
TEMP:   .BLKW  3           ;TEMP. STORAGE FOR HOUR, MIN, SEC
;
        .END    START

```

Figure 8.5 (cont.)

- 8.1 The following program is run on a PDP-11 model whose CM consists of 8K words. What are the contents of RO, SP, PC, and the system stack when the program halts?

```

LC=.
.= 4+LC
        .WORD X,0
.= 500+LC
START:  MOV   PC,SP
        TST  -(SP)
        CLR  RO
LOOP:   TST  (RO)+
        BR   LOOP
X:      SUB  #2,RO
        HALT
        .END  START

```

- 8.2 The following program consists of a main program and a clock-interrupt routine (starting at CLINT).

```

LC=.
.=100+LC
        .WORD CLINT,300      ;INITIALIZE INTERRUPT VECTOR
.=500+LC
START:  MOV   PC,SP          ;ALLOW FOR STACK SPACE
        TST  -(SP)          ;INITIALIZE SP TO START
        MOV  #100,@#177546  ;SET INTR ENBL BIT TO 1
LOOP:   BR   LOOP           ;WAIT FOR CLOCK INTERRUPT
NOW:    MOV  RO,R1          ;AFTER INTERRUPT STORE RO IN R1
        HALT

;
CLINT:  ← [Insert one instruction here]
        RTI                 ;RETURN FROM INTERRUPT
        .END

```

Fill in the missing instruction so that the following is accomplished: After the first return from interrupt, the main program resumes at location NOW (rather than LOOP), with the contents of SP set to 500.

- 8.3 In the following program, determine the contents of PC, SP, and the system stack. (Note: 177777 is an illegal instruction!)

```

LC=.
.=4*LC
        .WORD    6,0,12,0
.=500+LC
START:  MOV     PC,SP
        TST    -(SP)
        CLR    177776
        JSR    PC,SUB
        HALT
SUB:    MOV     #123123,-(SP)
        ASR    (SP)
        .WORD  177777
        RTS    PC
        .END   START

```

8.4 Verify that the following program eventually halts. What are the final contents of PC, PSR, and the system stack?

```

LC=.
.=10+LC
        .WORD    514,340
.=100+LC
        .WORD    100,300
.=500+LC
START:  MOV     PC,SP
        TST    -(SP)
        MOV     #100,@#177546
LOOP:   BR     LOOP
        HALT
        .END    START

```

8.5 Consider the following program equipped with a clock-interrupt routine INTHND:

```

LC=.
.=100+LC
        .WORD    INTHND,10
.=500+LC
START:  MOV     PC,SP
        TST    -(SP)
        MOV     #100,177546
        MOV     #5,R0
        TST
        BMI    L1
        BPL    L2
        INC     R0
        HALT
L1:    DEC     R0
L2:    HALT
INTHND: MOV     177776,2(SP)
        RTI
        .END    START

```

What are the contents of R0 when the program halts if the first clock interrupt occurs at: (a) point t1, (b) point t2, (c) point t3, (d) point t4?

- 8.6 Devices D1, D2, D3, and D4 have device priorities 4, 5, 6, and 7, respectively; the start locations of their interrupt vectors (i.e., the addresses α) are 60, 100, 170, and 174, respectively.

Consider the following program:

```

LC=.
.=4+LC
    .WORD    6,0
.=60+LC
    .WORD    INT1,200
.=100+LC
    .WORD    INT2,240
.=170+LC
    .WORD    INT3,0
.=174+LC
    .WORD    INT4,340
.=500+LC
START:
    -
    -
    -
ENTER1: MOV    #100,17776
    -
    -
    -
ENTER2: MOV    #300,17776
    -
    -
    -
INT1:
    -
    -
    -
INT2:
    -
    -
    -
INT3:
    -
    -
    -
INT4:
    -
    -
    -
.END START

```

} SP is initialized to 500¹ and the INTR ENBL bits of
all devices are set to 1

} subprogram ENTER1

} subprogram ENTER2

} subprogram INT1

} subprogram INT2

} subprogram INT3

} subprogram INT4

For each of the six subprograms (ENTER1, ENTER2, INT1, INT2, INT3, and INT4), determine which of the four devices (D1, D2, D3, D4) can cause an interrupt.

- 8.7 Consider the devices D1, D2, D3, and D4 specified in Problem 8.6. Modify the program in that problem so that:

ENTER1 can be interrupted by D3 and D4.
ENTER2 can be interrupted by D2, D3, and D4.
INT1 can be interrupted by D3 and D4.
INT2 can be interrupted by D1, D2, D3, and D4.
INT3 can be interrupted by none of the devices.
INT4 can be interrupted by D1, D2, D3, and D4.

- 8.8 Write trap routines for illegal address and illegal instruction errors which print out the type of error committed and the contents of PC at the time the error was made.
- 8.9 The INTYPE program of Figure 8.2 assumes sufficiently slow typing so that no input buffering is necessary. Revise the program for the case where this assumption can no longer be made.
- 8.10 Write and run a program that does the following: while a main program (simulated by LOOP: BR LOOP) is running, a bell rings every 10 seconds; when a keyboard key is struck, the main program and the bell ringing are suspended, and successive bytes of memory are printed out in ASCII, starting at some address stored in R0; when a keyboard key is struck again, printout stops and the main program and bell ringing resume. This process can be repeated any number of times.
- Include with the program a clock-interrupt routine CLINT for timing the bells, and a keyboard-interrupt routine KBINT for printing the bytes in (R0), (R0) + 1, (R0) + 2, Both CLINT and KBNIT should be able to interrupt the main program; CLINT may not interrupt KBINT.
- 8.11 Given an arbitrary program, propose a method for measuring its execution time. How accurate is the method?

THE ASSEMBLER AND LINKAGE EDITOR



High-level languages (such as FORTRAN, BASIC, PASCAL, etc.) have the advantage of enabling the user to easily write complex programs, unencumbered by the details of routine operations (e.g., a loop can be simply implemented by an instruction such as `D0 10 I = 1,100,3`). As compared with assembly language programs, high-level language programs are easier to write, comprehend, debug, and maintain. So why do we need assembly language? The reason is that, unlike assembly language programs, high-level language programs can rarely take full advantage of the organization (the CM, CP, I/O) of the particular computer on which they are intended to run. Hence, such programs—even when translated by a highly sophisticated compiler—are not always very efficient. Thus, in cases where speed is of prime importance (as in many utility programs), assembly language must be resorted to. Another case where one must program in assembly language is when the computer at hand is simply not equipped with (or cannot support) compilers for higher-level languages.

In this chapter we shall describe the two programs that make assembly language programming possible: the assembler and the linkage editor. Al-

though the discussion revolves around the PDP-11's MACRO-11 and LINKR-11, it is general enough to apply to many other assemblers and linkage editors currently being used.

9.1 THE TWO-PASS ASSEMBLY PROCESS

As we already know, the basic difference between assembly language and machine language is that in assembly language one can use symbolic op codes and addresses instead of numerical op codes and addresses. The correspondence between the symbols and their numerical values is recorded by the assembler in two tables, the *op-code table* and *symbol table*.

The op-code table (which is essentially predefined for all assemblies) lists all mnemonic op codes, each with its numerical equivalent. The symbol table (which varies from one program to another) lists all symbolic addresses and operands defined in the program, each with the numerical value that it represents.

Also central to the operation of the assembler is the *location counter* (symbolized in assembly language by `.`), which throughout the assembler's operation holds the CM address of the instruction currently being processed. Initially, the contents of the location counter is 0.

MACRO-11, like most assemblers, is a *two-pass assembler*—that is, it scans the source program *twice* before producing the object code. These two scans, or *passes*, have the following objectives:

Pass I: To search the source program for all symbol definitions and enter these into the symbol table.

Pass II: Using the symbol table constructed in pass I and the op-code table, to generate the machine language equivalent of every instruction in the source program.

The flowcharts of passes I and II are shown in Figure 9.1 and 9.2. These flowcharts are greatly simplified and convey only the highlights of the assembly process. In particular, they do not indicate the assembler's responses to assembler's directives. Here are some examples:

1. `.WORD d1,d2, . . . ,dr` (or `.BYTE d1,d2, . . . ,dr`) causes the assembler to fill consecutive words (or bytes) with the binary equiv-

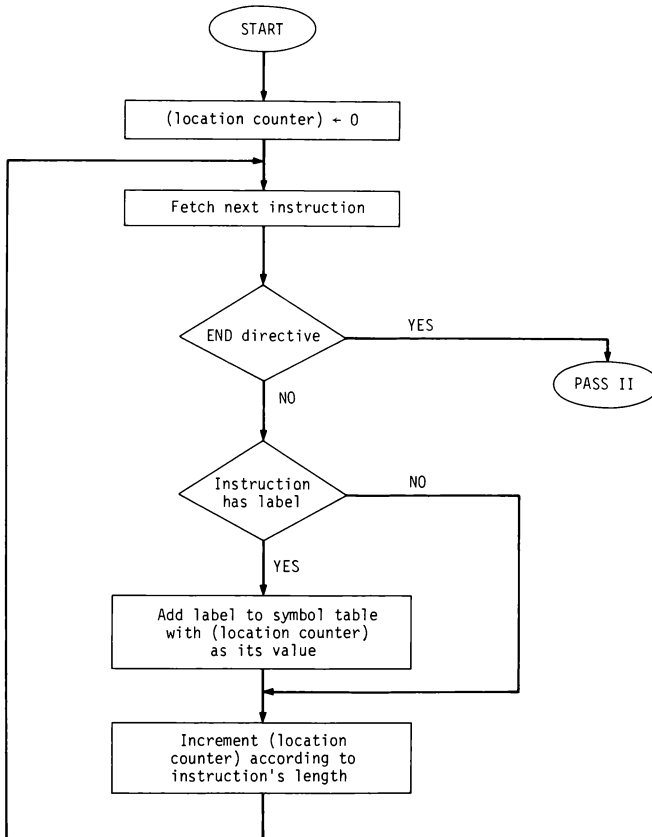


Figure 9.1 Assembler's pass I.

alents of d_1, d_2, \dots, d_r , incrementing the location counter by 2 (or 1) after each insertion.

2. `.ASCII /str/` causes the assembler to fill consecutive bytes with the binary ASCII code of the characters of "str," incrementing the location counter by 1 after each insertion.
3. `.BLKW n` (or `.BLKB n`) causes the assembler to increment the location counter by $2n$ (or n) — that is, to skip n words (or bytes).

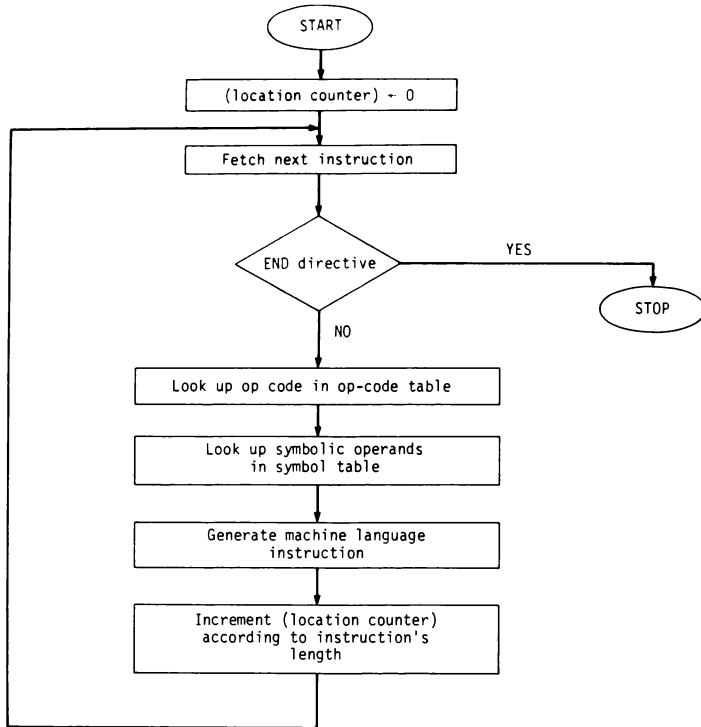


Figure 9.2 Assembler's pass II.

4. An assignment directive `sym = exp` causes the assembler (in pass I) to enter the symbol “sym” with its value “exp” into the symbol table.*

The flowcharts also fail to show various diagnostic facilities of the assembler. For example, the assembler flags a label as “multiply defined” if (in pass I) it is already found in the symbol table. It flags an instruction as erroneous if (in pass II) its op code is not found in the op-code table or its symbolic operand is not found in the symbol table.

*If “exp” involves a symbol not yet defined (a “forward reference”), an extra pass is required to determine its value.

EXAMPLE OF ASSEMBLER LISTING

In Section 7.7 we described a program ASCTOBIN which converts a number from decimal ASCII form into binary. Passing this program through MACRO-11 creates the object program of ASCTOBIN as well as the listing of the source program, object program, and symbol table. Figure 9.3 shows the printout of this listing.

The assembler listing shows each source instruction together with the object code to which it corresponds. Octal numbers followed by apostrophe (e.g., 000526') are addresses that require modification when the program is "relocated." More on that in the next section.

The symbol table contains (in alphabetic order) all the symbols defined by the program (those defined by an assignment directive are indicated by =). The meaning of the suffix R will be explained later.

