STEPHEN A. LONGO

INTRODUCTION TO DECSYSTEM 20 ASSEMBLY PROGRAMMING

Introduction to DECSYSTEM-20™ Assembly Programming

Introduction to DECSYSTEM-20™ Assembly Programming

Stephen A. Longo
La Salle College

Brooks/Cole Publishing Company
Monterey, California

To Rachael and Stevie for their excitement

To Janice for her strength

To all three for their love

Brooks/Cole Publishing Company
A Division of Wadsworth, Inc.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

Longo, Stephen A., [Date]
Introduction to DECSYSTEM-20™ assembly.

Includes index.
1. DECSYSTEM-20 (Computer)—Programming.
2. Assembly language (Computer program language)
I. Title.
QA76.8.D17L66 1984 001.64'2 83-7414
ISBN 0-534-02942-6

Sponsoring Editor: *Michael Needham*
Production Editor: *Richard Mason*
Manuscript Editor: *Adrienne Cordova*
Interior and Cover Design: *Katherine Minerva*
Illustrations: *John Foster*
Typesetting: *Graphic Typesetting Service, Los Angeles*

PREFACE

Today, with all of the advances in computer hardware and software, learning assembly language remains a challenging endeavor. Assembly is a highly complex language requiring more active programming skills than do, say, high-level languages. Only after an assembly course does a student start fully to appreciate high-level languages (and also to note some of the shortcomings of high-level languages). Even more important, a course in assembly gives students a better understanding of what takes place *inside* a computer.

There are many differences between a high-level language and an assembly language. Each statement in a high-level language represents a sequence of computer instructions that a person can read. In addition, the programmer here need not be concerned with some of the specific, highly detailed aspects of the computer hardware. In assembly language, however, the programmer must be aware of these complex details, which in turn allows the programmer more control of the computer. Assembly programming also demands that each operation be thought of as a simple, isolated step rather than as part of a sequence of steps.

I feel that a student's first course in assembly should not be comparative but that it should deal instead only with a specific assembly. The choice of which particular assembly language to use will depend on what hardware is available to the student. At La Salle College we have a number of different computers, but I feel it is best to teach the students DECSYSTEM-20 assembly because it is a very *complete* assembly; it contains many features that are not present in other assemblers. Because of the richness of the DECSYSTEM-20 assembly, students subsequently have very few problems in future courses that deal with microprocessor assembly (Z80, 6800, etc.).

The real challenge in most problems assigned to computer students lies not in the nature of the answer but in how to achieve that answer. In other words, challenge lies in designing a method, an algorithm, that will produce an expected result. The computer's fast turn-around time and helpful messages assist students in this endeavor. Students can learn from their mistakes and are motivated to correct them. Because of these encouragements I strongly believe in hands-on experience. Therefore, I have arranged topics and programs in such a way that a student will be able to use the computer as soon as possible. This may make the initial chapter a little oversimplified, but it does afford the student the opportunity to start programming after only a few lectures.

Rather than starting off with a discussion of machine code, this text deals first with single-character operations. There are two reasons for this arrangement: first, I feel that the single character (byte) and its encoding (ASCII) is the fundamental entity of assembly, and that therefore this should be dealt with as soon as possible; second, computers do not work in decimal—therefore students must learn a new radix. By postponing a study of machine code, students can first learn enough about assembly so that they can write simple assembly programs (homework assignments) that convert numbers from one radix to another. These programs will help students later on when they study machine code; it will also help them with exercises that require them to code radix conversions by hand.

This text has two parts. The first seven chapters introduce the student to some basic concepts in assembly: terminal input/output; jumps; addressing modes; numbers, radices, and bits; logical operators and shifts; transferring data using pointers. I have purposely delayed discussing those facilities that would switch the burden of programming from the student to the computer (e.g., MACROs) so as to ensure that the student first understands fundamentals. I have also demonstrated how many of the sophisticated JSYS (e.g., input/output) can be simulated by more primitive calls. By the end of Chapter 7, students should have a good command of basic assembly concepts, many of which will be transportable to other assemblers, especially microcomputer assemblers.

The last four chapters deal with concepts with which the student is supposed to be familiar—stacks and subroutines (Chapter 8), files (Chapter 10), and interrupts (Chapter 11)—with Chapter 9 given over to MACROs. The emphasis in these chapters is not on teaching these concepts but rather on showing how they are to be implemented in assembly. For instance, in the chapter dealing with files I supply some MACROs, but students will

also be expected to write their own for two reasons: first, students can always use good, practical exercises dealing with MACROs; second, writing all the basic steps necessary for file operations helps the student to understand files better.

This is a learning text rather than a reference book. But, so as to help the reader seeking to use this as a reference book, I have included a number of appendices to allow for easy access to definitions.

There were many individuals who helped me with this text. I would like to acknowledge Hal Dell as well as the many students who used the preliminary version of the book, with special thanks to Rick Smith. I would like to thank the people who reviewed the manuscript: Carl Fussell of the University of Santa Clara, Ralph E. Gorin of Stanford University, and Charles M. Shub. Of course, those who have suffered through writing a text know the contribution a wife makes—thank you Janice!

Stephen A. Longo

CONTENTS

CHAPTER

1

Introduction to Assembly

This chapter provides a bird's-eye view of assembly language as preparation for the chapters that follow. The discussion assumes that you are familiar with a high-level language like BASIC or FORTRAN and introduces you to the similarities and differences between these languages and assembly language. The analogous elements of the two types of languages offer helpful starting points; you will also find, however, that assembly language is more complex, requiring more active programming skills than do high-level languages. By the end of this chapter, the bare outlines of assembly programming will have begun to emerge.

1.1 Assembly versus High-Level Languages

The computer's three main parts are the control, input/output (I/O), and the memory. The memory stores information as an ordered set of locations. It is analogous to rows of boxes in which each box has an address (the boxes are numbered), and in which you can place numbers and retrieve them as you need them (Figure 1-1).

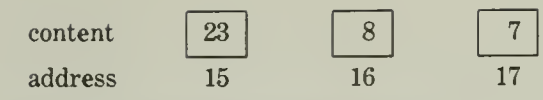


Figure 1-1 Memory Locations (Addresses) and Memory Contents

Let us look at how a translator (compiler, interpreter) of a high-level language uses memory. Consider the statement I = 3, which assigns a value of 3 to the variable I. The computer must pick a memory location (a box), place the 3 in the location, and remember that the name of the location is now I. To help it find the box I more easily, the computer constructs a table (symbol table) in which it writes the symbol I accompanied by the box number. The next time the computer needs the information represented by I, it goes to the symbol table, finds the I, reads the memory location (the box number), goes to that location, and looks in that box.

I = 3	I 54	3
statement	symbol table	address 54

To avoid multiple definitions (assigning the same name to more than one box), the translator checks the symbol table every time it encounters a variable. If the symbol is not in the table, the translator adds it. If the symbol is already there, the translator uses the previously defined location.

In general, one line in a high-level language causes the computer to do more than one operation. The simple statement I = 3 does two things, for example. It assigns a name, I, to a location, and it places a value, 3, in that location. In assembly language, however, each line of code causes one operation only and does not contain any hidden cues to perform other operations. Thus, a statement in high-level language is essentially made up of a number of assembly language statements.

Let us now consider the statement J = I + K, which assigns to the variable J the value of the sum of the variables I and K. Where does the computer perform addition? How do the memory locations interact with each other? To answer such questions, we need to look further at how the memory works. A computer has many memory locations (thousands, millions). For every memory location to be able to communicate directly with every other location, the memory works something like a telephone system. Clearly, it is not practical to have a separate telephone wire from your house to every location you call; instead one wire goes from your house to a switching station, which acts as a central connecting point. When you want to talk to a friend, the switching station connects your

line to your friend's line; it essentially takes your information and routes it to your friend (Figure 1-2). This preferred connection (switching station) eliminates many costly, cumbersome interconnections though it has its own cost: busy signals when connections are unavailable. Similarly, computers do not tie memory locations directly to each other but to preferred locations called *accumulators*. In general, then, the memory locations just hold information (numbers) and interact only through accumulators. The accumulators, on the other hand, can interact with any part of the computer, can hold information, and can operate on (change) information. Computers differ in the number of accumulators they have and in the operations that the accumulators can perform. The DECSYSTEM-20 has 16 accumulators.

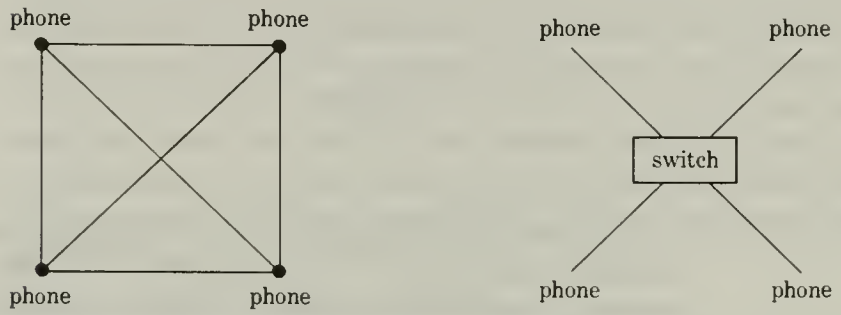
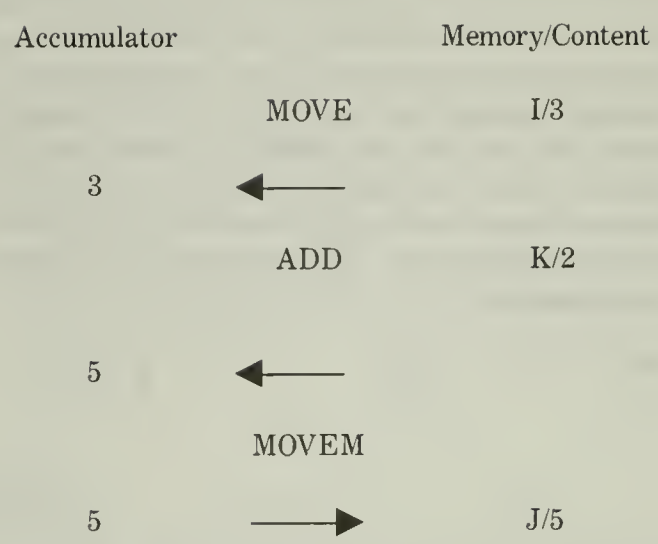


Figure 1-2 Telephone Interconnection Models

With this new information on accumulators, we can again look at the statement J = I + K. This statement requires the operation of addition. Since the accumulator is an integral part of such operations, we must first copy (MOVE) the contents of memory location I to an accumulator. Next, we add the contents of memory location K to the accumulator. Finally, we must copy (MOVEM) the contents of the accumulator to the memory location J.

J = I + K

5 = 3 + 2



Thus, the simple statement I = J + K in a high-level language in effect represents several lines of assembly code, with each line of code representing one step for the computer. Because assembly language works with the fundamental locations of information (memory and accumulators), users are responsible for more bookkeeping than they would be in a high-level language.