```

1          .TITLE  ASCTOBIN
2          ; CONVERTS A TYPED-IN DECIMAL NUMBER N INTO ITS BINARY EQUIVALENT.
3          ; N MAY BE PREFIXED WITH + OR - AND MUST BE FOLLOWED BY A CARRIAGE RETURN.
4          ; THE BINARY EQUIVALENT OF N IS LEFT IN R2. IF N'S MAGNITUDE EXCEEDS
5          ; 32767 DECIMAL, R2 WILL BE LEFT WITH 100000 OCTAL.
6          000000'LC=.
7          000004'=.+4+LC
8 000004 000006      .WORD  6,0,12,0      ;INITIALIZE ERROR VECTORS
          000000
          000012
          000000
9          000500'=.+500+LC      ;ALLOW FOR STACK SPACE
10 00500 010706 START:  MOV   PC,SP
11 00502 005746      TST   -(SP)      ;INITIALIZE SP TO START
12          ;
13          177560  KBSTAT=177560
14          177562  KBDATA=177562
15          177564  PRSTAT=177564
16          177566  PRDATA=177566
17          000012  LF=12
18          000015  CR=15
19          ;
20          ;
21 00504 012703      MOV   MAIN PROGRAM
          #STRING,R3      ;(R3)=STRING
          000526'
22 00510 004767      JSR   PC,INPUT      ;STORE INPUT STRING IN ARRAY
          000036
23 00514 012703      MOV   #STRING,R3      ;(R3)=STRING
          000526'
24 00520 004767      JSR   PC,ATOB      ;CONVERT STRING INTO BINARY
          000106
25 00524 000000      HALT
26          ;
27 00526      STRING: .BLKB  20.      ;STORAGE FOR TYPED-IN STRING
28          ;
29          ;      INPUT
30          ; ECHOES TYPED-IN CHARACTERS AND STORES THEM IN BYTE ARRAY WHOSE BASE

```

Figure 9.3 ASCTOBIN – assembler listing.

```

31 ;ADDRESS IS IN R3. EXITS AFTER CR IS TYPED. CHANGES R3, R5.
32 00552 105767 INPUT: TSTB KBSTAT ;IS CHARACTER IN?
    177002'
33 00556 100375 BPL INPUT ;IF NOT, WAIT
34 00560 016705 MOV KBDATA,R5 ;(R5)=CHARACTER
    176776'
35 00564 004767 JSR PC,PRINT ;PRINT CHARACTER
    000026
36 00570 042705 BIC #177600,R5 ;REMOVE CHECK BIT
    177600
37 00574 110523 MOVB R5,(R3)+ ;STORE CHAR. IN ARRAY, UPDATE INDEX
38 00576 022705 CMP #CR,R5 ;IS CHARACTER CR?
    000015
39 00602 001363 BNE INPUT ;IF NOT, ACCEPT NEXT CHARACTER
40 00604 012705 MOV #LF,R5 ;ELSE,
    000012
41 00610 004767 JSR PC,PRINT ; MOVE TO NEXT LINE
    000002
42 00614 000207 RTS PC ;EXIT
43 ;
44 ; PRINT
45 ; PRINTS CONTENTS OF R5. REGISTERS UNCHANGED.
46 00616 105767 PRINT: TSTB PRSTAT ;IS PRINTER READY?
    176742'
47 00622 100375 BPL PRINT ;IF NOT, WAIT
48 00624 010567 MOV R5,PRDATA ;IF SO, PRINT (R5)
    176736'
49 00630 000207 RTS PC ;EXIT
50 ;
51 ; ATOB
52 ; CONVERTS INTO BINARY A DECIMAL NUMBER N STORED IN ASCII IN BYTE ARRAY
53 ; WHOSE BASE ADDRESS IS IN R3. N MAY BE PREFIXED WITH + OR - AND MUST BE
54 ; FOLLOWED BY A NON-DIGIT. THE BINARY EQUIVALENT OF N IS LEFT IN R2. IF
55 ; N'S MAGNITUDE EXCEEDS 32767 DECIMAL, R2 IS LEFT WITH 10000 OCTAL.
56 ; REGISTER ALLOCATION:
57 ; (R0), (R1) ARE USED FOR INPUTTING MUL PARAMETERS
58 ; (R2) = CONVERTED NUMBER
59 ; (R3) = POINTER TO NEXT CHARACTER
60 ; (R4) = SCANNED CHARACTER
61 ; (R5) = SIGN FLAG (0 IF N IS POSITIVE, 1 OTHERWISE)
62 00632 005005 ATOB: CLR R5 ;ZERO SIGN FLAG (ASSUME N POSITIVE)
63 00634 112304 MOVB (R3)+,R4 ;(R4)=SCANNED CHARAC., UPDATE INDEX
64 00636 122704 CMPB #'+,R4 ;IS CHARACTER +?
    000053
65 00642 001406 BEQ ATOB2 ;IF SO, START CONVERTING
66 00644 122704 CMPB #'-,R4 ;IS CHARACTER -?
    000055
67 00650 001002 BNE ATOB1 ;IF NOT, N IS UNSIGNED
68 00652 005205 INC R5 ;IF SO, SET SIGN FLAG FOR NEGATIVE N
69 00654 000401 BR ATOB2 ;START CONVERTING
70 00656 005303 ATOB1: DEC R3 ;CHARACTER IS DIGIT. BACKTRACK
71 00660 005002 ATOB2: CLR R2 ;INITIALIZE RESULT TO 0
72 00662 112304 ATOB3: MOVB (R3)+,R4 ;(R4)=SCANNED CHARACTER. UPDATE INDEX
73 00664 122704 CMPB #'0,R4 ;IF '0'(R4) (NONDIGIT)
    000060
74 00670 101016 BHI ATOB4 ; PREPARE FOR EXIT
75 00672 122704 CMPB #'9,R4 ;IF '9'(R4) (NONDIGIT)
    000071
76 00676 103413 BLO ATOB4 ; PREPARE FOR EXIT
77 00700 042704 BIC #177760,R4 ;CONVERT DIGIT TO BINARY
    177760
78 00704 012700 MOV #10.,R0 ;(R0)=10. (MUL PARAMETER)
    000012

```

Figure 9.3 (cont.)

```

79 00710 010201      MOV     R2,R1      ;(R2)=(R1) (MUL PARAMETER)
80 00712 004767      JSR     PC,MUL     ;(R2)=(R0)*(R1)=10.*(R2)
                   000026
81 00716 102407      BVS    ATOB6      ;IF OVERFLOW, PREPARE FOR EXIT
82 00720 060402      ADD    R4,R2      ;(R2)=(R4)+(R2)
83 00722 102405      BVS    ATOB6      ;IF OVERFLOW, PREPARE FOR EXIT
84 00724 000756      BR     ATOB3      ;SCAN NEXT CHARACTER
85                   ;NORMAL EXIT
86 00726 005705      ATOB4: TST    R5      ;TEST SIGN FLAG
87 00730 001401      BEQ    ATOB5      ;IF NUMBER IS POSITIVE, EXIT
88 00732 005402      NEG    R2         ;ELSE, (R2)=- (R2)
89 00734 000207      ATOB5: RTS    PC      ;EXIT
90                   ;OVERFLOW EXIT
91 00736 012702      ATOB6: MOV    #100000,R2 ;(R2)=100000
                   100000
92 00742 000207      RTS    PC         ;EXIT
93                   ;
94                   ;           MUL
95                   ; COMPUTES (R0)*(R1) AND STORES RESULT IN R2. IF RESULT'S MAGNITUDE
96                   ; EXCEEDS 32767 DECIMAL, V BIT IS SET TO 0. R3, R4, R5 NOT USED.
97                   ;
98 00744 005002      MUL:  CLR    R2         ;(R2)=0
99 00746 032701      MUL1: BIT    #1,R1      ;TEST BIT 0 OF R1
                   000001
100 0752 001402      BEQ    MUL2        ;IF 0, DON'T ADD
101 0754 060002      ADD    RO,R2      ;ELSE, (R0)=(R0)+(R2)
102 0756 102406      BVS    MUL3        ;EXIT IF OVERFLOW
103 0760 000241      MUL2: CLC        ;CLEAR C BIT
104 0762 006001      ROR    R1         ;ROTATE R1 1 BIT RIGHT
105 0764 006300      ASL    RO         ;ARITH.-SHIFT (R0) 1 BIT LEFT
106 0766 102402      BVS    MUL3        ;EXIT IF OVERFLOW
107 0770 005701      TST    R1         ;TEST (R1)
108 0772 001365      BNE    MUL1        ;IF NOT 0, KEEP MULTIPLYING
109 0774 000207      MUL3: RTS    PC      ;EXIT
110                   ;
111                   000500' .END  START

```

```

SYMBOL TABLE
ATOB  000632R      ATOB1  000656R      ATOB2  000660R
ATOB3  000662R      ATOB4  000726R      ATOB5  000734R
ATOB6  000736R      CR     = 000015    INPUT  000552R
KBDATA= 177562     KBSTAT= 177560     LC     = 000000R
LF     = 000012     MUL    000744R      MUL1   000746R
MUL2   000760R      MUL3   000774R      PRDATA= 177566
PRINT  000616R      PRSTAT= 177564     START  000500R
STRING 000526R

```

Figure 9.3 (cont.)

9.3 ABSOLUTE AND RELOCATABLE ADDRESSES

As we have seen in Section 9.1, the assembler's location counter is always initialized to the value 0. Consequently, the object program expects to be loaded starting at address 0. There are many cases, however, in which

it might be desirable to “relocate” a program—that is, to change its *loading origin* (the address at which loading starts) from 0 to some other value. For example, when a large program is formed by “linking” a number of independently assembled programs, it is clear that only one of them can have the loading origin 0—all the rest must be relocated. Also, if several programs, belonging to different users, are to occupy the memory simultaneously—as in a time-sharing environment—only one of these can be loaded at 0 and all the rest must be relocated.

The questions arise: will the object program execute properly after relocation, and if not—how can it be modified so as to make it execute properly? Before answering these questions, we must make a distinction between two types of addresses defined by a program: *absolute addresses* and *relocatable addresses*. In their most rudimentary form, absolute addresses are simply numbers or characters (or symbols equated to numbers or characters via assignment directives); relocatable addresses are simply labels (or symbols equated to labels via assignment directives). For example, in the MOV, ADD, and BR instructions contained in the following program segment, the addresses 300 and X are absolute and the addresses A, B, and Y are relocatable:

```

X = 100
Y = B
    ...
    MOV A,300
    ADD X,B
    BR Y
A:   ...
    ...
B:   ...

```

Addresses may also be specified by means of *address expressions*, that is, by means of sums and differences of addresses. In deciding whether an address thus specified is absolute or relocatable, the following rules are employed:

$$\begin{aligned} \text{absolute} \pm \text{absolute} &= \text{absolute} \\ \text{relocatable} - \text{relocatable} &= \text{absolute}^* \\ \text{absolute} \pm \text{relocatable} &= \text{relocatable} \end{aligned}$$

For example, in the MOV and ADD instructions contained in the following program segment, the values $P+Q-100$, $D-C$, and $C-D+Q$ are absolute and the address $C-P$ is relocatable.

*Relocatable + relocatable is undefined.

```

P = 200
Q = 'Z
    ...
    MOV P+Q-100,C-P
    ADD D-C,C-D+Q
C:   ...
D:   ...

```

It is assumed (and this should be the programmer's intent) that *all absolute addresses remain intact regardless of the program's loading origin, while all relative addresses, upon relocation, be changed by the same amount as the loading origin*. For example, in ASCTOBIN we want KBSTAT and CR to retain the values 177560 and 15 regardless of the program's location (since the keyboard status register is always in 177560 and the ASCII code for carriage return is always 15). On the other hand, STRING is 526 only when the program's loading origin is 0; it should be changed to 1526 when the loading origin is changed to 1000.*

In the assembler's listing of the symbol table the relocatable symbols are marked with the suffix R.

The symbol . (representing the location counter) is regarded by MACRO-11 as relocatable. Hence, in MACRO-11 a directive such as . = 500 is illegal. What we can do instead (and this is what we have invariably done in the past) is initiate the program with LC = . (which defines LC as "relocatable 0") and then issue . = 500 + LC, which assigns to the location counter the relocatable value 500 + LC, which is 500 relative to the loading origin.

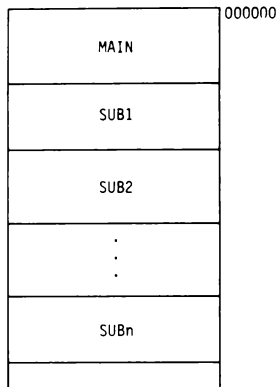
9.4 THE LINKAGE EDITOR

It is often the case that subroutines needed in a user's program are already available (for example, they have been previously written by the user for other programs, or they are included in the system's library of "standard" subroutines). In this case what one wishes to do is to combine the object codes of the main program and subroutines (all of which are referred to as *object modules*) into one large machine language program (called a *load module*) ready to be loaded and executed. This combining or "linking" job is performed by the *linkage editor*.†

*Addresses associated with .BLKW or .BLKB must always be absolute.

†The linkage editor described in this chapter is called LINKR-11.

Suppose that the main program MAIN and the subroutines SUB1, SUB2, . . . , SUBn have all been assembled independently. Passing the corresponding $n + 1$ object modules (and a specification as to their ordering) to the linkage editor results in a load module which looks as shown below. This object module can be loaded in the CM of the PDP-11 (starting at address 0) and executed (starting at the address originally attached to the .END directive).



Generally, the effect of the linkage editor is the same as if MAIN, SUB1, SUB2, . . . , SUBn were all assembled at the same time. The question may be asked: Why don't we store all the source codes (that of MAIN, SUB1, SUB2, . . . , SUBn) in a single file and then assemble them together, in which case no linking will be necessary. The answer is that sometimes the source codes for SUB1, SUB2, . . . , SUBn are simply not accessible to the programmer (for example, when these are library subroutines). In other cases the source language of SUB1, SUB2, . . . , SUBn is not assembly language but some other language (e.g., FORTRAN), and translation to machine language cannot be done with an assembler but with a special compiler. In all these cases the linkage editor is indispensable.

In converting the object modules into a load module, the linkage editor must perform the following two functions:

1. It must, whenever necessary, modify addresses affected by relocation.
2. It must supply "external" addresses—that is, addresses referred to in one object module but defined in another.

The details of these two functions are described in the following two sections.

9.5 ADDRESS MODIFICATION

In view of our discussion in Section 9.3, we wish the linkage editor to modify addresses according to the following criteria:

1. A relocatable address should be changed by the same amount as the loading origin.
2. An absolute address should remain intact regardless of the loading origin.

Table 9.1 shows examples of instructions and how they are to be modified after relocation. To make the linkage editor's job easier, each address that must be modified is marked by the assembler with an apostrophe.

From Table 9.1 we can deduce the following address modification rules:

1. All relocatable addresses should be *increased* by the same amount as the loading origin, except when used in the relative mode,* in which case no modification is required.
2. All absolute addresses are to be left intact, except when used in the relative mode, in which case they should be *decreased* by the same amount as the loading origin.

Note that branch instructions do not specify addresses, but *offsets*, and hence never require any modification.

9.6 GLOBAL SYMBOLS

Suppose that we wish to link the subprograms MAIN, SUB1, SUB2, . . . , SUBn, all assembled separately. It is unavoidable that some of them will be referring to symbols that are defined externally (that is, defined by

*In this chapter "relative mode" refers to both the relative and the relative deferred modes of addressing (see Sections 4.6 and 4.7).

TABLE 9.1 Examples of Relocation

Example	Source Program	Object Program	Loading Origin 1000	Modification
1	MOV A,R0	600: 016700	1600: 016700	← No change
	
	A: ...	704: ...	1704: ...	
2	MOV #A,R0	600: 012700	1600: 012700	← 1000 added
	...	602: 000704'	1602: 001704	
	A: ...	704: ...	1704: ...	
3	MOV @#A,R0	600: 013700	1600: 013700	← 1000 added
	...	602: 000704'	1602: 001704	
	A: ...	704: ...	1704: ...	
4	MOV A(R0),R1	600: 016001	1600: 016001	← 1000 added
	...	602: 000704'	1602: 001704	
	A: ...	704: ...	1704: ...	
5	MOV @A(R0),R1	600: 017001	1600: 017001	← 1000 added
	...	602: 000704'	1602: 001704	
	A: ...	704: ...	1704: ...	
6	MOV 177776,R0	600: 016700	1600: 016700	← 1000 subtracted
	...	602: 177172'	1602: 176172	
	A: ...	704: ...	1704: ...	
7	MOV #15,R0	600: 012700	1600: 012700	← No change
	...	602: 000015	1602: 000015	
	A: ...	704: ...	1704: ...	
8	MOV @#60,R0	600: 013700	1600: 013700	← No change
	...	602: 000060	1602: 000060	
	A: ...	704: ...	1704: ...	
9	MOV 20(R0),R1	600: 016001	1600: 016001	← No change
	...	602: 000020	1602: 000020	
	A: ...	704: ...	1704: ...	
10	MOV @20(R0),R1	600: 017001	1600: 017001	← No change
	...	602: 000020	1602: 000020	
	A: ...	704: ...	1704: ...	

other subprograms). For example, MAIN may be referring to a label defined in SUB1 (by issuing JSR PC,SUB1), and SUB1 may be referring to an array defined in SUB2 (by issuing, for example, MOV TABLE(R0),R1, where TABLE is defined in SUB2 with a .BLKW directive).

A symbol in a program is called *global* if it is: (1) defined in another program, or (2) it is referred to by another program. All global symbols $\text{sym}_1, \text{sym}_2, \dots, \text{sym}_r$ in a subprogram must be declared (anywhere within the program) by a global directive:

.GLOBL sym₁,sym₂,...,sym_r

Nonglobal symbols are said to be *local*. As we have seen, in pass II of the assembly process, all local symbols in a program are replaced by numerical addresses (with or without apostrophes). Thus, by the time the object modules are presented to the linkage editor, local symbols cease to exist. For this reason, local symbols in one program may be duplicated in another program without risking confusion. Global symbols, however, must be uniquely defined in all linked programs, since they still exist “unresolved” (i.e., with no numerical equivalent available) when the linkage editor takes over.

As an example, Figures 9.4, 9.5, and 9.6 show the assembler listings (including symbol tables) of programs MAIN, SUB1, and SUB2 (which are nonsense programs serving only for illustration). Each one of these

```

1          .TITLE  MAIN
2          000000'LC=.
3          000004'=.+4+LC
4 000004 000006 .WORD  6,0,12,0          ;INITIALIZE ERROR VECTORS
          000000
          000012
          000000
5          000500'=.+500+LC          ;ALLOW FOR STACK SPACE
6 000500 010706 START: MOV    PC,SP
7 000502 005746          TST    -(SP)          ;INITIALIZE SP TO START
8          .GLOBL  SUB1,A,B,D,E,G
9 000504 004767          JSR    PC,SUB1
          000000G
10 00510 005767          TST    A
          000026
11 00514 005767          TST    B
          000142
12 00520 005767          TST    C
          000256
13 00524 005767          TST    D
          000000G
14 00530 005767          TST    E
          000000G
15 00534 005767          TST    G
          000000G
16 00540 000000          HALT
17 00542          A:   .BLKW  50
18 00662          B:   .BLKW  50
19 01002          C:   .BLKW  50
20          000500' .END  START

```

```

SYMBOL TABLE
A      = 000542RG      B      = 000662RG      C      = 001002R
D      = ***** G    E      = ***** G    G      = ***** G
LC     = 000000R      START = 000500R      SUB1  = ***** G

```

Figure 9.4 Assembler listing of MAIN.

```

1          .TITLE  SUB1
2          .GLOBL  SUB1,SUB2,A,B,D,E
3 000000 004767 SUB1: JSR   PC,SUB2
              000000G
4 000004 005767      TST   A
              000000G
5 000010 005767      TST   B
              000000G
6 000014 005767      TST   D
              000006
7 000020 005767      TST   F
              000402
8 000024 000207      RTS   PC
9 000026          D:   .BLKW 100
10 00226          E:   .BLKW 100
11 00426          F:   .BLKW 100
12          000001      .END

```

SYMBOL TABLE

A	=	***** G	B	=	***** G	D	000026RG
E		000226RG	F		000426R	SUB1	000000RG
SUB2	=	***** G					

Figure 9.5 Assembler listing of SUB1.

```

1          .TITLE  SUB2
2          .GLOBL  SUB2,A,D,G
3 000000 005767 SUB2: TST   A
              000000G
4 000004 005767      TST   D
              000000G
5 000010 005767      TST   G
              000006
6 000014 005767      TST   H
              000202
7 000020 000207      RTS   PC
8 000022          G:   .BLKW 100
9 000222          H:   .BLKW 100
10          000001      .END

```

SYMBOL TABLE

A	=	***** G	D	=	***** G	G	000022RG
H		000222R	SUB2		000000RG		

Figure 9.6 Assembler listing of SUB2.

programs contains global symbols. For example, MAIN refers to the global symbols SUB1, D, and E, which appear in subroutine SUB1, and to G, which appears in subroutine SUB2. In addition, MAIN defines the symbols A and B which are referred to by the subroutines SUB1 and SUB2. Hence, MAIN must contain the directive

```
.GLOBL SUB1,A,B,D,E,G
```

In the listing of the object code, global symbols defined externally are shown as 000000G; however, in the object module they actually appear symbolically (as character strings). In the symbol table, all global symbols are suffixed with G; externally defined global symbols appear in the table as ***** G.

In the example above, the symbol C is local to MAIN, F is local to SUB1, and H is local to SUB2. (Both F and H could have been named “C” without any risk.)

9.7

THE TWO-PASS LINKAGE PROCESS

Each of the object modules presented to the linkage editor consists of the machine language translation of the corresponding source program. This translation, however, is deficient in two respects: (1) some of the addresses (those suffixed with apostrophes) may need modification as a result of relocation, and (2) some of the addresses (those which still appear in symbolic form) are global and require resolution.

In addition to the code, each object module provides the linkage editor with a *global symbol table*, which consists of all global symbol definitions found within the module. These correspond to all symbols in the original symbol table which are suffixed with G and are not “defined” as *****. For example, for the three programs of Section 9.6, we have

Global symbol table	(A	000542R
for MAIN	B	000662R
Global symbol table	D	000026R
for SUB1	E	000226R
	SUB1	000000R
Global symbol table	G	000022R
for SUB2	SUB2	000000R

Each object module also transmits to the linkage editor the module’s length.

The linkage editor's job of converting the object modules into a load module is accomplished in two "passes," in roughly the following manner:

Pass I: Using the information on the lengths of the object modules and on the order in which the object modules are to appear in the memory, the linkage editor constructs a *load map* which shows how the modules are to be relocated in the memory. Using the load map, the linkage editor now scans each of the object modules and modifies (whenever necessary) each apostrophed address according to the rules given in Section 9.5. It also modifies (whenever necessary) the addresses in each of the global symbol tables so as to make them conform with the loading origins of the corresponding modules. It then combines all the global symbol tables into a single *merged global symbol table* which lists all the global symbols used in the programs and their correct values.

Figure 9.7 shows the listing of the load map produced by the linkage editor for the programs of Section 9.6. This listing shows the start address of the program (500) and its total length (2372), as well as the loading origins (0, 1122, 1750) and the lengths (1122, 626, 422) of the three modules. Figure 9.8 shows the same information in a diagrammatical form. The information in this load map is used by the linkage editor to modify all apostrophed addresses appearing in the object modules of MAIN, SUB1, and SUB2 and to construct the merged global symbol table. The table (which is shown interspersed between the lines of the load map in Figure 9.7) looks as follows:

Merged global symbol table	}	A	000542
		B	000662
		D	001150
		E	001350
		SUB1	001122
		G	001772
		SUB2	001750

For example, the address D, which in the global symbol table of SUB1 was 26, is now $1122 + 26 = 1150$; the address G, which in the global symbol table of SUB2 was 22, is now $1750 + 22 = 1772$. [Remember, however, that the values of *absolute* global symbols (i.e., those without the suffix R) remain unchanged.]

Pass II: Pass I produced an almost-complete load module, except for the global symbols, which are still embedded in the code as character strings. The linkage editor now scans the object modules and, using the merged global symbol table, replaces all global symbols with their values as listed