1.2 Constructing Statements

Statements in high-level languages generally take the following form:

label variable assignment expression

For example,

300 I = J + K

On the other hand, a statement line in assembly can have this form:

label opcode operand, operand

The label, if used, is the first thing on the line in both high-level and assembly languages. Labels are very important because they allow for the nonsequential flow of a program. (The program need not progress according to the physical progression of lines; a program can loop back or

forward as necessary, using the labels as reference points.) In high-level languages like BASIC that require line numbering, the line numbers are the labels. Those languages that don't require line numbering may restrict the form of the label (for example, by specifying numbers only) or possibly specify the position (for example, by limiting the label to certain columns).

Assembly language uses labels and generally allows almost any character and any number of characters in forming a label. DECSYSTEM-20 labels have very few restrictions, other than that they must end with a colon (:).

The expression part of a high-level language statement can contain more than one computer operation (I * J + K), whereas assembly language allows only one operation per statement. The *opcode* symbol in the assembly statement represents the operation. Examples of opcodes are MOVE and ADD. The variables in the operation are the operands. Assembly languages differ in the number of operands they allow per line. Most computers with more than one accumulator use two operands per line. Some microcomputers use only one operand, and some stack-oriented systems have no operands. Since operations generally require accumulators, one operand must be an accumulator. Thus, a DECSYSTEM-20 assembly statement line has the following form:

```
label:    opcode    ac,    operand
```

The comma after the accumulator is important. Just as the colon identifies the preceding symbol as a label, the comma identifies the preceding symbol as an accumulator. (*Note:* Some DECSYSTEM-20 assembly opcodes do not appear to reference an accumulator—for example, SET to Zero a memory location, or SETZ memory). The DECSYSTEM-20's 16 accumulators are numbered starting with zero (0). If an assembly statement does not reference an accumulator—that is, it has no comma—then the translator assumes accumulator 0. Because accumulator 0 requires different treatment from other accumulators, beginning assembly programmers should generally avoid using it.

The DECSYSTEM-20 also uses spaces and/or tabs as *delimiters* (characters that separate symbols within the line). The space or tab separates the opcode and the accumulator. The DECSYSTEM-20 also allows on-line comments (they are generally necessary). A semicolon signifies that a comment follows. A complete DECSYSTEM-20 statement is composed of five fields: label, opcode, accumulator, operand, and comment. A statement, then, will take the form:

```
label:    opcode    ac,    operand          ;comment
```

A computer scans (reads) a line character by character, starting at the beginning of the line and constructing the fields. The delimiters inform the computer when to start and/or stop a field. The computer ignores leading spaces and tabs and reads the first nonspace or nontab as the first symbol. The trailing delimiter identifies the field represented by the symbol. When the computer reads a colon, it identifies the previous symbol as a label and begins to construct the next symbol when it reads the next nonspace or nontab. This form of scanning allows the label field to have more than one label because the label field does not end until the computer reads a symbol not delimited by a colon. For example:

```
label1:    label2:    opcode . . .
```

Once the label field closes—that is, the system constructs another symbol that ends with a space or tab—a new symbol will most probably be an opcode. Once the opcode field closes, the system starts the next field, and so on. The computer continues to ignore leading spaces and tabs and does not start the next symbol until it finds a nonspace or nontab. If a symbol ends with a comma, then it is an accumulator; if it ends with a space or tab, then the field is an operand. In most cases, if the system reads a semicolon or a carriage return, it stops scanning the line.

1.3 Simple Assembly Programs

The computer (interpreter/compiler) reads programs written in a high-level language and converts them into a specific language (machine code). The computer then uses the machine language to execute the program. Statements can generate executable and nonexecutable code.

A *data line* is an example of a high-level statement that does not generate executable code. A data line requests that values be placed in locations for use at run (execution) time. Another example is a dimension statement, which warns the computer to set aside contiguous memory. In high-level languages, nonexecutable statements generally have a specific position in the beginning of a program; if these statements appear in the body of the program, a computer error can result.

Assembly language also has executable and nonexecutable statements, which must be separated to avoid errors. As in high-level languages, assembly comments, though nonexecutable, can appear anywhere in your program. The simple label statement I: 3 is an example of a nonexecutable statement in assembly language. This statement identifies a location, this particular line, as I, which in this location is a 3. The computer does not

treat the 3 as an opcode in this situation: if only one field exists and it could be an opcode, an ac, or an operand, then the computer treats the field as an operand. In other words, if an opcode has nothing to operate on, then it ceases to be an opcode and defaults to an operand.

All symbols, like opcodes, operands and labels, must be represented in the computer memory as unique numbers. This conversion is the responsibility of the assembly program, also referred to as the *assembler*. (The assembler is comparable to the compiler or interpreter that scans statements in high-level languages.) The assembler first scans an assembly program and generates machine code. If the scanned line has an opcode, the code generated is executable and is placed in (sequential) memory location. If the scanned line has no opcode the code generated becomes nonexecutable. The (relative) address of the memory location presently being filled is kept in a register called the *location counter*. As each word is placed in memory the location counter is incremented. When a label is used, like I: 3, the symbol, I, is assigned the present value of the location counter and is placed in the symbol table.

```
1003                . . .
1004                I:3          I      1004
1005                . . .

location counter    program      symbol table
```

Location 1004 here now contains a 3. When the I appears in a nonlabel field the translator will search the symbol table and replace the I with its original location counter value.

A coded program by itself, essentially a dump of memory, shows no distinction between data and instructions and gives no indication where the program starts or stops. The assembly programmer (you) must inform the computer where the program ends and where the executable part begins by labeling the first executable statement and placing the word END after the last line of the program. The symbol on the line with the word END labels the first executable line. Nonexecutable lines can appear after the executable lines as long as the computer can avoid these lines during execution. High-level languages use a STOP or GOTO END for this purpose; in assembly, EXIT serves the same purpose (we will discuss a better choice later). An assembly program then has this structure:

```
        nonexecutable
start:  executable
        EXIT
        nonexecutable
        END start
```

The label *start* is not a reserved word like END; any label can appear in its place. A stop symbol, like EXIT, is necessary in all assembly programs. Also, the information in an assembly statement, as in a high-level language statement, normally flows from right to left. One exception is the trailing M in the opcode MOVEM, which changes the flow of information to left to right.

A sample assembly program similar to J = I + K follows.

```
        EXAMPLE 1, ADDING TWO NUMBERS

        I:      3       ;I=3
        K:      2       ;K=2
        J:      0       ;save a spot called J
START:  MOVE 1,I        ;move contents of I to ac 1
        ADD 1,K         ;add to ac 1 the contents of K
        MOVEM 1,J       ;move contents of ac 1 to J
        EXIT
        END START
```

The answer, 5, is in location J. To see the answer, we need I/O routines, which we will discuss in Chapter 2.

1.4 Summary

One of the main differences between assembly programming and high-level language programming is that the assembly program has to contain within it some understanding of the basic interaction between a program and memory. The basic operations in assembly involve addressing data via memory locations, which themselves interact only through central accumulators. Each operation on the data must be thought of in terms of simple steps, for each line of code causes one operation alone to be performed rather than a succession of operations. This is similar to constructing English sentences gradually with only simple subjects, verbs, and occasional objects.

CHAPTER

2

Terminal Input/Output

The I/O section of the computer is the computer's link to the outside world—to the terminal, discs, other programs, and so on. Thus, if you wish information to go to or come from the computer, it must pass through the computer's representative, the I/O device. Because the I/O section can deal with only one character at a time, all data exchange must be in the form of simple characters. (There are more sophisticated methods that we will not discuss here.) I/O programs help you break down data into single-character elements. We will discuss terminal I/O in this chapter.

2.1 Building Blocks for Input/Output

If you were to look at text written in characters unfamiliar to you, like Chinese or Arabic, you would need someone to translate the characters into symbols you understand. Computers are similar; they deal only with numbers. If you want them to deal with letters, you must translate (code) the letters into numerals. In fact, when you write a program, you are not dealing so much with letters or numerals as with symbols; these symbols

have no inherent meaning other than that they are different from each other. For instance, the letters *xse* undoubtedly affect you differently than the letters *sex*. The arrangement of the letters and your previous experience with the arrangement give them meaning.

Programming allows you to develop the same range of meaning with numbers. You probably see any arrangement of numerals as a new number. For example, you probably read 123 as one hundred twenty-three. If so, you are assuming a base 10 structure and assigning meaning to the positions of the symbols (1 times 100, plus 2 times 10, plus 3). However, computers assume nothing; you must tell them everything. To the computer, 123 has no more meaning than *xse*. When you program the computer, however, you give meaning to a group of characters.

You must assign a code (a numeral) to every character (letter and numeral) you want the computer to use. The most common code is ASCII, the American Standard Code for Information Interchange. For example, assuming base 10, ASCII assigns A the value 65 and 1 the value 49. The I/O device (buffer) works with the ASCII value of each character.

When you press a key on a terminal, the terminal generates the ASCII value of the key and sends that code to the computer's input buffer. The computer does not read 123 as one hundred twenty-three. The terminal sends the computer the ASCII code for a 1, then the code for a 2, and finally the code for a 3. When the terminal acts as an output device (printer) it expects the characters it receives to be in ASCII code. Therefore the computer must strip down a number to its individual characters to send it to the terminal. It sends the ASCII value of a 1 to the output buffer, then the value of a 2, and so on. The need for programs that will assist in I/O operations is obvious.

2.2 Monitor Calls

Eventually you will write I/O routines, but for now we will focus on the standard (canned) routines supplied with the DECSYSTEM-20. These routines are in the monitor, and you can access them via a symbol table in a file called MONSYM (MONitor SYMbols). The monitor is a management and protection system that aids users in I/O work. In a time-sharing environment, the I/O may link the computer to other users or even to operating system programs. The monitor via MONSYM helps coordinate I/O requests and shows whether users' requests interfere with existing requests.

No matter how fancy the monitor's I/O routines are, they must still deal with the single-character I/O buffer. The fundamental building blocks of

all terminal I/O are the monitor calls PBIN% and PBOUT%, Primary (terminal) Byte (character) IN and OUT. These calls are also known as JSYS, (Jump to SYStem). (The JSYS routine of the DECSYSTEM-20 is one main difference between it and the DECSYSTEM-10.) A PBOUT% is necessary if a program is to send a character to the terminal. The ASCII equivalent of the character must be placed in accumulator 1 and then PBOUT%ed. The PBOUT% takes the character from accumulator 1 and passes it to the I/O buffer, which in turn sends it to the terminal.

A PBIN% is necessary if the program requires a character during run time. When the system executes the PBIN%, it waits for a keyboard character. When a key is pressed, the program accepts the character and continues. The ASCII value of the character is in accumulator 1. (*Note:* Only a few monitor symbols require the trailing %. This text recommends its use to make monitor calls stand out.)