```

TRANSFER ADDRESS: 000500
LOW LIMIT:        000000
HIGH LIMIT:       002372
*****
MODULE MAIN
SECTION          ADDRESS SIZE
<. ABS.>         000000 000000
< >             000000 001122
                000542' B      000662'
*****
MODULE SUB1
SECTION          ADDRESS SIZE
<. ABS.>         000000 000000
< >             001122 000626
                001150' E      001350'      SUB1  001122'
*****
MODULE SUB2
SECTION          ADDRESS SIZE
<. ABS.>         000000 000000
< >             001750 000422
                001772' G      001750'

```

Figure 9.7 Load map.

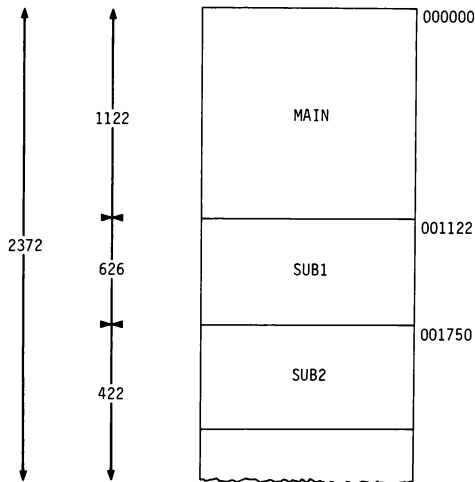


Figure 9.8 Load module.

in the table. Here the merged global symbol table is used in exactly the same manner as the symbol table was used by the assembler in pass II of the assembly process. No address, however, need to be “apostrophed” since no further relocation takes place.

From previous examples we can see that, in a typical program, most addresses need not be modified by the linkage editor after assembly, no matter where the loading origin is. A source code whose addresses (after assembly) need no modification regardless of loading origin is called *position-independent code* (PIC). It is sometimes preferable to write programs—especially utility routines intended for frequent use by many users—in PIC, since this results in object codes ready to be loaded anywhere in the memory without additional processing.

To decide what is and what is not permitted in PIC, we can refer to Section 9.5, where we spelled out the conditions under which addresses may require modification by the linkage editor (i.e., the conditions under which MACRO-11 attached apostrophes to addresses). Using these conditions, we can deduce that, in PIC:

1. Relocatable addresses must be used in relative mode only.
2. Absolute addresses can be used in any mode *except* the relative mode.

These rules make the writing of programs in PIC a tricky business. For example, we have already seen that the non-PIC

```
START: MOV #START,SP
```

can be replaced by the PIC

```
START: MOV PC,SP
       TST -(SP)
```

As another example, consider the copying of the 100-word array based at A into the 100-word array based at B:

```

                                MOV    #100,R0
                                MOV    #A,R1
                                MOV    #B,R2
LOOP:  MOV    (R1)+,(R2)+
                                DEC    R0
                                BNE    LOOP
                                - - -
A:     .BLKW 100
B:     .BLKW 100
```

The above non-PIC can be replaced with the following PIC:

```

MOV    #100,R0
MOV    PC,R1
ADD    #A-,R1 ← α
MOV    PC,R2
ADD    #B-,R2 ← β
LOOP:  MOV    (R1)+,(R2)+
        DEC    RO
        BNE   LOOP
        ...
A:     .BLKW 100
B:     .BLKW 100

```

During execution, `MOV PC,R1` puts in `R1` the (absolute) address α of the next instruction; the next instruction, `ADD #A-,R1`, adds to `R1` the distance d_1 from α to `A` and hence results in `R1` holding the absolute address `A`. Similarly, `MOV PC,R2` and `ADD #B-,R2` result in `R2` holding the absolute address `B`. (Note that `A-` and `B-`, being of the form relocatable - relocatable, are absolute addresses.)

EXERCISES

- 9.1 Indicate which of the following instructions are erroneous. Which errors are assembly-time errors and which are run-time errors?

```

NEG    @#37775
MOVE   RO,R5
RTI    ;SUB R4,R1
JMP    R3
BIC    (PC),SP
ADD    R2+5,R3
CLR    TABLE (RO)+

```

- 9.2 The programs `PROG1` and `PROG2` shown below are assembled independently.

- List the object module of `PROG1`.
- List the load module resulting from linking `PROG1` and `PROG2` (in that order).

```

                .TITLE PROG1
LC=:
.=476+LC
START:  HALT
        .END START

                .TITLE PROG2
X:      MOV    #X,Y-X
        MOV    #X-Y,Y
        MOV    @#X-5,Y-X(R5)
Y:      HALT
        .WORD X,X-Y+3
        .END

```

- 9.3 (a) List the object module and the symbol table of the following program. Indicate in the symbol table which addresses are absolute and which are relative.

```

LC=.
.=500+LC
A = 12
B = F+'h
C = E-F
D = A+C-2
START:    MOV A,B
E:        MOV #C, E(R0)
F:        MOV #A+B,@0(R1)
          HALT
G:        .BLKW D-C
          .WORD G+C,G-B
          .END  START

```

- (b) Suppose that the program is loaded at loading origin 2200. List the program as modified by the linkage editor.
- 9.4 Shown below are three programs, MAIN, SUB1, and SUB2, which are assembled separately and then linked (in that order).
- (a) Complete the .GLOBL declarations on the three programs.
- (b) Show the complete assembler listing (including the symbol table, the global table, and all apostrophes and suffixes) for each program.
- (c) Construct the load map and the merged global symbol table for the three programs.
- (d) List the load module of the linked program.

```

          .TITLE  MAIN
          .GLOBL  ---
LC=.
.=500+LC
START:    MOV      #400,100(R1)
          MOV      A,B(R0)
          MOV      #L,150
          MOV      X,@#60
          MOV      K,@B
          JSR      PC,SUB1
          HALT
A:        .WORD    C
B:        .BLKB   30
          .END     START

          .TITLE  SUB1
          .GLOBL  ---
K=12
SUB1:    MOV      @45(R1),L
          MOV      #K,B(R3)
          MOV      #B,76
          MOV      X,@#M

```

```

                JSR      PC,SUB2
                MOV      177700,B
                RTS      PC
                .WORD    X
L:              .BLKW    K
M:              .WORD    K-20
                .END

                .TITLE   SUB2
                .GLOBL   ---
SUB2:          MOV      #12,L
                MOV      X,@A(R5)
                MOV      300,@#Y
                BPL      Z
                JMP      L(R1)
Z:             MOV      44(R1),1234
                RTS      PC
X:             .WORD    Y
                .BLKW    X-Z
Y:             .BLKB    16
                .END

```

9.5 Replace the following program segment with a PIC that takes precisely the same action. (Don't use any registers other than R3, R4, and PC!)

```

CLR      100
MOV      #K,R3
MOV      L(R3),R4

```

9.6 Write a PIC program which clears the 100_{10} words that precede the program's starting location.

ADVANCED ASSEMBLY LANGUAGE TECHNIQUES

10

In this section we describe some assembler facilities that make the writing of assembly language programs easier and “cleaner.” In particular, we discuss macros, repeat directives, and conditional assembly.

10.1 MACROS

In chapter 6 we enumerated the advantages of writing modular programs and saw how subroutines can be employed to implement such programs. We must remember, however, that the use of subroutines exacts a certain overhead cost — the time required to transmit arguments, to save and restore registers, and to execute the JSR and RTS instructions. It is not difficult to conceive of subroutines where the time spent on overhead is

comparable to or even exceeds the time required for executing the subroutine proper.

Consider, for example, the subroutine `DIV8`, which divides the contents of `R0` by 8:

```
DIV8: ASR R0 ;SHIFT (R0)
      ASR R0 ; ARITHMETICALLY
      ASR R0 ; 3 BITS TO THE RIGHT
      RTS PC :EXIT
```

To divide the contents of `R3` by 8, we can call `DIV8` as follows:

```
MOV R0,-(SP) ;SAVE (R0)
MOV R3,R0 ;TRANSMIT DIV8 ARGUMENT
JSR PC,DIV8 ;CALL DIV8
MOV R0,R3 ;(R3) = (R3)/8
MOV (SP)+,R0 ;RESTORE (R0)
```

We thus see that six overhead instructions are required to execute a subroutine which consists of only three instructions. As far as execution time is concerned, we are much better off ignoring the subroutine and simply writing

```
ASR R3 ;SHIFT (R3)
ASR R3 ; ARITHMETICALLY
ASR R3 ; 3 BITS TO THE RIGHT
```

But suppose that we are dealing with a large program where division of various registers (`R3` as well as others) by 8 occurs at dozens of different places. Writing the three `ASR` instructions (plus comments) in all these places can get pretty monotonous and time consuming. Here is where the macro facility comes in handy. We can define the three-instruction sequence as a “macro” called `DIV8`, in which the operand register is included as an argument `REG`:

```
.MACRO DIV8 REG
ASR REG ;SHIFT (REG)
ASR REG ; ARITHMETICALLY
ASR REG ; 3 BITS TO THE RIGHT
.ENDM
```

Once this is done, “macro calls” such as


```

DIV8 R3
...
DIV8 R0
...
DIV8 R2

```

can be issued. During pass I of the assembly process the assembler “expands” these calls into the three instructions contained in the definition of the macro DIV8, replacing REG with the particular operand appearing with the call:

```

ASR R3 ;SHIFT (R3)
ASR R3 ; ARITHMETICALLY
ASR R3 ; 3 BITS TO THE RIGHT
...
ASR R0 ;SHIFT (R0)
ASR R0 ; ARITHMETICALLY
ASR R0 ; 3 BITS TO THE RIGHT
...
ASR R2 ;SHIFT (R2)
ASR R2 ; ARITHMETICALLY
ASR R2 ; 3 BITS TO THE RIGHT

```

Once pass I of the assembly is completed, the macro definition for DIV8 is no longer needed and can be eliminated.

We can see that, besides saving the programmer time and effort, the macro facility makes the program more readable: the “instruction” DIV8 R3 is much more explicit than the three ASR instructions. In fact, we can see how, by means of macros, a programmer can invent a whole new instruction set so as to make the program look as if it were written in a high-level language, with all the concomitant advantages.

Figure 10.1 shows schematically how the macro mechanism compares with the subroutine mechanism. It is seen that, while a subroutine appears in the memory only once, the corresponding macro is duplicated as many times as it is called. Thus, if the overhead is relatively low, the macro mechanism may result in a source program that may require much more space than is the case with subroutine mechanism. This is an important point to bear in mind when memory space is at a premium.*

(In some assemblers, including MACRO-11, the macro expansion is actually done in the object code during pass II rather than in the source code during pass I. The macro definitions, however, must be consulted during pass I in order to build up the symbol table. This alternative scheme results in more economical memory-space utilization.)

*Macros are sometimes called “open subroutines.” However, this name should not lead the reader to confuse macros with ordinary (or “closed”) subroutines.

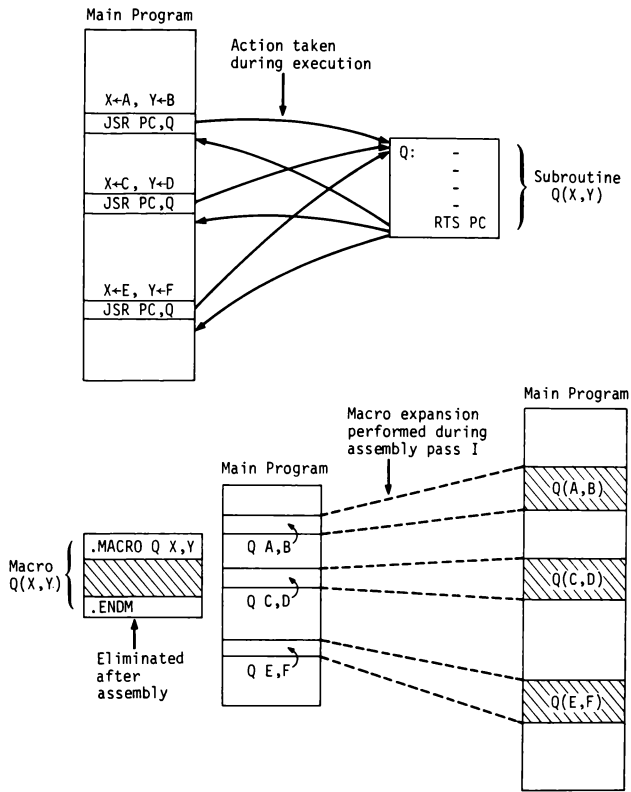


Figure 10.1 Comparison of subroutines and macros.

10.2 MACRO DEFINITIONS AND MACRO CALLS

The general format of a *macro definition* is