Let us write a program that will accept a character from the terminal and return the next sequential character—for example, the user types in an A and the computer sends back a B, or the user types in a 3 and receives back a 4. The symbols PBIN% and PBOUT% are in the program, but we will not define them there since the file MONSYM has access to their definitions. The assembler, the program that converts mnemonics (MOVE, ADD) into code, understands opcodes and some other symbols (for example, END). It can handle symbols (labels) defined within the program by constructing a symbol table. However, the computer will not know the meaning of PBIN% or any monitor call unless we give it the directive SEARCH. Thus, when the assembler cannot find a symbol in the program table, it knows to look in those tables following the directive SEARCH. (*Note:* It is a good habit to title a program by TITLE followed by a user-assigned name. Only the first six characters of the name have meaning. Also, for the DECSYSTEM-20, the preferred way to stop the execution of a program is the monitor call HALTF% rather than EXIT, which is primarily a DECSYSTEM-10 call.

In the format of an assembly program it is customary (though not necessary) to place labels at the beginning of a line and then tab to the opcode field. If there are no labels, you will still tab to the opcode field.

```
        TITLE EXAMPLE 2     ;this will be referred to as EXAMPL
        SEARCH MONSYM
I:      1                   ;needed for incrementing

START:  PBIN%               ;get a character
        ADD 1,I             ;add one to the ASCII value in ac 1
        PBOUT%              ;send character to the terminal
        HALTF%              ;stop execution
        END START
```

In the above program, the original character came from the terminal and was thus already in ASCII. The program instructed the computer to add 1 to the ASCII and then sent that value back to the terminal.

Some beginners might misread the statement ADD 1, I as instructing the computer to add 1 to I. However, the comma following the 1 is a clue that it is an accumulator. As in high-level languages, the normal flow of a statement is from right to left. Thus, the statement tells the computer to add the contents of I to the contents of accumulator 1. Another assembly directive, an assignment symbol (=), may help in this type of statement. For example, the statement T1=1 (no spaces and tabs on either side of the equals sign) tells the computer to place the symbol T1 and its meaning, 1, in the symbol table. The symbol has no memory location associated with it as would a label. Assembly directives also do not generate code themselves; they assist the assembly program in generating code. Assembly directives can appear anywhere in a program but must appear before an assembly statement that needs their assistance. Once the definition of T1 is carried out in the above statement, then the statement ADD T1,I can be used. The statement ADD T1,T1 doubles the contents of accumulator 1.

Now that we know how to get information from the computer, let us return to the addition example at the end of Chapter 1. In that example, we placed the sum of the two numbers in location J. This time we will send the answer to the terminal via PBOUT%. Even though the answer is already in accumulator 1, we cannot simply PBOUT%. The answer in accumulator 1 is 5; the computer must convert it to an ASCII 5. In base 10, the ASCII of a 0 is 48, of a 1 is 49 (48 + 1), of a 2 is 50 (48 + 2), and so on. To send the 5 to the terminal, you might assume that you should add the ASCII value of 0, or 48, and then PBOUT%. However, this approach will not work because the computer does not work in decimal. The DECSYSTEM-20 assumes that all numbers are octal numbers—that is, in base 8. (We

will discuss the reasons for this fact in Chapter 5.) Thus, if you use a 23 in your program, the computer will treat it like 2 times 8, plus 3, rather than 2 times 10, plus 3. We can temporarily postpone the problem of dealing with octal numbers by using the assembly directive RADIX 10. This directive states that all numbers that appear thereafter will be treated in base 10, as illustrated below.

```
        TITLE EXAMPLE 3  ;displaying a sum
        SEARCH MONSYM
        RADIX 10
I:      3
K:      2
OFFSET: 48
START:  MOVE 1,I
        ADD 1,K
        ADD 1,OFFSET     ;add ASCII offset
        PBOUT%
        HALTF%
        END START
```

2.3 Three Plus Seven Does Not Equal Ten

If we change the value of K to 7 in the above program, the answer will be colon (:) rather than 10. If we add 48 to 10, the answer is the ASCII for a colon, not for 10. No ASCII exists for a ten; the computer reads a ten as two symbols, a 1 followed by a 0. A routine is available for stripping down a number of more than one symbol to its individual characters and for converting these characters to their ASCII equivalent. This routine, NOUT% (Number OUT), can send an integer to any device, not just the terminal, and can convert the number in the computer to any *radix* (base) from base 2 to base 36. (This capability has nothing to do with the assembly directive RADIX used by the assembler.)

NOUT% needs three parameters: the name of the device to which the number should be sent, the number, and the desired base. The computer passes these parameters to the routine in the monitor via accumulators. For NOUT%, accumulator 1 must contain the name of the device, accumulator 2 must contain the number, and accumulator 3 must contain the base. The printing part of the terminal is .PRIOU, PRImary OUt; the leading dot is part of the name. The NOUT% routine, like many other monitor routines, checks errors. If the routine is error-free, then the computer skips the statement after the NOUT%. If the NOUT% routine receives a

non-number, a letter, or a decimal point—all errors—the machine will execute the instruction following NOUT%. This type of monitor call, which either skips or executes the instruction following it, is a "plus one or plus two return." Your program must provide an instruction in both locations. In the error location (plus one), use an ERJMP (ERror JuMP), such as ERJMP LABEL, where LABEL is a labeled location that can contain a simple HALTF% or a fancier user routine. The example below adds two numbers and displays the numbers via NOUT%.

```
        TITLE EXAMPLE 4
        SEARCH MONSYM
        RADIX 10
I:      3
K:      7
BASE:   10              ;base used by output
TTYOUT: .PRIOU
START:  MOVE 2,I        ;for NOUT% ac 2 must have data
        ADD 2,K
        MOVE 1,TTYOUT   ;set up device
        MOVE 3, BASE    ;set up base
        NOUT%
          ERJMP ERROR   ;in case of error
        HALTF%          ;program done
ERROR:  HALTF%
        END START
```

A similar routine is available for accepting a number: NIN%, Number IN. This routine also uses three accumulators. Accumulator 1 has the name of the device containing the number, and accumulator 3 contains the radix of the number. If NIN% is successful, accumulator 2 contains the number. The terminal keyboard is .PRIIN, or the PRImary INput device. NIN%, like NOUT%, is a plus one, plus two monitor call. The NIN% statement is like INPUT in BASIC or READ in FORTRAN, except that it includes no prompts, question marks, or stars. NIN% can handle bases from 2 to 10. The following program accepts two numbers from the terminal, adds them, and displays their sum at the terminal.

```
        TITLE EXAMPLE 5
        SEARCH MONSYM
        RADIX 10
I:      0                   ;save a place for
                            ;the first number
BASE:   10
TTYOUT: .PRIOU
TTYIN:  .PRIIN
START:  MOVE 1,TTYIN        ;set up NIN% device
        MOVE 3, BASE        ;set up NIN% radix
        NIN%                ;get first number
          ERJMP ERROR
        MOVEM 2,I           ;save first number
        NIN%                ;get second number
          ERJMP ERROR
        ADD 2,I             ;add first to second
        MOVE 1, TTYOUT      ;set up NOUT% device
        NOUT%
          ERJMP ERROR
ERROR:  HALTF%
        END START
```

Upon executing the above program, the computer will send a message something like the following:

```
MACRO: EXAMPL
LINK: Loading
[LNKXCT EXAMPL execution]
```

EXAMPL is from the TITLE line in the program; the rest of the message is the same for any program. The system is now waiting to do NIN%. The user types in numbers on the keyboard, PRIIN. The NIN% statement will end at the first nonnumber, letter, or carriage return. The program will continue until the second NIN%. When this NIN% ends, the program will continue to the NOUT% and display the answer. When the computer reaches a HALTF%, it will end the program and return to the TOPS-20 level.

We could use two improvements in this program, however. First, we would like to be able to use the on-line editor (for example, control U, delete key) but cannot since NIN% ends at the first nonnumerical character. Second, we need a message to prompt the user about NIN%. We will treat the problem of messages in the next section. Chapter 7 will discuss the difficulty presented by nonnumerical characters.

2.4 Messages

PBOUT% is one way to have the computer send messages, or strings, to the terminal. You must first place the message, one character per memory location, in the program. Then MOVE the characters, one at a time, to accumulator 1 and PBOUT%. For example, if you want to send the message AH to the terminal, you cannot place the message in the program this way:

```
I:    A
J:    H
```

Because the computer only understands numbers, it treats the letter A as an undefined symbol. You must therefore place the ASCII value of the character in a memory location.

```
I:    65    ;A
J:    72    ;H
```

To simplify this process, you may use an assembly directive of double quotation marks. The assembler will replace any character within quotation marks by its ASCII equivalent. (You may place up to five characters within quotation marks, but since we are dealing with PBOUT%, we will use only one character.) The following program will send a message to the terminal.

```
          TITLE EXAMPLE 6
          SEARCH MONSYM
I:        "A"
J:        "H"
START:    MOVE 1,I
          PBOUT%
          MOVE 1,J
          PBOUT%
          HALTF%
          END START
```

Most messages are longer than two characters, however. Thus, the monitor call PSOUT% (Primary String OUT) deals with strings. Though we will discuss PSOUT% in more detail later, you need to know now that the message you wish to PSOUT% is packed into the computer by the assembly directive ASCIZ. After the ASCIZ directive, you need a leading character to signify the beginning and end of the message. This character also enables

you to use spaces, tabs, or carriage returns in the message. Designation of a special character for this purpose would restrict you from using it in your message; thus, you may use any character to block off a message, as illustrated below.

```
MESS1:  ASCIZ /the moon is blue/
MESS2:  ASCIZ ? HOW ARE YOU ?
MESS3:  ASCIZ * FILL IN THE DASHED LINE
---------------*
```

The delimiters, /, ?, and *, are not part of the message. Note also the carriage return between the asterisks (*) in the last line.

To use PSOUT%, place the address of the message in accumulator 1. PSOUT% returns plus one (no error checking). In this case, you should not MOVE the address of the message to accumulator 1; you must HRROI (an opcode) the address—for example, HRROI 1,MESS. The program below accepts a number from the terminal in base 10 and sends the base 8 equivalent of the number to the terminal.

```
        TITLE EXAMPLE 7
        SEARCH MONSYM
        RADIX 10
BASE10: 10
BASE8:  8
TTYIN:  .PRIIN
TTYOUT: .PRIOU
MESS1:  ASCIZ / TYPE IN A NUMBER /
MESS2:  ASCIZ / THE OCTAL EQUIVALENT IS /
START:  HRROI 1, MESS1      ;set up PSOUT%
        PSOUT%              ;send mess1
        MOVE 1, TTYIN       ;set up NIN%
        MOVE 3, BASE10
        NIN%
          ERJMP OOPS
        HRROI 1, MESS2
        PSOUT%              ;send mess2
        MOVE 1, TTYOUT      ;set up NOUT%
        MOVE 3, BASE8       ;output radix is 8
        NOUT%               ;because of NIN%
                            ;number is already in ac 2
          ERJMP OOPS
        HALTF%
MESS3:  ASCIZ * I/O ERROR *
OOPS:   HRROI 1, MESS3
        PSOUT%
        HALTF%
        END START
```

More opcodes will be necessary to write more meaningful programs.

2.5 Summary

The main point in this chapter is to realize that most I/O data exchange must be in the form of streams of simple characters. The structure associated with groups of characters must be placed in algorithms. These algorithms are collected in a library called MONSYM. Parameters needed/produced by the algorithms are passed to/from the library via accumulators.

2.6 Exercises

1. Write a program that will accept a character using a PBIN%; then print the character using NOUT%. (The output should be the ASCII equivalent of the character.)
2. Write a program that accepts an integer via NIN%; then display the integer using a PBOUT%. (This problem is the opposite of problem 1. Note that for integers less than 32, radix 10, two characters are printed.)
3. Using only PBIN%s, PBOUT%s, and PSOUT%s, write a program that will accept three single-digit integers. Add the first two integers, then subtract the third integer from the sum. (Choose integers such that your answer is greater than 0 and less than 10. Why can you ignore the offset problem in this exercise? Try adding three integers and subtracting one; you will need to consider offset.)
4. Write a program that converts integers from base 10 to any other base. (Accept via NIN% two integers, the integer to be converted, and the new base; then NOUT% the result. What happens if the output base is larger than 10?)
5. Accept an integer from the terminal. Add the integer to an integer stored in your program and subtract it from the same integer. Print the output as follows (assume the integer in the program is 20):

```
TYPE IN AN INTEGER 24
SUM 44
DIF  4
```

CHAPTER

3

Jumps

This chapter discusses unconditional and conditional JUMP statements, which allow flexibility in the flow of a program. This "jump" capability sets apart computers from calculators, for the value of the variables at run time can determine the path through a computer program. GOTO is an example of an unconditional jump statement in a high-level language. IF/THEN is an example of a conditional jump. Though some programmers question the need for unconditional jumps in programs, no one challenges conditional jumps.

3.1 A Jump Scorecard

In assembly language, you may represent an unconditional jump in two ways: JRST and JUMPA. JRST is preferred because of execution timing.

Conditional jumps must compare two things, and they rely on the same conditions as conditional jumps in high-level languages—namely, less than, less than or equal to, equal to, greater than, greater than or equal to, and not equal. The assembly jump opcodes are JUMP*x*, SKIP*x*, and CAM*x*, where

x is one of the following letters: L (less than), LE (less than or equal to), E (equal to), GE (greater than or equal to), G (greater than), or N (not equal to).

If the stated condition is satisfied, the computer takes an action. Thus, conditional jumps involve three variables: the two objects compared and the location of the next instruction. However, an assembly instruction has only two variable positions, the ac and the operand. Because no room exists for a third variable, the program must imply either the jump location or one of the compared values. The JUMP and the SKIP serve this purpose; they are implied arithmetic compares. The JUMP always compares the contents of the ac with 0; the SKIP always compares the contents of the operand with 0. If the JUMP compare is satisfied, the computer treats the operand as a label referencing the next instruction. If the SKIP compare is satisfied, the computer skips the next instruction. Whether the compare is satisfied or not, SKIP will place the contents of the operand into the ac (just like a MOVE). (Accumulator 0 is the one exception; in this case, the MOVE is not performed.) CAMx compares the contents of the ac with the contents of the operand. If the condition is satisfied, the computer skips the next instruction. For all three jumps, failure to satisfy the specified condition results in execution of the next instruction as in a high-level language. (See Table 3-1.)

Adding the suffix A to a jump statement means "always." Thus, JUMPA means "jump always," treating the operand as a "go to" label. The SKIPA and CAMA always skip the next instruction. (As mentioned above, SKIPA will also perform a MOVE.) If the jumps have no suffixes, then the computer ignores the statements (except SKIP, which it will treat as a MOVE).

3.2 Where to Go and How to Get There

The following is an IF/THEN statement in high-level language.

```
IF A > B THEN GOTO THERE
```

TABLE 3-1. Jump Scorecard

Opcode	*Value*	*Compare to*	*Action if satisfied*
JUMPx	*ac*	*zero*	*"go to" operand*
CAMx	*ac*	*operand*	*skip next inst*
SKIPx	*operand*	*zero*	*skip next inst*

(*Remember:* SKIP affects accumulators.)

Below is an assembly language version of this statement.

```
MOVE 2,A
CAMLE 2,B
JRST THERE    ;goto 'there' if A is greater than B
              ;here if A is less than or equal to B
```

Or you could rewrite the condition in the following form.

```
IF (A - B) > 0 THEN GOTO THERE
```

The assembly version would then look like this:

```
MOVE 2,A
SUB 2,B
JUMPG 2, THERE
```

The opcode SUB directs the computer to subtract the contents of the operand from the contents of the ac, placing the difference in the ac. To create the equivalent of an IF/THEN/ELSE statement, simply add a JRST ELSE after the JUMPG in the above example.

If the THEN is not a "go to" statement but several statements,

```
IF A > B THEN statement1 . . . statementN
STATEMENT AFTER IF
```

your assembly version will be

```
          MOVE 2,A
          CAMG 2,B
           JRST SKIPTHEM
           statement1
           . . .
           statementN
SKIPTHEM: STATEMENT AFTER IF
```

FORTRAN has an arithmetic IF similar to the SGN function in BASIC.

IF (expression) label1, label2, label3

If the value of the expression is negative, label1 references the next executable statement. If the value of the expression is 0, the computer uses

label2; if the expression is positive, label3 is the reference. You can accomplish the same thing in assembly by the following statements (assuming that ac contains the value of the expression).

```
JUMPL AC, LABEL1
JUMPE AC, LABEL2
JRST LABEL3      ;if here, ac must have been positive
```

Another conditional used in high-level languages is the ON/GOTO.

ON expression GOTO label1, . . . , labelN

If the value of the expression is 1, then label1 references the next executable statement; if the value is 2, then label2 references the next statement, and so on. Assuming that ac contains the integer value of the expression and location ONE contains a 1, the following is one way assembly can execute an ON/GOTO.

```
SUB AC,ONE
JUMPE AC, LABEL1
SUB AC,ONE
JUMPE AC,LABEL2
...
SUB AC, ONE
JUMPE AC,LABELN
                ;here if expression not in range 1 to n
```

3.3 Loops

Once a language has the ability to jump, it can repeat calculations by looping. Two types of loops are possible: those that repeat a calculation a fixed number of times, like FOR/NEXT or DO; and those that repeat a calculation an indefinite number of times, like WHILE/DO or REPEAT/UNTIL. In assembly programming, you can loop a definite number of times by placing the number of repetitions into an accumulator and then executing the repetitive calculations. The computer then subtracts 1 from the accumulator on each repetition and jumps back to the beginning of the calculation every time the result is not 0. The preceding ON/GOTO example subtracted and jumped. Because the combination "subtract 1 and jump" occurs frequently, the opcode SOJx is designed for this purpose. The computer reads this statement as "subtract 1 and jump if the result satisfies condition x, where x is an L, LE, E, GE, G, or an A." Similarly, AOJx is

an instruction to add 1 and jump on condition x, and a third opcode using SKIP is also available. Skips deal with operands, not accumulators. Thus, AOSx is an instruction to add 1 to the contents of the operand and skip if the result satisfies condition x. SOSx subtracts 1 from the contents of the operand and skips if the result satisfies condition x. In a FOR/NEXT statement the terminating condition NEXT is at the end of the loop. Below is an assembly example of a FOR/NEXT loop.

```
      MOVE AC, NUMBEROFTIMES   ;for
LOOP: statement
      ...
      SOJN AC, LOOP            ;next
```

A WHILE statement is an example of a loop that continues an indefinite number of times. For example, if you want the computer to accept characters from the terminal, specifying A as the last character it should accept, you can type any number of characters before you type A.

```
...
CHAR:   "A"
...
        PBIN%           ;initialize while condition
WHILE:  CAMN 1, CHAR
        JRST DONE
        ...             ;body of loop
        PBIN%           ;change while condition
        JRST WHILE
DONE:   ...
```

The WHILE loop establishes the condition before the loop calculation. In a REPEAT/UNTIL loop, the condition appears after the calculation.

3.4 A Program Using Jumps

Let us look at a simple program using jumps that will make the computer function much like a hand-held calculator. The program on the next page will accept an integer number and any of the following operations: addition (+), subtraction (−), multiplication (*), and done (=). If you type in 3+4*2+5=, the answer will be 19 (3 plus 4 is 7, times 2 is 14, plus 5 is 19). Here is the algorithm for this problem:

```
set operation to plus
repeat
  get a number
  do operation
  get new operation
until operation is =
```

In the following example, the opcode IMUL multiplies integers.

```
                TITLE EXAMPLE 8
                SEARCH MONSYM
                RADIX 10
                ANS=4
                OP=7
        TTYIN:  .PRIIN
        TTYOUT: .PRIOU
        CLEAR:  0
        BASE:   10
        ADD:    "+"
        SUB:    "-"
        MUL:    "*"
        EQ:     "="
        START:  MOVE OP,ADD             ;set op to plus
;REPEAT
        GETNUM: HRROI 1, NUMMESS        ;send message
                PSOUT%
                MOVE 1, TTYIN           ;set up NIN%
                MOVE 3, BASE
                NIN%
                  ERJMP OOPS
        DOOP:   CAMN OP, ADD            ;is op an add
                  JRST PLUS             ;yes
                CAMN OP, SUB            ;no, then is it sub
                  JRST MINUS            ;yes
                CAMN OP, MUL            ;no, then is it mul
                  JRST TIMES            ;yes
                                        ;if here then invalid operation
                HRROI 1, BADOP
                PSOUT%
                PBIN%                   ;try again
                MOVE OP,1
                JRST DOOP
;
```

```
      PLUS:           ADD ANS, 2          ;number is in ac 2
                       JRST NEWOP
      MINUS:           SUB ANS, 2
                       JRST NEWOP
      TIMES:          IMUL ANS,2
      ;
      NEWOP:          HRROI 1,OPMESS
                PSOUT%
                PBIN%                     ;get new op
                MOVE OP,1                 ;put it in op
;UNTIL
                       CAME OP,EQ
                        JRST GETNUM
      ;
      SHOWIT:          MOVE 1, TTYOUT     ;set up NOUT%
                       MOVE 2, ANS
                       NOUT%
                        ERJMP OOPS
                       HALTF%
                        MOVE ANS, CLEAR   ;clear ans
                        JRST START        ;restartable
      NUMMESS:  ASCIZ/
TYPE IN A NUMBER/                         ;note enclosed carriage return
;
      OPMESS:   ASCIZ/ TYPE IN AN OPERATION/
      ;
      BADOP:    ASCIZ/
INVALID OPERATION, RETYPE/
      ;
      OOPS:     HALTF%
                END START
```

This program is restartable. When the computer executes a HALTF%, it returns to the TOPS-20 level of the system. If you type a CONT (continue) command, the program will continue at the line after the HALTF%. In the above example, the line after the first HALTF% clears the answer, and the next line restarts the program. If an I/O error occurs and the program stops because of the OOPS: HALTF%, you cannot use a CONT because no executable code is present after that HALTF%. END is a nonexecutable command used by the assembler program.