```

.MACRO name d1,d2,... ,dr  ← macro heading*
    }
macro body
.ENDM                       ← terminator

```

*A "name" can be followed by a comma; the d_i can be separated by spaces instead of commas.

where d_1, d_2, \dots, d_r are the *dummy arguments*. The general format of a *macro call* is

Label: name a_1, a_2, \dots, a_r

where a_1, a_2, \dots, a_r are the *actual arguments*. The macro called “name” must be defined *prior* to its first call. The call is expanded into (i.e., substituted by) the macro body, with every d_1 replaced by a_1 , every d_2 by a_2, \dots , and every d_r by a_r . After the expansions are performed, the definition serves no useful purpose and can be eliminated.

If more arguments appear in the macro call than in the macro definition, the excess arguments are ignored. If fewer arguments appear in the call than in the definition, the missing arguments are to be null (consisting of no characters). The number of actual arguments passed to a macro by a macro call can be obtained with the directive

.NARG symb

When placed within a macro definition, this directive assigns to “symb” a value that equals the number of actual arguments appearing in the call.

Examples

1. The macro DPADD performs double-precision addition (A) to (B) (see Section 7.3).

<i>Definition</i>	<pre>.MACRO DPADD A,B ADD A,B ADC A+2 ADD A+2,B+2 ENDM</pre>
<i>Call</i>	<pre>DPADD ZZ,SUM</pre>
<i>Expansion</i>	<pre>ADD ZZ,SUM ADC ZZ+2 ADD ZZ+2,SUM+2</pre>

Note: The call assumes that the operands are in ZZ, ZZ + 2 and in SUM SUM + 2.

2. The macro SWAP interchanges (P) and (Q).

<i>Definition</i>	<pre>.MACRO SWAP P,Q MOV P,TEMP MOV Q,P MOV TEMP,Q .ENDM</pre>
<i>Call</i>	<pre>NOW: SWAP F+6,(R3)</pre>

Expansion

```

NOW:  MOV    F+6,TEMP
      MOV    (R3),F+6
      MOV    TEMP,(R3)

```

Note: The label of a macro call is used as the label of the first instruction in the expansion.

3. The macro **FLIP** reverses the order of the contents in **P1, P2, P3, P4**. (**SWAP** is assumed to be defined as in Example 2.)

Definition

```

.MACRO FLIP P1,P2,P3,P4
SWAP   P1,P4
SWAP   P2,P3
.ENDM

```

Call

```

FLIP   B,B+2,B+4,B+6

```

Expansion

```

(a) SWAP   B,B+6
    SWAP   B+2,B+4

(b) MOV    B,TEMP
    MOV    B+6,B
    MOV    TEMP,B+6
    MOV    B+2,TEMP
    MOV    B+4,B+2
    MOV    TEMP,B+4

```

Note: Macros can be “nested” to any number of levels, in which case the expansion is performed in a number of successive stages. Nested macros can be defined in any order, as long as the call to the outer one occurs after the definition of the inner one.

4. The macro **PRINT** prints a single character.

Definition

```

.MACRO PRINT CHAR
TSTB   177564
BPL    .-4
MOV    #CHAR,177566
.ENDM

```

Call

```

PRINT 'Y

```

Expansion

```

TSTB   177564
BPL    .-4
MOV    #'Y,177566

```

5. The macro **NULINE** moves the teletype to a new line. (**PRINT** is assumed to be defined as in Example 4.)

Definition

```

.MACRO NULINE
PRINT  15
PRINT  12
.ENDM

```

Call

```

NULINE

```

Expansion

```

(a) PRINT 15
    PRINT 12

```

```

(b) TSTB 177564
    BPL  .-4
    MOV  #15,177566
    TSTB 177564
    BPL  .-4
    MOV  #12,177566

```

Note: There are macros with no arguments at all.

6. The macro WAIT executes an instruction and skips over data.

```

Definition      .MACRO  WAIT INSTR,SIZE,DATA,LOC
                  INSTR
                  JMP     LOC+SIZE+2
                  .BYTE  DATA
LOC:              .BLKB  SIZE
                  .ENDM

Call            WAIT    <JSR PC,SUB3>,100.,<5,22,'H','L>,MATRIX

Expansion      JSR    PC,SUB3
                  JMP    MATRIX+100.+2
                  .BYTE  5,22,'H','L'
MATRIX:          .BLKB  100.

```

Note: Enclose an actual argument in < > if it contains commas and/or spaces. WAIT is also a legal PDP-11 instruction, but the macro definition takes precedence.

7. The macro DATA stores up to 10 items in an array LIST, followed by as many reserved words as there are stored items.

```

Definition      .MACRO  DATA LIST,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10
                  .NARG  N
LIST:             .WORD  P1,P2,P3,P4,P5,P6,P7,P8,P9,P10
                  .BLKW  N-1
                  .ENDM

Call            DATA  KIT,17,1098.,'*

Expansion      .NARG  N
KIT:             .WORD  17,1098.,'*,,,,,,
                  .BLKW  3

```

Note: N is assigned the value 4, which is the number of actual arguments passed to DATA. Blank data items in .WORD result in storage of 0's.

8. The macro ROTATE rotates a register, moves it to memory, and branches.

```

Definition      .MACRO  ROTATE D,REG,BASE,K,COND,DEST
                  RO'D  REG
                  MOV   REG,BASE'K
                  B'COND DEST
                  .ENDM

Call            ROTATE L,R2,SWITCH,+30,EQ,OUT

```

```

Expansion
          ...
          ROTATE R,R5,MAT,-16,R,+.12
          ROL    R2
          MOV    R2,SWITCH+30
          BEQ   OUT
          ...
          ROR   R5
          MOV   R5,MAT-16
          BR   .+12

```

Note: Use ' to separate adjacent symbols, to avoid ambiguity. (For example, without ' the definition of ROTATE will include the symbols ROD, BASEK, and BCOND.) In the expansion process the ' is deleted and the adjacent symbols become "concatenated."

9. Macro JOE stores a string "MESSAGE NO. k" in address Xk.

Definition

```

          .MACRO JOE A,J
          MARY A,\J
J=J+1
          .ENDM

          .MACRO MARY X,K
X'K: .ASCII /MESSAGE NO. K/
          .ENDM

```

Call

```

I=0
          JOE TEXT,I
          ...
          JOE TEXT,I
          ...
          JOE TEXT,I

```

Expansion

```

(a) I=0
          MARY TEXT,\I
I=I+1
          ...
          MARY TEXT,\I
I=I+1
          ...
          MARY TEXT,\I
I=I+1

(b) I=0
TEXT0: .ASCII /MESSAGE NO. 0/
I=I+1
          ...
TEXT1: .ASCII /MESSAGE NO. 1/
I=I+1
          ...
TEXT2: .ASCII /MESSAGE NO. 2/
I=I+2

```

Note: \I indicates that, in the expansion, the integer I should be inserted in character form. (For example, 2 in TEXT2 is the *character* 2.)

One of the basic rules in writing assembly language programs is to ensure that no label be multiply defined, that is, that no label appear more than once in the label field. However, there is an exception to this rule: Labels having the form $n\$$, where n is a decimal integer between 1 and 65535, can be repeated as long as they are separated by at least one “ordinary” label (i.e., a label not of the form $n\$$). For example:

```

LABEL1: ...
3$: ...
15$: ...
18$: ...
LABEL2: ...
15$: ...
18$: ...
22$: ...
LABEL3: ...
3$: ...
22$: ...

```

is perfectly legal. The assembler considers the $n\$$ labels as *local* to the region bounded by the ordinary labels, and creates unique definitions for them in the symbol table.

Returning to macros, let us consider the following macro `MULT` which computes $C \leftarrow (A) * (B)$ (see Section 6.5):

```

.MACRO MULT A,B,C
CLR C
LOOP: DEC B
      BMI EXIT
      ADD A,C
      BR LOOP
EXIT: .ENDM

```

Suppose that we call `MULT` twice:

```

MULT R1,R2,R3
...
MULT P,Q,R

```

(1)

After the expansion, we have*

```

                CLR  R3
LOOP:          DEC  R2
                BMI  EXIT
                ADD  R1,R3
                BR   LOOP
EXIT:
                ...
                CLR  R
LOOP:          DEC  Q
                BMI  EXIT
                ADD  P,R
                BR   LOOP
EXIT:

```

which, of course, is illegal since LOOP and EXIT are multiply defined. To avoid this problem we could, to be sure, list LOOP and EXIT as parameters of MULT:

```

.MACRO  MULT A,B,C,LOOP,EXIT
...
.ENDM

```

and instead of (1), issue the calls

```

MULT  R1,R2,R3,LOOP1,EXIT1
...
MULT  P,Q,R,LOOP2,EXIT2

```

(2)

However, if neither LOOP nor EXIT is referred to outside the macro, specifying them in the call is not really necessary. One can declare LOOP and EXIT as *local symbols* (local to the macro) by listing them in the macro's list of parameters as follows:

```

.MACRO  MULT A,B,C,?LOOP,?EXIT
...
.ENDM

```

(3)

If the call makes no mention of LOOP or EXIT (i.e., the corresponding actual parameters are absent or null), the assembler replaces them with 64\$ and 65\$ in the first expansion, with 66\$ and 67\$ in the second ex-

*A label such as EXIT: can appear in a line preceding that which it labels. In fact, any number of distinct labels are permitted to appear (one beneath the other), all labeling the same word (and hence having the same value in the symbol table).

pansion, and so on. Thus, if `MULT` is defined as in (1) and called as in (3), the expansion is

```

        CLR  R3
64$:   DEC  R2
        BMI  65$
        ADD  R1,R3
        BR   64$

65$:   ...
        CLR  R
66$:   DEC  Q
        BMI  67$
        ADD  P,R
        BR   66$

67$:

```

If the two calls are separated by an “ordinary” label, the labels generated in the second expansion are the same as in the first, namely `64$` and `65$`. As explained above, this is quite legal and will not result in multiple definition of `64$` and `65$`.

In general, local symbols are designated in the list of parameters in the macro definition with the prefix `?`. If they are absent from the call, they are replaced in the expansion by `64$`, `65$`, . . . , `127$`. If they are specified in the call [as in (2)], the expansion is done as specified, in the normal manner. The labels `n$` will be repeated if the calls are separated by ordinary labels.

10.4 REPEAT DIRECTIVES

Occasionally, an assembly language program contains successive repetitions of identical or almost identical copies of the same code sequence. In this case a great deal of effort can be saved by using the `.REPT` (“repeat”) directive:

```

.REPT exp  ← heading
           }  repeat block
.ENDM     ← terminator*

```

During pass I of the assembly process, the assembler duplicates the repeat block as many times as specified by the value of “exp.”

* `.ENDR` can also be used here.

Examples

1. Leave four blank lines. (Use the macro NULINE of Example 5 in Section 10.2.)

```
.REPT 5
NULINE
.ENDM
```

2. Set up a 100₁₀-word array A, with each word containing the address of the next word, except that the last word contains the address of the first. (This array is called a “circular list.”)

```
A:
.REPT 99.
.WORD .+2
.ENDM
.WORD A
```

3. Fill an array TABLE with the ASCII code of the characters A to Z.