If you were to write the same example in a high-level language, you could use a CASE statement to determine the operations. Your CASE set would then be [+, −, *, =]. The CASE statement is similar to the ON/GOTO except that the CASE set need not be in a particular order. In the ON/GOTO example, the expression's value had an order—a 1, 2, or 3—which determined the label to be used in the jump. The elements in the CASE

set could be apples, oranges, or other things that do not necessarily have mathematical relationships. You can't subtract a 1 from an orange, for example, and get an apple. Thus, you establish an order by placing the elements in contiguous memory locations. The next challenge is to retrieve the objects, which we will discuss in the next chapter.

3.5 Summary

This chapter shows how conditional and unconditional JUMP statements permit a certain amount of flexibility in the flow of a program. Once a language has the ability to jump, it can repeat a number of calculations by looping, for example repeating a calculation a fixed number of times or an indefinite number of times. All of the high-level language conditional statements and loop statements can be expressed using assembly statements. One of the milestones in high-level language development (ALGOL) was that the IF/THEN/ELSE statement forced the compiler rather than the programmer to select labels. In assembly, however, the programmer must label all jumps to locations. This chapter also introduces opcodes that perform two functions, for example AOJX, an opcode more powerful than an ADD followed by a JUMP.

3.6 Exercises

1. Accept an integer via NOUT%. If the integer is positive, display the integer using NOUT%. If the integer is negative, do not NOUT% the integer; instead PSOUT% the phrase NEG.
2. Use NIN%s to accept two integers and PBIN% to accept an operation. If the operation is a + or −, perform the appropriate operation and NOUT% a result; otherwise, PSOUT% an error message.
3. PBIN% a character. PBOUT% the character only if it is an A or B. Place the above in a loop and terminate the loop after the fifth PBIN% or if the letter C is PBIN%ed.
4. Accept two integers using NIN%s. NOUT% the largest of the two integers. (Make this program restartable.)
5. Set up a loop that will accept five random integers from the terminal. Design the program to NOUT% the position of the largest integer after all the integers are typed in.
6. Accept an integer via NIN%. Considering only positive integers, PSOUT% a phrase stating whether the integer is EVEN or ODD. (*Hint:* Repeat subtractions of 2.)

CHAPTER

4

Addressing Modes

So far we have concentrated on opcodes. Now we will discuss another important part of assembly language, the addressing mode, which determines the value of the operand. In earlier chapters, we used the direct addressing mode. In the direct mode, the operand is the address of the value to be used with the contents of the ac and the opcode. Other ways to determine the value of the operand are also available for the DECSYSTEM-20, including the immediate, indirect, and index modes. These addressing modes give us greater flexibility; thus, your standard assembly statement now takes the following form:

opcode ac, E

where E stands for the effective address of the operand.

4.1 Immediate Addressing

The direct addressing mode loads an ac with a number in the following manner:

```
Value:   4
...
MOVE AC, value
```

The immediate mode accomplishes the same thing in one line:

```
MOVEI AC, 4
```

Here the operand E is the value to be used in the assembly instruction. The suffix I on the opcode establishes the immediate mode. Opcodes we have used so far that allow the suffix I are ADD, SUB, MOVE, IMUL, and IDIV. For example, to add 1 to the contents of an accumulator, use the following form:

```
ADDI AC, 1    ; or AOJ AC,
```

The only jump instruction that has an explicit immediate mode is the compare, CAI*x*. Thus, to compare the contents of an accumulator with a plus sign and to program a skip instruction if the two elements are not equal, write:

```
CAIN AC, "+"
```

4.2 Index Addressing

Index addressing is very similar to specifying subscript variables in a high-level language. In a high-level language, the expression A(I) represents the Ith element in the A collection. To find the A(I) element, the program adds the value of I (offset) to the address of A (base). This method of offsetting a base works only if the elements in the collection are in contiguous memory locations. A high-level language sets aside the necessary memory locations with a statement like DIM A(20) or DIMENSION A(20). In assembly language, an indexed address statement has this form:

opcode ac, base(offset)

The parentheses signify index addressing. The offset should be an accumulator, and the base, if present, is generally a memory location. The computer finds the effective operand for indexing by adding the contents of the offsetting accumulator to the numerical value of the base symbol. For example, the following places a 5 into ac 2.

```
HERE:   ANYTHING
        5
...
MOVEI 4, 1      ;put a 1 in ac 4
MOVE 2, HERE(4)
```

A statement (word) fills one memory location in the computer. The first line (HERE: ANYTHING) places the value of ANYTHING in a computer word and calls the word HERE. The next word is simply a 5, which goes into the next memory location, "HERE + 1." Later the program immediately loads ac 4 with a 1. The indexing step—"MOVE 2, HERE(4)"—calculates the operand by finding the contents of the indexing register (the contents of ac 4 is 1) and adding the contents to the base ("HERE"), yielding the operand "HERE + 1."

At this point, the addressing is the same as direct addressing. The program moves the contents of "HERE + 1" into ac 2. You can use the assembly directive BLOCK to reserve contiguous memory locations. BLOCK 6 saves the next six memory locations in a program, for example. The following program accepts five characters from the terminal, stores the characters via indexing, sends a carriage return to the terminal after the fifth character, and then presents the five characters in reverse order. The OUT loop uses the indexing accumulator to count the number of elements.

```
        TITLE EXAMPLE 9
        SEARCH MONSYM
        RADIX 10
HERE:   BLOCK 5
CR:     ASCIZ/
/                           ;carriage return
START:  MOVEI 4, 5          ;number of char
        MOVEI 2, 0          ;clear ac 2
NEXT:   PBIN%               ;get char
        MOVEM 1,HERE(2)     ;save char
        ADDI 2,1            ;increment ac 2
        SOJG 4,NEXT         ;more chars?
        SUBI 2,1            ;no, but counted too many
        HRROI 1, CR         ;send carriage return
        PSOUT%
OUT:    MOVE 1, HERE(2)
        PBOUT%
        SOJGE 2,OUT         ;any more
        HALTF%
         JRST START         ;restartable
        END START
```

4.3 Indirect Addressing

In direct addressing, the operand points to the data; that is, the operand is the address telling the computer where to find the data. In indirect addressing, the operand points to an address that contains another address that points to the data. The high-level expression A(B(I)) serves a similar

purpose. The computer uses the offset, I, to find an element in the B collection; it then uses the B(I) element to find an element in the A collection. The symbol @ signifies indirect addressing. An indirect way of placing a 3 into ac 1 follows:

```
        ...
here:   there
        ...
there:  3
        ...
        MOVE 1,@here
```

The following program demonstrates how indirect addressing can act like indexing.

```
        TITLE EXAMPLE 10
        SEARCH MONSYM
        RADIX 10
LIST:   48
        49
        50
        51
        52
START:  MOVEI 2, LIST      ;address of list in
                           ;ac 2, not the 48
        ADDI 2, 4          ;now ac 2 contains
                           ;address of list+4
        MOVEI 3, 5         ;ac 3 will be a counter
OUT:    MOVE 1, @2
        PBOUT%
        SUBI 2, 1          ;next element
        SOJG 3, OUT
        HALTF%
        END START
```

What does this program do? The assembly directive EXP can simplify this program by directing the assembler to place those quantities delimited by commas into separate memory locations. Thus, you can place the five list elements above on one line as follows:

```
LIST:   EXP 48,49,50,51,52
```

4.4 Calculating Effective Addresses

Combinations of the addressing modes are also possible to further enrich the assembly language. For example:

opcode ac, @base(offset)

In order to calculate an effective address, we must set up a hierarchy among the addressing modes. First, look for indexing. If parentheses are present, evaluate the quantity inside the parentheses to determine the referenced accumulator. The partial effective address is then the contents of the indexing accumulator. If the program does not use indexing—that is, it contains no parentheses—then the partial effective address is 0. Second, add to the partial effective address the value of any symbol or numerical constants present in the operand—for example, the value of the symbol base. Third, if the program uses indirect addressing, use the present partial effective address as an address. Go to that address, start over at step one, and proceed through all three steps again. Repeat these three steps until you find no indirect addressing. The final partial effective address is the one to use with the original opcode and ac that started the calculation. ALWAYS CALCULATE THE EFFECTIVE ADDRESS FIRST. THEN PERFORM THE OPCODE. (See Figure 4-1.)

The following example uses RADIX 10 and the solidus notation to specify location/value; thus, 400/4 indicates that location 400 contains a 4.

```
RADIX 10
OFFSET=2
BASE=400
1/400
2/50
51/5
52/1
400/4
450/2
```

Operand	*Value*	*Effective Address*
(offset)	0 + (2) = 50	50
@offset	@2	50
@base	@400	4
@base(offset)	@400(2) =	
	@ < 400 + 50 >	2
@1(offset)	@1(2) =	
	@ < 1 + 50 >	5
@offset(offset)	@2(2) =	
	@ < 2 + 50 >	1
@base(2)	@400(2) =	
	@ < 400 + 50 >	2

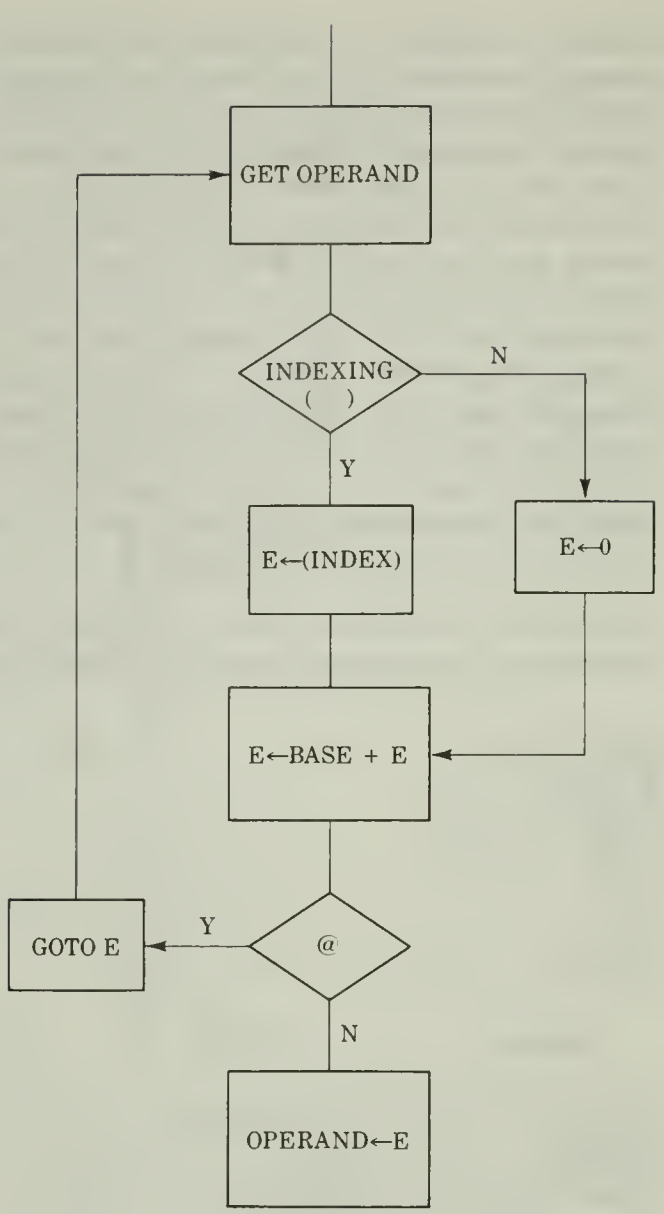


Figure 4-1. Calculating the Effective Address

This example shows that the instruction "opcode ac, @base(offset)" is equivalent to "opcode ac, 2." Note that referencing the first 16 memory locations is the same as referencing the accumulators. (The next chapter will give an example of indirect addressing referencing indirect addressing.)

The assembler can also understand arithmetical expressions within a statement:

opcode ac + 2, @ base −2 (offset * 3)

Using the above values, with ac = 4 and radix 10, this statement is equivalent to the statement below.

opcode 6, @ 398(6)

The arithmetical operations permitted are addition (+), subtraction (−), multiplication (*), and division (/). Nesting operations are permitted, but you must use angle brackets, <>, rather than parentheses, which signify indexing. The following example uses nesting:

opcode <ac + 3> * 2, @ < base + 4 > /2(offset + 5 * < offset − 1>)

Again, using the above values and ac = 4, the statement is equivalent to the statement below.

opcode 14, @ 202(7)

4.5 Opcode Function Suffixes

As we discussed earlier, the flow of an operation is normally from right to left. However, the trailing M in MOVEM is an example of an opcode function suffix that changes the flow from left to right. The opcodes we have used so far that allow the trailing M are ADD, SUB, MOVE, IMUL, and IDIV. The instruction "ADDM ac, operand" adds the contents of ac to the contents of the operand and places the result in the operand location. You can use the M suffix with all modes of address but the immediate mode. For example (using the above values), you could write the following:

```
ADDM 3, @ BASE
```

This instruction adds the contents of accumulator 3 to accumulator 4 and places the results in accumulator 4. Don't forget to find the effective address first, before you perform the opcode. You may use other letters in opcodes, but you will find no simple pattern for their use. (See Appendix B on opcodes.) One opcode worth noting is MOVN*x*, where N stands for negative. The *x* is generally an I, meaning immediate. The following operation will *not* move a negative 3 into ac.

```
MOVEI AC, -3
```

This MOVN operation will:

```
MOVNI  AC, 3
```

To understand this it is necessary to examine machine language, a subject for the next chapter.

4.6 Summary

Before any serious programming can be performed, a language must have flexible methods in order to reference data. Try writing a program in a high-level language without subscript variables! In assembly language opcodes alone will not suffice for the execution of serious programming. A family of addressing modes is required to determine the value of these opcodes. There are four addressing modes available in the DECSYSTEM-20—direct, immediate, index and indirect. Combinations of these addressing modes are also possible. To learn other addressing modes consult a text on Digital's LSI-11 or Motorola's 6809.

In the next chapter we will be learning how the computer translates the source code—opcodes and the various address modes—into machine code.

4.7 Exercises

1. What are the addressing modes of the DECSYSTEM-20? Write a one-line example of each mode.
2. Draw a flowchart showing how to calculate an effective address in the DECSYSTEM-20.
3. Write a program that will add the same two integers four times using a different addressing mode each time.
4. Code the following high-level language notation into assembly language. (Assume the array A contains 4,2,9.)

```
FOR I = 1 TO 3
 SUM = SUM + A(I)
NEXT I
PRINT SUM
```

5. The following high-level language program will interchange array elements.

```
DUMMY = A(I)
A(I) = A(I + 1)
A(I + 1) = DUMMY
```

Write an assembly program that will interchange the second and third elements of the array (4,2,9) and then print the new array.

CHAPTER

5

Numbers, Radices, and Bits

5.1 Machine Code

Most computers are *digital computers*, rather than analog computers. The digital computer is based on an electronic switch that can be either on or off; because it has only these two positions or states, this switch is known as a *binary switch*. The mathematical symbols 1 and 0 can represent the on-off states inside the computer. For problems or questions with only two possible outcomes or answers, one switch can represent the answer. For example, a 1 can represent "yes" (on) and a 0 can represent "no" (off), or alternatively, a 1 can represent "true" and a 0 "false."

For problems with more than two possible answers, more than one switch is necessary. For instance, for a computer to allow choice among four possible colors—red, blue, white, or green—it would need two switches. The two switches could then represent the combinations 00, 01, 10, and 11, with off-off representing red, off-on representing blue, and so on. The more unique outcomes a problem has, the more switches required to represent each possible outcome. With each additional switch, the number of unique states doubles. Thus, three switches allow eight unique patterns

(000, 001, 010, 011, 100, 101, 110, 111), twice as many as those allowed by two switches. Similarly, four switches produce 16 patterns (2 times 8). The number of unique patterns or states is always 2 raised to the number of switches.

The patterns of 1's and 0's can also represent numbers. A computer with three switches can represent eight unique numbers. The natural choice (but not the only choice) is for the patterns to represent the numbers 0 through 7. Because the switches have two possible positions, the preferred radix of a digital computer is a power of 2. Thus, digital computers usually work in octal (8), or hexadecimal (16), but not decimal (10) since 10 is not an integer power of 2. In order to read machine code, the binary representation of a program, you will need a good understanding of binary-related bases.

5.2 Radix N

This section discusses how to represent a number in different bases. The best way to understand bases is to reexamine base 10. Base 10 has ten unique symbols (0 through 9). To represent numbers larger than 9, the basic numbers are *concatenated* (linked). For example, we can represent the number 123 as follows:

$$1 \times 10^{**}2 + 2 \times 10^{**}1 + 3 \times 10^{**}0$$

where ** means "raised to the power." The value of 10**2 is the symbol 100, which in base 10 is one hundred. The symbol 100 guarantees that the third and only the third digit will be affected. Similarly, the value of the symbol 10**1 is 10, which in base 10 is ten. The symbol 10 guarantees that the second digit and only the second digit will be affected. Let's use the same reasoning to discuss base 8, or octal. Since base 8 has eight unique symbols (0 through 7), we can form a number larger than 7 by concatenating the basic symbols. Written in terms of base 8, the number 123 is:

$$1 \times 8^{**}2 + 2 \times 8^{**}1 + 3 \times 8^{**}0$$

In base 8, the symbol 100 represents the value of 8**2. This guarantees that only the third digit is affected, etc. To determine the decimal equivalent of the octal number 123, we must convert it to base 10. The above expression then represents the following number.

$$1 \times 64 + 2 \times 8 + 3 \times 1 = 64 + 16 + 3 = 83$$

Remember the following general points when working in different bases:

- The number of unique symbols in any base is the same as the value of the base. Decimal has ten unique symbols, and octal has eight, for example.
- The largest unique symbol in any base is 1 less than the base. Thus, the largest unique symbol in decimal is 9 and in octal is 7.
- The symbol 10 represents the base in any radix. In base 10, 10 is a ten; in base 8, 10 is an eight.

Consider the hexadecimal system, or base 16, which must have 16 unique symbols. Since we have only ten numerical symbols, we must take the other six symbols from the alphabet: A, B, C, D, E, and F. An A is a 10, an F is a 15. We can decipher the number 1A4 in this manner:

$1 \times 16^{**}2 + A \times 16^{**}1 + 4 \times 16^{**}0$

The hexadecimal version of the symbol 16**2 is 100, and the decimal equivalent is 256. The following expression represents the decimal value of 1A4.

$1 \times 256 + 10 \times 16 + 4 \times 1 = 256 + 160 + 4 = 420$

With this understanding of how numbers are constructed, let us now simulate a NIN% via PBIN%. Remember, when you type 123 into the terminal, the computer receives the ASCII equivalent of each digit. To understand how to convert the stream of ASCII characters to a number, first assume that the 1 is already in an accumulator called *temp*. The terminal now sends the ASCII of a 2 to the computer, which subtracts the ASCII offset (48) from the character yielding the number 2. Before adding this number to temp, the current contents of temp, 1, must shift left to the second-digit position. The computer accomplishes this shift by multiplying the contents of temp by the base in which the number is to be expressed. After this operation, the temp contains a 10 and is ready for the addition of 2; as a result, the temp now stores a 12. This process—subtracting the ASCII offset, multiplying the temp contents by the base, then adding the new digit—repeats for each character, even the first character! When does the process stop? The NIN% monitor call ends when a typed character is not in the base range ($0 \leqq$ char $<$ base).

If we were to use a WHILE loop to accept characters, the condition for doing the loop appears to be the following:

```
WHILE (CHAR > = '0' AND CHAR < BASE) DO
```

The one character that will not work with this condition is the carriage return—more specifically, the implied line feed that follows every carriage return typed in from the keyboard. A carriage return causes the print head (cursor) to return to the beginning of the line; its ASCII value is 13 (base 10). A line feed causes the print head to advance to the next line, and its ASCII value is 10. Because most applications require that the carriage return start a new line, the computer operating system program automatically sends both a carriage return (ASCII 13) and a line feed (ASCII 10) to the terminal when it receives the ASCII equivalent of a carriage return. Though the I/O buffer deals with one character at a time, it does store characters for use on the next PBIN%. Thus, if you terminate with a carriage return, you must clear the line feed from the I/O buffer (this step occurs automatically with NIN%). The following algorithm simulates a NIN% via PBIN%s:

```
get char
while char not carriage return do
  begin
       change char to num
       if num < 0 then goto done+1
       else if char >= base then goto done + 1
            else begin
                      multiply temp by base
                      add to temp num
                      get char
            end
  end
done: get line feed
      rest of program
```

Here is the assembly version of this routine:

```
          PBIN%
WHILE:    CAIN 1,13              ;carriage return
           JRST DONE             ;yes
          SUBI 1,"0"             ;subtract offset
          JUMPL 1, DONE+1        ;less than a 0
          CAML  1, BASE          ;less than the base
            JRST DONE+1          ;no
          IMUL TEMP, BASE
          ADD  TEMP, 1           ;add number in ac 1 to temp
          PBIN%                  ;next char
       JRST WHILE
DONE:     PBIN%                  ;get line feed
          ...
```

You must still limit the bases you use to 2 through 10. Do you know why? As you remember, the NOUT% monitor call converts a number in the computer to a stream of ASCII characters, which requires that the number be broken down into its individual digits. The radix in which the number is to be expressed determines the digits the computer will use. How would you send the octal equivalent of the hexadecimal value 1A4 to the terminal? The easiest way is to convert the number to decimal and then from decimal to the desired base. Essentially, then, to do base conversions, you must know how to convert from base 10 to another base.