```
CHAR='A
TABLE:
.REPT 26.
.BYTE CHAR
CHAR=CHAR+1
.ENDM
```

4. Push the contents of TAB, TAB + 1, TAB + 2, . . . , TAB + 16 onto the system stack, using the macro PUSH. (See Example 9 in Section 10.2 for the explanation of \.)

```
.MACRO PUSH K
MOVB TAB+K,-(SP)
.ENDM
I=0
.REPT 17
PUSH \I
I=I+1.
.ENDM
```

5. The macro SAVE pushes the contents of TAB + I, TAB + I + 1, TAB + I + 2, . . . , TAB + J (J ≥ I) onto the system stack, using the macro PUSH of Example 4.

Definition

```
.MACRO SAVE I,J
COUNT=I
.REPT J-I+1
```

```

                PUSH  \COUNT
COUNT=COUNT+1
                .ENDM
                .ENDM

Call           SAVE   12,22

Expansion (a) COUNT=12
                PUSH  \12
COUNT=13
                PUSH  \13
                .
                .
COUNT=22
                PUSH  \22
                }
                22 - 12 + 1 = 11
                macro calls

(b) COUNT=12
                MOVB  TAB+12,-(SP)
COUNT=13
                MOVB  TAB+13,-(SP)
                .
                .
COUNT=22
                MOVB  TAB+22,-(SP)
    
```

□

Another useful repeat directive is the `.IRP` (“indefinite repeat”) directive:

```

.IRP d,<a1,a2, . . . ,ar>  ← heading
    }                       ← repeat block
.ENDM                       ← terminator
    
```

where `d` is a dummy argument and `a1, a2, . . . , ar` are actual arguments. The assembler (during pass I) duplicates the repeat block `r` times, first with `d` replaced with `a1`, next with `d` replaced with `a2`, and so forth. For example, to restore R0, R3, R4, and R5 from the system stack, we can write

```

.IRP REG,<R0,R3,R4,R5>
MOV  (SP)+,REG
.ENDM
    
```

A similar directive is

```

.IRPC d,str  ← heading
    }        ← repeat block
.ENDM       ← terminator
    
```

where `d` is a dummy argument and “`str`” is a string of characters. The assembler duplicates the repeat block—first with `d` replaced by the first character of “`str`,” next with `d` replaced by the second character of “`str`,”

and so on. For example, to restore R0, R3, R4, and R5 from the system stack, we can write

```
.IRPC N,0345
MOV (SP)+,R'N
.ENDM
```

(As in macros, ' serves to separate adjacent symbols but is deleted during duplication.)

10.5 CONDITIONAL ASSEMBLY

Conditional assembly directives enable the programmer to either include or exclude a segment of source code, depending on certain conditions. This facility is frequently used inside macro definitions, where parameter values (as defined during pass I of the assembly process) determine which version of the macro should be expanded.

The general form of the conditional directive is

```
.IF cond,s    ← heading
               } conditional block
.ENDC         ← terminator
```

where “cond” specifies a condition that *s* may or may not satisfy. If *s* does satisfy the condition, the conditional block is assembled; otherwise, it is ignored. Table 10.1 lists some of the allowable IF directives. If the directive

```
.IFF
```

TABLE 10.1 Conditional Directives

Directive	Assemble Block If
.IF EQ, <i>s</i>	$s = 0$
.IF NE, <i>s</i>	$s \neq 0$
.IF GT, <i>s</i>	$s > 0$
.IF LE, <i>s</i>	$s \leq 0$
.IF LT, <i>s</i>	$s < 0$
.IF GE, <i>s</i>	$s \geq 0$
.IF DF, <i>s</i>	<i>s</i> is defined
.IF NDF, <i>s</i>	<i>s</i> is not defined
.IF B,< <i>s</i> >	macro argument <i>s</i> is blank (absent)
.IF NB,< <i>s</i> >	macro argument <i>s</i> is not blank (present)

appears inside the conditional block, then the part of the block lying below the IFF is assembled only if the preceding IF condition is not satisfied. Thus, the IFF directive partitions the block into two subblocks, only one of which is actually assembled.

Examples

1. The macro BRANCH generates the instruction BR X if the relative distance to X is less than 255₁₀ bytes, and the instruction JMP X otherwise. X must be defined when this macro is called (that is, the branch must be a backward branch).

<i>Definition</i>	.MACRO BRANCH X .IF LT,,-X-255. BR X .IFF JMP X .ENDC .ENDM
-------------------	---

Note: Here we have an IF nested within a macro. The IFF splits the IF block into two mutually exclusive subblocks (BR X and JMP X).

2. The macro GOTO L,X,REL,Y (where REL can be EQ, NE, GT, etc.) generates the instructions CMP X,Y and B'REL L (which during run time cause a branch to L if X and Y are related by REL). A call of the form GOTO L generates the unconditional branch BR L. (This is an attempt to introduce a FORTRAN-like instruction into assembly language.)

<i>Definition</i>	.MACRO GOTO L,X,REL,Y .IF B,<REL> BR L .IFF CMP X,Y B'REL L .ENDC .ENDM
-------------------	--

<i>Call</i>	GOTO LOOP,SUM,NE,#15 ... GOTO EXIT
-------------	--

<i>Expansion</i>	CMP SUM,#15 BNE LOOP ... BR EXIT
------------------	---

3. The macro MAX puts in R0 the maximum of one to three arguments. The code generated depends on the number of arguments, which is determined by the NARG directive.

Definition

```

.MACRO MAX A,B,C,?NEXT,?OUT
.NARG K
MOV A,RO
.IF NE,K-1
.IF NE,K-2
CMP C,RO
BLE NEXT
MOV C,RO
.ENDC
NEXT: CMP B,RO
      BLE OUT
      MOV B,RO
      .ENDC
OUT:
      .ENDM

```

Call

```

MAX P
...
MAX P,Q
...
MAX P,Q,R

```

Expansion

```

(K=1) MOV P,RO
65$:
...
(K=2) MOV P,RO
      CMP Q,RO
      BLE 67$
      MOV Q,RO
67$:
...
(K=3) MOV P,RO
      CMP R,RO
      BLE 68$
      MOV R,RO
68$: CMP Q,RO
      BLE 69$
      MOV Q,RO
69$:

```

4. Conditional assembly can be used to implement “macro recursion.” The macro POWER shown below multiplies the contents of X by 2^N by shifting X left (arithmetically) N times. The macro actually calls itself recursively, with the recursion terminating by virtue of the IF which keeps comparing a running tally (COUNT) against N.

Definition

```

.MACRO POWER X,N
ASL X
COUNT=COUNT+1
.IF NE,COUNT-N
POWER X,N
.ENDC
.ENDM

```

Call

```

COUNT=0
POWER R5,6

```

Expansion

```

ASL   R5
ASL   R5
ASL   R5
ASL   R5
ASL   R5
ASL   R5

```

EXERCISES

10.1 The macro FUN is defined as follows:

```

LOOP: .MACRO FUN A,B,C,X,Y,N,M,?LOOP
      X*Y
      A*B      (PC)
      BV'C    LOOP
      .WORD   N,"B*M
      .ENDM

```

Expand the following four calls, which appear consecutively in a program. Show the contents of the .WORD directive resulting from each call.

```

FUN   AS,L,C,CL,C,15.,C
FUN   RO,R,S,SE,V,'X,X
FUN   RO,L,S,CL,V,13,< >,L
FUN   AS,R,C,SE,N,<1,2,3>,<,>

```

10.2 The macros MAC1 and MAC2 are defined as follows:

```

      .MACRO MAC1 K,X
      MOVB R'K,X+K
      .ENDM
      .MACRO MAC2 N,Z
I=0   .REPT N
      MAC1 \I,Z
I=I+1
      .ENDM
      .ENDM

```

Explain what MAC2 accomplishes and expand the call

```
MAC2   5,TEMP
```

10.3 (a) The macro STORE is defined by

```

                .MACRO   STORE X,N
                MOV     #I,X+I
I=I+1
                .IF     NE,I-N
                STORE  X,N
                ENDC
                ENDM

```

Expand the call

```

I=0
                STORE  TAB,7

```

(b) Write a *nonrecursive* macro that takes the same action as STORE.

10.4 The macro FOO is defined as follows:

```

                .MACRO   FOO R5,ITEMS,LENGTH,BETA
                CMP     BETA,R0,R1
                R5
                .WORD   ITEMS
                .IF     B,<BETA>
                .BLKW  LENGTH
                .IFF
                .BLKB  LENGTH
                .ENDC
                ENDM

```

Expand the calls

```

L5:            FOO     <BPL L5>,<5,25,3>,100,B
                FOO     <BNE L5>,,52

```

- 10.5 Write a subroutine POWER which multiplies the contents of X by 2^N (where X is found in R0 and N in R1). Compare this subroutine with the macro POWER of Section 10.5, indicating their relative advantages and disadvantages.
- 10.6 Write a macro XOR which, when called with XOR, A,B, puts in R0 the exclusive-or product of A and B. (That is, R0 will contain 1's only in those bits where A and B differ.)
- 10.7 Write a macro JMPSR, headed

```

                .MACRO   JMPSR SUB,REGS,ARRAY

```

which stores the registers listed in REGS onto a block of words starting at ARRAY, performs JSR PC,SUB, and then restores the contents of all these registers. For example, the call

JMPSR DPADD,024,TEMP

stores R0, R2, and R4 in TEMP, TEMP + 2, and TEMP + 4; performs JSR PC,DPADD; and then restores R0, R2, and R4.

- 10.8 A (fictitious) computer called SIMCOM (SIMple COMputer) has one general-purpose register called an *accumulator* (ACC). Its central memory and number representation are identical to those of the PDP-11. The following is its instruction set, written in SIMCOM assembly language (where d is a CM address):

Instruction	Action
LDA d	<i>Load:</i> (ACC) \leftarrow (d)
STO d	<i>Store:</i> (d) \leftarrow (ACC)
ADD d	<i>Add:</i> (ACC) \leftarrow (ACC)+(d)
SUB d	<i>Subtract:</i> (ACC) \leftarrow (ACC)-(d)
TRA d	<i>Transfer:</i> Branch to d (unconditionally)
TRZ d	<i>Transfer on zero:</i> Branch to d if (ACC) = 0
TRN d	<i>Transfer on negative:</i> Transfer to d if (ACC) > 0
HALT	<i>Halt</i>

(I/O instructions are omitted.) The assembler directives in SIMCOM are:

Directive	Action
OCT n	Store the octal constant n
BSS n	Reserve n (octal) words
END	End assembly

Using R0 to simulate ACC, write a collection of PDP-11 macros whose calls coincide with the 11 SIMCOM instructions and directives, and which take the same actions as these instructions and directives. For example, the macro for the load instruction will be