Suppose you want to express the number 10, in base 10, in octal. The number is really 8 + 2, or (1 × 8**1) + (2 × 8**0), which is simply 12 octal. Thus, you break down the original number (10) into powers of the desired base (8). Let us convert the decimal number 68 to octal in the same manner:

68 = 64 + 4 = 1 × 8**2 + 0 × 8**1 + 4 × 8**0 = 104

To calculate the powers of 8 for a large number, you must continually divide the number by 8, collecting remainders, until the new quotient is no longer divisible by 8. For example, 68 divided by 8 is 8 with a remainder of 4. The 4 becomes the last symbol of the octal number 104. Now divide the quotient (8) by the base (8). Your answer is 1 with a 0 remainder. The 0 becomes the second symbol in 104. Again, divide the quotient (1) by the base (8). The new quotient is 0 with a remainder of 1, the 1 in 104.

Let us now go back to our original problem of finding the octal value of the hexadecimal number 1A4. 1A4 in decimal is 420. Continually dividing 420 by 8 yields 644. Use of base 10 is not sacred in performing conversions. The only condition for converting is to use the same base for expressing the two numbers and for dividing them. We might, for example, first express both numbers, the 1A4 and the 8, in hexadecimal and then divide the number using hexadecimal rules! However, we use base 10 because we are most familiar with the decimal system. The computer, on the other hand, is most comfortable with base 2 because of its binary nature. An algorithm for converting a number to a new base follows:

```
quotient gets number
while quotient not 0
        begin
          divide quotient by divisor
          save remainder
          quotient gets quotient
          count a digit
        end
```

You have probably noted that the first remainder in your division becomes the symbol farthest to the right in your new number, and the last remainder becomes the first symbol on the left. The left symbol prints out first and the right symbol last (a stack). It would be necessary to use a counter to keep track of the number of remainders (symbols). The problem with printing the remainder is similar to example 9.

In performing IDIV, necessary for base conversions, the assembler places the quotient in the ac associated with the opcode, but the remainder is in ac + 1, the next accumulator, whose original contents are lost when using IDIV. (The accumulators are cyclic; the first ac reappears after the last.) The simulated NOUT% program below converts the value in location NUM to the radix in location BASE.

```
        TITLE EXAMPLE 11
        RADIX 10
        SEARCH MONSYM
        QUO=5
        REM=6                           ;quo+1
        COUNTER=7
SAVE:   BLOCK 10                        ;remainders are saved here
BASE:   8
NUM:    68
START:  MOVEI COUNTER, 0                ;show at least one digit
        MOVE QUO, NUM
WHILE:  JUMPE QUO, DONE
        IDIV QUO, BASE
        MOVEM REM, SAVE(COUNTER)
        AOJA COUNTER, WHILE

DONE:   SUBI COUNTER,1
PRINT:  MOVE 1, SAVE(COUNTER)           ;get last digit
        ADDI 1, "0"                     ;ASCII offset
        PBOUT%
        SOJGE COUNTER, PRINT
        HALTF%
        END START
```

This simulated NOUT% program, like the simulated NIN%, will not work for a base larger than 10 due to the ASCII offset. For radices larger than 10, the letter A represents 10. The ASCII for 9 minus the ASCII for 0 yields a result of 9. But, the ASCII for A minus the ASCII for 0 does not produce an answer of 10! The result is 17 (base 10), which accounts for the seven ASCII characters between ASCII 9 and ASCII A. Thus, for the above exam-

ple to handle bases larger than 10, you must add the following lines before the line ADDI 1, "0"

```
CAILE 1, 9          ;need extra 7?
ADDI 1, 7           ;yes
```

5.3 Assembly Language and Machine Code

In this section, we will examine the relationship between a mnemonic assembly statement (opcode ac, operand) and the corresponding machine code (1's and 0's). Memory locations store patterns of 1's and 0's. The 1 or 0 is a Binary digIT and is referred to as a *bit*. Each memory location contains the same number of bits. At one time, the number of bits making up a computer word established computer classifications: an 8-bit system was a microcomputer; a 16-bit, a minicomputer; and a 32-bit or more, a mainframe computer. Because of progress in solid-state physics and microelectronics, however, classifying by word size is not always valid, and distinctions are blurring among micros, minis, and mainframes. The DECSYSTEM-20 has 36 bits in each word, allowing 2**36, or 6,719,476,736 different patterns of 1's and 0's!

Let us examine how to code assembly statements, opcode, ac and operand, into patterns of 1's and 0's that represent data and executable code. Systems with small word sizes generally need more than one word to code instructions. However, because of the number of bits, most DECSYSTEM-20 assembly statements need only one word per statement, which makes coding easier. The 36 bit positions that make up a word are numbered, starting on the left with the 0 bit and ending on the right with bit 35. A DECSYSTEM-20 assembly statement contains the following information: the opcode, the accumulator, and the operand, with its addressing modes. The one word of 36 bits must encode all this information (see Figure 5-1). The opcode uses the first 9 bits, bits 0 to 8, which allow for up to 2**9, or 512, opcodes. Bits 9, 10, 11, and 12 specify which of the 16 accumulators an assembly statement is using. (*Remember:* Four bits are necessary to represent 16 different items.) Bit 13 signifies indirect addressing. If @ is

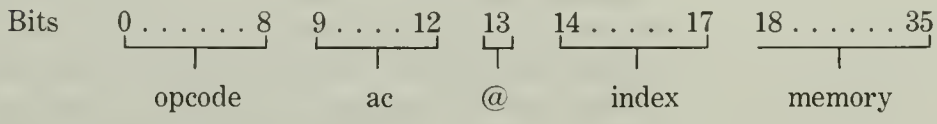


Figure 5-1. DECSYSTEM-20 Word

present in the operand field, then bit 13 is set (made equal to 1). When finding the effective address, the program examines bit 13 to see whether indirect addressing is present, and the search for the effective address continues until bit 13 is clear (has a value of 0). Index addressing uses an accumulator, and again, 4 bits are necessary to identify the indexing accumulator—bits 14, 15, 16, and 17. We now have accounted for the left half of the DECSYSTEM-20 word.

The right half of the word, bits 18 to 35, references the memory location to be used with an instruction. These bits identify the part of the operand that is not the symbol @ and not inside parentheses. It is thus the value of base in the operand, "@base(offset)," or simply the value of base in the expression "opcode ac, base." Since this half contains 18 bits, only 2**18, or 262,144 memory locations can be addressed. Let us look at the machine code equivalents of several opcode values.

Opcode	Machine Code
BIT	012 345 678
MOVE	010 000 000
ADD	010 111 000
ADDM	010 111 010

Below are machine code versions of several assembly statements (the mnemonics are in base 10):

```
MOVE 1,2            010000000 0001 0 0000 000000000000000010
ADD 3, 2(1)         010111000 0011 0 0001 000000000000000010
ADDM 12, @ 20(7)    010111010 1100 1 0111 000000000000010100
```

Octal is the preferred base for the DECSYSTEM-20 because it makes it easier to read code. (Some computers use base 16, but 36 is not divisible by 16.) Three binary bits are necessary to make an octal number; therefore, the octal machine code has 12 digits (36 divided by 3). Coding the same examples in octal yields the following:

```
MOVE 1,2            200040 000002
ADD 3, 2(1)         270141 000002
ADDM 12, @ 20(7)    272627 000024
```

Because the number of accumulators is not divisible by 3, the ac bits, indirect bit, and indexing bits overlap. Thus, the octal code is not as transparent as the binary code, making it more difficult to determine what accumulator is in use, for example, or whether indirect addressing is present. The example below is a program that reads itself and sends its machine code to the terminal.

```
        TITLE EXAMPLE 12
        SEARCH MONSYM
        RADIX 10
        COUNT=4
BASE:   8                           ;using octal
START:  MOVEI 1, .PRIOU             ;set up terminal
        MOVE 3, BASE
        MOVE 2, START-1(COUNT)      ;-1 so that base will
                                    ;be shown first
        NOUT%
         ERJMP DONE
        MOVEI 1, 13                 ;carriage return
        PBOUT%
        MOVEI 1, 10                 ;line feed
        PBOUT%
        CAME 2, DONE                ;last line?
        AOJA COUNT, START
DONE:   HALTF%
        END START
```

In this example, only 13 lines generate code, from BASE to the DONE. Execution of this program yields the following printout:

```
000000000010
201040000101
200140000140
200104000140
104000000224
320700000154
201040000015
104000000074
201040000012
104000000074
312100000154
344200000141
104000000170
```

Note that the monitor calls NIN%, PBOUT%, and HALTF% all have the same opcode, 104, which is a Jump to SYStem (JSYS). The right half of the word—

224, 74, 170—specifies the JSYS being called. The second line shows the value of .PRIOU as 101 (found in MONSYM). Either .PRIOU or 101 can reference the terminal printer. The address in the third line, the right half of the word, is 140. The mnemonics reveal this part of the code as the address of the location BASE. Octal 140 is in fact the normal location for the beginning of a program. Note that START is location 141 and that one counts 11 lines to the HALTF%. The HALTF% is referenced as location 154. How does 141 + 11 equal 154? The error here is that the DECSYSTEM-20 uses octal, and the line count is 13 octal lines, 141 + 13 = 154.

5.4 Macro

The TOPS-20 command MACRO generates a utility program that allows you to examine the machine code version of a program. Besides generating code, MACRO checks for assembly errors. It can create two files from the source file that you supply: the *relocatable file* (rel file) and the *listing file*. The *linker* uses the rel file when the program is executed. (It is possible to execute parts of programs from different files. It is the job of the linker program to form these program fragments into a workable unit.) The listing file contains the original source file, a machine code version of the source file, and the symbol table. You can invoke MACRO as follows: at TOPS-20, type MACRO followed by a carriage return. A star prompt (*) will appear. The command line for MACRO then takes this form:

*relocatable, listing = source/switches

The characters you type before the comma become the name of the rel file. The characters you type between the comma and the equal sign become the name of the listing file. The source file name appears after the equals sign. MACRO assumes extensions REL, LST, and MAC so you need not type them. If you do not desire these extensions, then add the desired extension after the name of the file—for example, JUNK.BLA. The default mode of the generated machine code separates the bits into opcode bits (0 to 8), ac bits (9 to 12), indirect bit (13), index bits (14 to 17), and memory (18 to 35). Though not true machine code, this format is easy to read. Including the switch /G will make all 36 bits continuous. For further information about the switches, type /H, and a help file will print out. Type the following to print out a listing of example 12:

```
*,EXAM12 = EXAM12
```