```
.MACRO LDA X
MOV X,R0
.ENDM
```

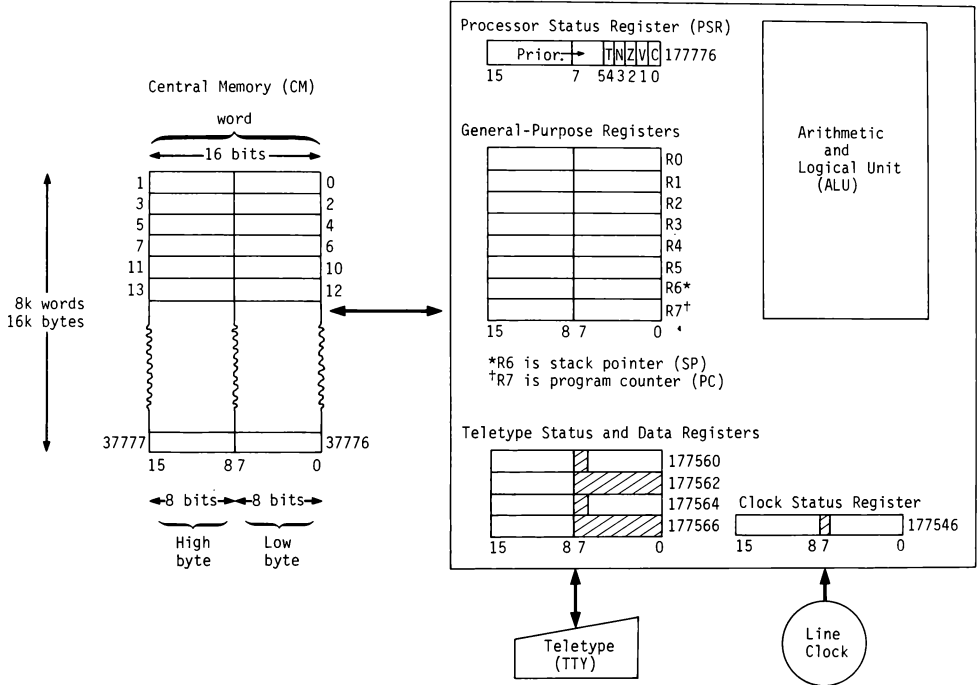
Using these macros, a SIMCOM program can be run on the PDP-11. Each SIMCOM instruction or directive is actually a PDP-11 macro call (e.g., LDA SUM is actually MOV SUM,R0) which results in the simulation of the SIMCOM instruction or directive.

Try your macros with a SIMCOM program which divides (A) by (B) and puts the result (with fraction truncated) in C.

APPENDIXES

Appendix A PDP-11 ORGANIZATION (PARTIAL)

Central Processor (CP)



Appendix B SEVEN-BIT ASCII CODE (PARTIAL)

Character	Code	Character	Code
Bell	007	N	116
Line feed	012	O	117
Carriage return	015	P	120
Space	040	Q	121
!	041	R	122
”	042	S	123
#	043	T	124
\$	044	U	125
%	045	V	126
&	046	W	127
'	047	X	130
(050	Y	131
)	051	Z	132
*	052	[133
+	053	\	134
,	054]	135
-	055	↑	136
.	056	←	137
/	057	`	140
0	060	a	141
1	061	b	142
2	062	c	143
3	063	d	144
4	064	e	145
5	065	f	146
6	066	g	147
7	067	h	150
8	070	i	151
9	071	j	152
:	072	k	153
;	073	l	154
<	074	m	155
=	075	n	156
>	076	o	157
?	077	p	160
@	100	q	161
A	101	r	162
B	102	s	163
C	103	t	164
D	104	u	165
E	105	v	166
F	106	w	167
G	107	x	170
H	110	y	171
I	111	z	172
J	112	{	173
K	113		174
L	114	}	175
M	115	~	176

Appendix C PDP-11 ADDRESSING MODES

Mode	Name of Mode	Assembly Language	Operand's Location	Explanation
0	Register	Rn	Rn	Operand is in Rn.
1	Register deferred	(Rn)	(Rn)	Address of operand is in Rn.
2	Autoincrement	(Rn)+	(Rn)	Address of operand is in Rn; (Rn) \leftarrow (Rn)+2 after operand is fetched.*
3	Autoincrement deferred	@(Rn)+	((Rn))	Address of address of operand is in Rn; (Rn) \leftarrow (Rn)+2 after operand is fetched.
4	Autodecrement	-(Rn)	(Rn)	(Rn) \leftarrow (Rn)-2 before address is computed †; address of operand is in Rn.
5	Autodecrement deferred	@-(Rn)	((Rn))	(Rn) \leftarrow (Rn)-2 before address is computed; address of address of operand is in Rn.
6	Index	X(Rn)	X+(Rn)	Address of operand is X plus (Rn). Address of X is in PC; (PC) \leftarrow (PC)+2 after X is fetched.
7	Index deferred	@X(Rn)	(X+(Rn))	Address of address of operand is X plus (Rn). Address of X is in PC; (PC) \leftarrow (PC)+2 after X is fetched.
<u>PC addressing</u>				
2	Immediate	#k		Operand is k. (k follows instruction.)
3	Absolute	@#A	A	Address of operand is A. (A follows instruction.)
6	Relative	A	A	Address of operand is A. [A-(PC) follows instruction.]
7	Relative deferred	@A	(A)	Address of address of operand is A. [A-(PC) follows instruction.]

*But (Rn) \leftarrow (Rn)+1 if instruction is byte instruction and n < 6.
†But (Rn) \leftarrow (Rn)-1 if instruction is byte instruction and n < 6.

Appendix D

PDP-11 INSTRUCTIONS (PARTIAL LIST)

Machine Language	Assembly Language	Name of Instruction	Resulting Action
<u>Single Operand</u>			
0050DD	CLR d	clear	$(d) \leftarrow 0$
0051DD	COM d	complement	$(d) \leftarrow \sim(d)$
0052DD	INC d	increment	$(d) \leftarrow (d) + 1$
0053DD	DEC d	decrement	$(d) \leftarrow (d) - 1$
0054DD	NEG d	negate	$(d) \leftarrow -(d)$
0057DD	TST d	test	$(d) \leftarrow (d)$
0060DD	ROR d	rotate right	$(d) \leftarrow (d)$ shifted right 1 bit
0061DD	ROL d	rotate left	$(d) \leftarrow (d)$ shifted left 1 bit
0062DD	ASR d	arith. shift right	$(d) \leftarrow (d)/2$
0063DD	ASL d	arith. shift left	$(d) \leftarrow 2 * (d)$
0003DD	SWAB d	swap bytes	$(d)_{low} \leftrightarrow (d)_{high}$
0055DD	ADC d	add carry	$(d) \leftarrow (d) + C$
0056DD	SBC d	subtract carry	$(d) \leftarrow (d) - C$
0001DD	JMP d	jump	$(PC) \leftarrow d$
<u>Double Operand</u>			
0155DD	MOV s,d	move	$(d) \leftarrow (s)$
0255DD	CMP s,d	compare	form $(s) - (d)$
0655DD	ADD s,d	add	$(d) \leftarrow (s) + (d)$
1655DD	SUB s,d	subtract	$(d) \leftarrow (d) - (s)$
0355DD	BIT s,d	bit test	form $(s) \wedge (d) *$
0455DD	BIC s,d	bit clear	$(d) \leftarrow [\sim(s)] \wedge (d) *$
0555DD	BIS s,d	bit set	$(d) \leftarrow (s) \vee (d) *$

* \vee is "OR," \wedge is "AND," \sim is "NOT."

Base Code	Assembly Language	Branch to a if*	
000400	BR a	(unconditionally)	
001000	BNE a	not equal to 0 ($Z = 0$)	
001400	BEQ a	equal to 0 ($Z = 1$)	
100000	BPL a	plus ($N = 0$)	
100400	BMI a	minus ($N = 1$)	
102000	BVC a	overflow clear ($V = 0$)	
102400	BVS a	overflow set ($V = 1$)	
103000	BCC a	carry clear ($C = 0$)	
103400	BCS a	carry set ($C = 1$)	
002000	BGE a	greater than or equal to 0 ($N \vee V = 0$)	} Signed
002400	BLT a	less than 0 ($N \vee V = 1$)	
003000	BGT a	greater than 0 [$Z \vee (N \vee V) = 0$]	
003400	BLE a	less than or equal to 0 [$Z \vee (N \vee V) = 1$]	} Unsigned
101000	BHI a	higher ($C \vee Z = 0$)	
101400	BLOS a	lower or same ($C \vee Z = 1$)	
103000	BHIS a	higher or same ($C = 0$)	
103400	BLO a	lower ($C = 1$)	

*The rules for the \vee operation are: $0+0=0$, $0+1=1$, $1+1=0$; the rules for the \wedge operation are: $0+0=0$, $0+1=1$, $1+1=1$.

Machine Language	Assembly Language	Resulting Action	
<u>Condition Code Operators</u>			
000241	CLC	clear C	
000242	CLV	clear V	
000244	CLZ	clear Z	
000250	CLN	clear N	
000257	CCC	clear C, V, Z, N	
000261	SEC	set C	
000262	SEV	set V	
000264	SEZ	set Z	
000270	SEN	set N	
000277	SCC	set C, V, Z, N	
<u>Miscellaneous</u>			
004nDD	JSR Rn,d	jump to subroutine at d	} Rn is linkage register
00020n	RTS Rn	return from subroutine	
000001	WAIT	wait for interrupt	
000002	RTI	return from interrupt	
000003	BPT	breakpoint trap	
000240	NOP	(no operation)	
000000	HALT	halt	

Appendix E

MACRO-11 DIRECTIVES (PARTIAL LIST)

Directive	Explanation
.TITLE title	Title used in assembly listing.
.END	End of source program.
<i>Data storage</i>	
.BYTE exp ₁ , . . . , exp _n	Store the binary values of exp ₁ , . . . , exp _n in successive bytes.
.WORD exp ₁ , . . . , exp _n	Store the binary values of exp ₁ , . . . , exp _n in successive words.
.ASCII /string/	Store the ASCII value of string in consecutive bytes. (/ is any character not in string.)
.ASCIZ /string/	Same as ASCII, but inserts a zero byte after last character of string.
<i>Location counter control</i>	
.=exp	Set counter to value of exp (relocatable).
.EVEN	If current value of counter is odd, add 1 to it.
.ODD	If current value of counter is even, add 1 to it.
.BLKB exp	Reserve a storage block of exp bytes (absolute).
.BLKW exp	Reserve a storage block of exp words (absolute).
<i>Assignments</i>	
sym=exp	Assign to sym the value of exp.
<i>Globals</i>	
.GLOBL sym ₁ , . . . , sym _n	Define sym ₁ , . . . , sym _n as global symbols.
<i>Notation for numbers and characters</i>	
.RADIX r	Henceforth all numbers are base r (2, 4, 8, or 10).
↑Dn or n.	Take n as a decimal number.
↑On	Take n as an octal number.
↑Bn	Take n as a binary number.
↑Cn	Take the 1's complement of n.
'p	Take the ASCII value of the character p.
"p ₁ p ₂	Take the ASCII value of the two-character string p ₁ p ₂ .

Appendix F POWERS OF 2

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1 024
11	2 048
12	4 096
13	8 192
14	16 384
15	32 768
16	65 536
17	131 072
18	262 144
19	524 288
20	1 048 576
21	2 097 152
22	4 194 304
23	8 388 608
24	16 777 216
25	33 554 432
26	67 108 864
27	134 217 728
28	268 435 456
29	536 870 912
30	1 073 741 824
31	2 147 483 648
32	4 294 967 296
33	8 589 934 592
34	17 179 869 184
35	34 359 738 368
36	68 719 476 736
37	137 438 953 472
38	274 877 906 944
39	549 755 813 888
40	1 099 511 627 776
41	2 199 023 255 552
42	4 398 046 511 104
43	8 796 093 022 208
44	17 592 186 044 416
45	35 184 372 088 832
46	70 368 744 177 664
47	140 737 488 355 328

A good assembly language program must be correct, well structured, and properly documented. It is taken for granted that a final version of a program must provide correct answers. The remaining criteria can be classified as “programming style.” Good programming style includes careful annotation and commentary, clear organization, and lucid coding. In writing your program, bear in mind that the reader knows relatively little about the problem you are trying to solve, and next to nothing about your programming habits. Here are some specific pieces of advice:

1. Programs should be modular. Well-defined portions of an algorithm should be coded as subroutines.
2. Each program or subroutine should have a concise introductory comment describing the purpose of the program or subroutine, the algorithm used, the significance of important internal variables, and the meaning and formats of required inputs and generated outputs. For subroutines, describe the meaning and format of parameters and the call convention (e.g., JSR PC,SUB); also list the registers used by the subroutine.
3. Standardize your instruction format. For example:

Label field — column 1
Operation field — column 9
Operand field — column 17
Comments field — column 33

If the operand field extends beyond column 32, simply leave a space and start the comment.

4. A comment should appear with almost every instruction. Such a “marginal” comment should not be simply a literal translation of the instruction, but an explanation of its effect on the program. For example, a comment for BR LOOP should not be “branch to LOOP,” but “return for processing next character.”
5. If marginal comments do not make the operations clear enough, do not hesitate to insert full-line commentary between instructions.
6. A marginal comment extending over several lines should be successively indented. For example:

```

ASR R0      ;DIVIDE
ASR R0      ; OPERAND
ASR R0      ;   BY EIGHT

```

7. Separate distinct parts of the code with blank lines (or a line containing only ; in column 1).
8. Avoid “tricky” or “clever” instruction sequences. If you must include such sequences, provide them with full explanations.
9. Pay attention to all possibilities. For example, in writing a division routine, include a check for zero divisor. If a divisor is found to be zero, some specific action must be taken (an error message is to be printed out, a certain condition code is to be set, etc.).
10. At all costs, avoid writing self-modifying programs (where one or more instructions are altered during execution). Such “nonreentrant” programs are extremely difficult to debug.
11. Subroutines that stand a good chance of being useful in other programs (e.g., I/O routines, multiplication and division routines) should be as “transparent” as possible to the calling program. All registers used by such subroutines should be saved on entry and restored before exit.
12. Storage areas should be grouped together in a single place in the program, not mixed with executable code.
13. To the extent possible, symbols should be mnemonic. For example, points in the program should be named LOOP, NEXTCH (“next character”), EXIT, etc., rather than X1, X2, X3. Similarly, storage locations should be named TEMP, TABLE, CONST, etc., rather than P, Q, R.
14. For better readability, predefine symbols for absolute addresses and constants used in your program: for example, PSR = 177776, BELL = 7, MASK = 177700. Write SP and PC for R6 and R7.
15. Avoid wasteful instructions if efficient ones are just as lucid. For example, write

```

CLR R0      not MOV #0,R0
DEC R3      not SUB #1,R3
TST (SP)+   not ADD #2,SP
JMP LOOP    not MOV #LOOP,PC

```

16. Do not insert test and compare instructions where they are not needed. For example, in

```

DEC R4
CMP R4,#0
BNE LOOP

```

the CMP instruction is redundant. In

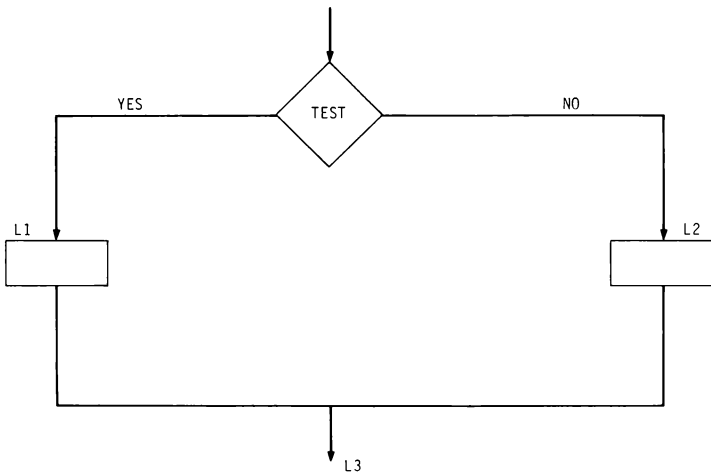
```
SUB  A,B
TST  B
BEQ  NEXT
```

the TST instruction is redundant.

17. Avoid using instructions as data, even if it may save you space.
For example, do not write

```
MOV  @PC,R1
BIC  R1,R5
```

18. The flowchart



should be coded as

```
TST...
BNE L2
L1: ...
...
BR L3
L2: ...
...
L3: ...
```

rather than

```
TST...  
BNE L2  
L1: ...  
...  
L3: ...  
...  
L2: ...  
...  
BR L3
```

19. To reserve a word of storage, write `.BLKW 1`, not `.WORD 0`. (The latter directive indicates that the initial contents of the word is significant.)

INDEX

A

- Absolute address, 137, 138
- Actual argument, 155
- ADC, 28, 89, 175
- ADD, 28, 29 85–87, 175
- Addition, 85–87
 - binary, 7
 - octal, 7
 - two's complement, 19, 20
- Address, 11
- Address expression, 137
- Addressing mode, 29–35, 174
 - absolute, 32, 33, 174
 - autodecrement, 30, 174
 - autodecrement deferred, 30, 174
 - autoincrement, 30, 174
 - autoincrement deferred, 30, 174
 - examples, 31, 39–42

Addressing mode (*cont.*):
 immediate, 30–32, 174
 index, 30, 174
 index deferred, 30, 174
 register, 30, 174
 register deferred, 30, 174
 relative, 33, 34, 174
 relative deferred, 34, 35, 174
Address modification, 140
ALU (*see* Arithmetic and logical unit)
Argument, 64, 155
Argument transmission, 65, 67–70
Arithmetic and logical unit, 10, 12
ASCII, 47, 132, 177
ASCII code, 22, 173
ASCII-to-binary example, 99–105, 134–136
ASCIZ, 47, 177
ASL, 28, 97–99, 175
ASR, 28, 97–99, 175
Assembler, 26, 131–136
Assembler listing, 134–136
Assembly language, 26, 45–51
 format, 50, 51, 179
Assignment directive, 48, 133, 177

B

Backward echo example, 62–64
Basis, 72
BCC, 36, 93, 176
BCS, 36, 93, 176
BEQ, 36, 93, 176
BGE, 36, 93–96, 176
BGT, 36, 93–96, 176
BHI, 36, 93–96, 176
BHIS, 36, 93–96, 176
BIC, 28, 53, 175
Binary number, 1
Binary point, 23
Binary tree, 82–84
BIS, 28, 175
Bit, 10
BIT, 28, 92, 175
BLE, 36, 93–96, 176
BLKB, 48, 132, 177
BLKW, 48, 132, 177, 182

BLO, 36, 93-96, 176
BLOS, 36, 93-96, 176
BLT, 36, 93-96, 176
BMI, 36, 93, 176
BNE, 36, 93, 176
BPL, 114, 176
BPT, 114, 176
BR, 36, 176
Branch instruction, 35-38, 93-97, 176
BVC, 36, 93, 176
BVS, 36, 93, 176
Byte, 10
 high, 11
 low, 11
BYTE, 47, 131, 132, 177

C

Call, 64
Calling program, 64
Card reader, 9
Carry, 20, 85-88
Cathode-ray tube display, 9
C bit, 20, 85-88, 91, 92
CCC, 38, 176
Central memory, 9-11
Central processor, 9, 10, 12
Character, 21
 nonprinting, 21
 special, 21
Character string, 23
CLC, 38, 176
CLN, 38, 176
Clock (*see* Line clock)
Clock status register, 10, 13, 115, 117
CLR, 28, 175
CLV, 38, 176
CLZ, 38, 176
CM (*see* Central memory)
CMP, 28, 87, 88, 91-97, 175
Coding hints, 55-57, 180-182
COM, 28, 175
Comment field, 50, 179
Concatenation, 158
Conditional assembly, 164-167

Conditional block, 164
Condition codes, 12, 20, 38, 93, 97, 176
Conversion method, 2-7
 binary-to-decimal, 5
 binary-to-octal, 7
 decimal-to-binary, 2, 3
 decimal-to-octal, 3-5
 octal-to-binary, 6
 octal-to-decimal, 5, 6
Coroutine, 79, 80
CP (*see* Central processor)
CP priority, 118

D

DD field, 27
Debugging, 113
DEC, 28, 175
Decimal number, 1
Destination operand, 27
Diagnostic, 133
Digit, 21
Directive, 46-50, 177
Dot, 49, 131
Double-operand instruction, 27, 28, 175
Double precision, 23
Double-precision arithmetic, 88-91
Dummy argument, 155

E

END, 49, 177
ENDC, 164
ENDM, 154, 161, 163
ENDR, 161
EVEN, 49, 177
Execution cycle, 27
Exponent, 23

F

Factorial function, 72, 73
Fibonacci numbers, 73
Floating point, 23
Fraction, 23
Front panel, 9

G

General purpose register, 10, 12
Global symbol, 140-146
Global symbol table, 144
GLOBL, 53, 142-144, 177

H

HALT, 27, 38, 176
High-level language, 130

I

IF, 164
IFF, 164, 165
Illegal address trap, 112, 113
Illegal instruction trap, 112, 113
INC, 28, 175
Induction step, 72
Instruction, 27, 28, 175, 176
 byte, 28
 word, 28
Interrupt, 114-125
Interrupt enable bit, 13, 115, 118
Interrupt handler, 114, 120
Interrupt routine, 114, 120
Interrupt vector, 114
INTR ENBL bit (*see* Interrupt enable bit)
IRP, 163
IRPC, 163

J

JMP, 28, 29, 37, 175
JSR, 65, 176

K

K, 11
Keyboard, 12, 115
Keyboard data register, 10, 13
Keyboard status register, 10, 12, 115

L

Label, 50, 160

- Label field, 50
- Letter, 21
- Line clock, 9, 10, 13, 115, 117
- Linkage, 65
- Linkage editor, 130, 131, 138-146
- Linkage register, 65
- LINKR-11, 131, 138
- Loader, 49
- Loading origin, 137
- Load map, 145
- Load module, 138
- Local symbol, 142, 159-161
- Location, 11
- Location counter, 49, 131, 177

M

- Machine language, 26, 45, 46
- Macro, 151-167
 - nested, 156
- MACRO, 154, 155
- Macro body, 154
- Macro definition, 154, 155
- MACRO-11, 46, 131-136, 153
- Macro heading, 154
- Macro recursion, 166
- Magnetic disk, 9
- Magnetic tape, 9
- Mantissa, 23
- Merged global symbol table, 145
- Mode subfield, 29
- Modular programming, 64, 151, 179
- Monitoring program, 114
- Most significant bit, 18
- MOV, 28, 175
- MSB (*see* Most significant bit)
- Multiple echo example, 51-54
- Multiplication, 67, 68, 99, 102

N

- NARG, 155
- N bit, 12, 91, 92
- NEG, 28, 88, 175
- No-operand instruction, 38

NOP, 176
Null, 23
Number, 1, 15
Number system, 1-8

O

Object module, 138
Octal contents, 11
Octal number, 1
ODD, 49, 177
Offset, 35
One's complement, 16
Op-code, 27
Op-code table, 131
Open subroutine, 153
Operand field, 50
Operation code, 27
Operator field, 50
Overflow, 20, 85-88, 91

P

Paper tape punch, 9
Parameter, 64
Parity bit, 21
PC (*see* Program counter)
PDP-11 organization, 9-14, 172
Peripheral device, 9, 116
Permutations, 73
PIC (*see* Position-independent code)
Polling, 116
Pop, 61, 62
Position-independent code, 34, 36, 147, 148
Powers of 2, 6, 178
Powers of 8, 6
Printer, 9, 12, 13, 115
Printer data register, 10, 13
Printer status register, 10, 13, 115
Priority field, 118
Priority interrupt, 118-125
Processor status register, 10, 12, 118, 119
Processor trap, 111-114
Program counter, 10, 12, 27
Programming style, 57, 179-182

PSR (*see* Processor status register)
Pure procedure, 65
Push, 61, 62

R

RADIX, 48, 177
Ready bit, 13, 114, 115
Real number, 23
Recursive function, 72
Recursive subroutine, 72-79
Reentrant code, 65, 180
Register (*see* specific register name)
Register subfield, 29
Relocatable address, 137, 138
Relocation, 136-148
Repeat block, 161
Repeat directive, 161-164
REPT, 161
Return, 64
ROL, 28, 97-99, 175
ROR, 28, 97-99, 175
RTI, 115, 176
RTS, 66, 176

S

SBC, 28, 90, 175
SCC, 38, 176
Scientific notation, 23
SEC, 38, 176
SEN, 38, 176
SEV, 38, 176
SEZ, 38, 176
Shift instruction, 97-99
Sign bit, 18
Single-operand instruction, 27, 28, 175
Single precision, 23
Source operand, 28
SP (*see* Stack pointer)
SS field, 28
Stack, 60-64
Stack calculator, 108-110
Stack overflow, 62
Stack pointer, 10, 12, 61, 66

Stack underflow, 62
SUB, 28, 29, 87, 88, 175
Subroutine, 64–80, 116, 138, 151–154, 179
 nested, 70–72
Subtraction, 19, 87, 88
SWAB, 28, 29, 175
Symbol table, 131, 134, 136
System stack, 66, 112

T

T bit, 113, 114
Teletype, 9, 10, 12, 13
Time request example, 120–125
TITLE, 49, 177
Top-down programming, 64
Tower of Hanoi example, 74–79
Trap, 111–114
Trap bit, 113, 114
Trap routine, 112
Trap vector, 112, 113
TST, 28, 91, 92, 175
TTY (*see* Teletype)
Two-pass assembly, 131, 132
Two-pass linkage, 144–146
Two's complement, 15–19

V

V bit, 20, 85–88, 91, 92

W

WAIT, 176
Word, 10
 high order, 88
 low order, 88
WORD, 47, 131, 132, 177, 182

Z

Z bit, 12, 91, 92