(Because file names must be no longer than six characters, we call the file EXAM12.MAC.) MACRO would look for a file EXAM12.MAC and create a file EXAM13.LST. Since no file name appears before the comma, MACRO would not create an EXAM12.REL file. If all is well, a "no error" message will appear (if the program contains errors, the printout will list them and you will need to return to the editor to make corrections). Finally, you must type CONTROL-Z to get out of MACRO and use the TOPS-20 TYPE command, TYPE EXAM12.LST, to obtain a copy of the list file. The following example illustrates this process. (Due to space limitations, comments were removed before sending example 12 to MACRO; a real program would contain comments.)

```
EXAMPLE 12      MACRO %53A(11    52) TIME DATE PAGE 1
                                00100       TITLE EXAMPLE 12
                                00200       SEARCH MONSYM
                                00300       RADIX 10
             000004             00400       COUNT=4
000000'  000000  000010         00500    BASE:      8
000001' 201 01 0 00 000101      00600    START: MOVEI 1,.PRIOU
000002' 200 03 0 00 000000'     00700           MOVE 3, BASE
000003' 200 02 0 04 000000'     00800           MOVE 2,START-1(COUNT)
000004' 104 00 0 00 000224      00900           NOUT%
000005' 320 16 0 00 000014'     01000            ERJMP DONE
000006' 201 01 0 00 000015      01100           MOVEI 1,13
000007' 104 00 0 00 000074      01200           PBOUT%
000010' 201 01 0 00 000012      01300           MOVEI 1,10
000011' 104 00 0 00 000074      01400           PBOUT%
000012' 312 02 0 00 000014'     01500           CAME 2, DONE
000013' 344 04 0 00 000001'     01600           AOJA COUNT,START
000014' 104 00 0 00 000170      01700           DONE: HALTF%
          000001'           01800               END START
NO ERRORS DETECTED
PROGRAM BREAK IS 000015
CPU TIME USED 00:00.599
60P CORE USED
EXAMPLE 12      MACRO %53A(1152) time date Page S-1
EXAM13  MAC     DATE TIME               SYMBOL TABLE
BASE            000000'
COUNT           000004
DONE            000014'
ERJMP   320700  000000  int
HALTF%  104000  000170  int
NOUT%   104000  000224  int
PBOUT%  104000  000074  int
START           000001'
 .PRIOU         000101  SIN
```

The first six digits on each line represent the relocatable address in which the machine code is stored. The apostrophe signifies that the address is relocatable. For the problem above, the linker will reassign locations by adding an offset (generally 140). For example, 000001′ references the START label. When the program was executed, START had an address of 141. The symbol table shows the program labels (for example, base 000000′) and the symbols defined by the monitor (for example, HALTF%).

If you do not want a permanent listing of the file, you can type this command:

```
*,TTY: = EXAM12
```

It will send the LST file directly to the terminal. The following command designates the terminal as both source and destination.

```
*,TTY: = TTY:
```

This command is a quick way to check machine code. If you will be using labels and symbols, type PASS1 after the star prompt and follow with a carriage return and any standard assembly expression, such as I=3. After you have typed all symbols, type PASS2 after the star. If you do not plan to use labels, you can skip PASS1. Once you are in PASS2, after you have typed an assembly line and a carriage return, the computer will immediately generate the machine code for that line and display it on the TTY.

5.5 Negative Numbers

The computer stores numbers in memory locations in binary form. Since memory locations contain a finite number of bits (36 bits in the DECSYSTEM-20), a memory cell has a ceiling on the largest number it can hold. For example, if only 4 bits were available, then the largest binary number would be a 15. The number of bits also restricts the number of different numbers that the memory word can represent. A 4-bit word can represent only 16 unique numbers, one possible representation being the numbers 0 to 15.

What about negative numbers? In arithmetic, negative numbers are as useful as positive numbers. Given a finite number of symbols, half the symbols could represent positive numbers and the other half, negative numbers. A 4-bit word would then allow eight symbols for positive num-

bers 0 to 7, leaving the other eight symbols (8 to 15) to represent negative numbers (-1 to -8) in some way. Consider a three-digit automobile odometer, for example. If you clear the odometer and then drive 1 mile, it will read 001. If you clear the odometer and drive in reverse for 1 mile, it will not read -1 but 999. If you drive the car backward 2 miles, the odometer will read 998, representing -2. We refer to positive-looking symbols (998) that represent negative numbers (-2) as *signed numbers*. When treated as positive numbers, they are called *unsigned numbers*.

In any base, we can represent -1 by the largest unsigned number in the base, since the largest symbol is 1 less than 0—that is, adding 1 to the largest number equals 0 ($-1 + 1 = 0$)! (If the odometer reads 999 and you drive one more mile, it will then read 000.) One problem with this representation is the possibility of overflow. If half the numbers on our 1000-number odometer are to be negative numbers, then the largest positive number the odometer can represent is 499. If we add 1 to 499, we would expect a result of 500, but since 500 to 999 are negative numbers, we cannot get a positive sum of $499 + 1$. The computer will catch this error and print an overflow error message (which we will discuss more later).

Let us now look at how to calculate the signed representation of a number. If we assume a positive number A, we must solve the following equation to find its negative symbol:

$$A + \text{symbol} = 0$$

Algebraically, the solution is

$$\text{symbol} = -A$$

However, we must take into account the finite number of digits. With a fixed number of bits, the largest unsigned number represents a -1. On our odometer, this number is 999. Given this fact, we can proceed as follows:

$$\begin{aligned} A + \text{symbol} &= 0 \\ A + (\text{symbol} - 1) &= -1 \\ \text{symbol} - 1 &= -1 - A \\ \text{symbol} - 1 &= 999 - A \\ \text{symbol} &= 999 - A + 1 \end{aligned}$$

Thus, to determine the negative value of 123, for example (-123), we perform this calculation:

symbol = 999 − 123 + 1
symbol = 876 + 1
symbol = 877

Since the representation of A −1 is the largest number (999), we will never borrow when doing subtraction. The result of each digit subtraction is the complement of the digit. Two numbers are *complements* if, when added, they equal the largest digit in the base—for example, 0 and 9, 1 and 8, 2 and 7, and so on. The simple way to find a negative number is to form the complement of the number (using all digits, even leading 0s) and add 1. Using a three-digit, base 10 representation, the following table shows the negative values of two numbers.

Number	Complement	Add 1	Symbol
−21	978	979	979
−423	576	577	577

The above process finds the representation of a number such that a 0 results when the number is added to itself. We can start with a negative representation and find its magnitude using the same method. (Negative representation plus magnitude yields 0.)

Symbol	Complement	Add 1	Magnitude
979	020	021	21
999	000	001	1

The process is the same in the binary system; in fact, it is even simpler. The only two characters, 0 and 1, are complements of each other. The following table works with a 5-bit binary number.

Number	Complement	Add 1	Symbol
110	11001	11010	11010
1010	10101	10110	10110

You can easily recognize a negative binary number by the 1 in the left-most bit; positive numbers always have a 0 in the left-most bit. Because the left bit provides these clues, it is called the *sign bit*.

Converting negative numbers to magnitude follows the same process we used above.

Symbol	Complement	Add 1	Magnitude
11011	00100	00101	101
10010	01101	01110	1110

A few examples using three-digit octal numbers will help in working with the DECSYSTEM-20. Remember that the largest octal digit is 7; therefore −1 is 777, and complements are 0 and 7, 1 and 6, 2 and 5, and 3 and 4.

Number	Complement	Add 1	Symbol
21	756	757	757
357	420	421	421
140	637	640	640

The following examples convert negative numbers to positive numbers:

Symbol	Complement	Add 1	Magnitude
757	020	021	21
421	356	357	357
640	137	140	140

Now that we have unique representations of negative numbers, subtraction becomes addition!

$$A - B = A + (-B)$$

where we replace the symbol (−B) with its signed complement and add it to A. Assuming a three-digit, base-10 word, we can perform the following calculations:

$$55 - 21 = 055 + 979 = 034 = 34$$
$$8 - 21 = 008 + 979 = 987$$

You might expect the first example to produce an answer of 1034, but because we have no room for the 1, the correct answer is 34! This homeless 1 is a *carryout*, and we ignore such carryouts when doing signed arithmetic. The second example produces a negative result, 987. To find the magnitude, first determine the complement (yielding 012) and then add 1 (013).

Now that we understand how negative numbers are formed we can explain the problem with MOVEI AC, −1. This does not place a minus 1 into ac. The immediate mode of addressing deals only with bits 18 to 35, the right half of a memory word. Any information in the left half of the operand is not used and is replaced by 000000! A −1 in octal is 777777777777, representing 1's in all 36 bits. The left half contains information. A MOVEI AC, −1 would load ac with 000000777777, not −1. The instruction MOVEI AC, 1 works because the word 1 is 000000000001, with the left half containing 000000.

A final note on word size and precision. Do not assume that word size determines the largest number the computer can produce (precision). The word size simply limits the number that a single memory word can hold. We can produce larger numbers by concatenating memory words. Thus, a ten-digit number could use ten memory locations; each word holds one digit. The programmer must keep track of all carrying, digit loading, and so on. Overflow occurs only in the memory location storing the most significant digit.

5.6 Summary

This chapter has examined how source code (mnemonics) is converted into machine code (1's and 0's). To understand this process it is necessary to learn other radix systems, especially base 2. Because the DECSYSTEM-20 has such a large word size (36 bits), opcodes, accumulator, and effective operand information can be expressed conveniently in just one word (we have not discussed extended addressing). However, other computers with smaller word sizes generally require more than one word to code instructions.

MACRO allows for examination of the machine code version of an assembly program, checks for assembly errors, and has the capability of creating a relocatable file and a listing file from the supplied source file.

Words take on meaning through controlling, setting, and clearing bits. In the next chapter we will examine such operations on bits.

5.7 Exercises

1. How many bits are there in a DECSYSTEM-20 word? Explain the part each bit plays in coding the general instruction "label: opcode ac, operand ;comment."
2. Convert the following numbers to the radix requested:
   (a) 10110111 (binary) to decimal
   (b) 7435 (octal) to decimal
   (c) 143 (decimal) to octal
   (d) 143 (decimal) to binary
   (e) 1A4 (hexadecimal) to decimal
3. Assume a word size of 6 bits in the following questions and calculations.
   (a) What is the largest positive binary number represented by this word size?

(b) What is the largest negative binary number possible?
(c) Express 10 and −15 in binary.
(d) Calculate (10 − 15) in binary.
(e) Convert the answer to d to decimal.

4. Hand code the following program into machine language. (Check your work by using MACRO.)

```
        EXAMPLE 13
        SEARCH MONSYM
HERE:   BLOCK 5
CR:     ASCIZ/
/
START:  MOVEI 4,5
        MOVEI 2,0
NEXT:   PBIN%
        MOVEM 1, HERE(2)
        ADDI 2,1
        SOJG 4, NEXT
        SUBI 2,1
        HRROI 1, CR
        PSOUT%
OUT:    MOVEI 2, HERE
        MOVE 1, @2
        PBOUT%
        HALTF%
         JRST START
        END START
```

5. Encode the following program.

```
000000' 000000  000012
000001' 000000  000104
000002' 200100  000000'
000003' 271140  000065
000004' 201200  000000
000005' 322140  000011'
000006' 200244  000001'
000007' 302240  000041
000010' 274160  000002'
000011' 104000  000170
```

6. Write a table that will make an ASCII table of capital letters and their hexadecimal values.

```
A       41
  ...
Z       7A
```

CHAPTER

6

Logical Operators and Shifts

In a digital computer, electronic switches are the *hardware* that perform all operations. The computer recognizes the 1's and 0's of machine code as voltage levels (typically 5 volts and 0 volts). You can think of the hardware as a series of black boxes with input and output lines. On the input lines are voltages representing machine code. The *logic gates* inside a black box—representing AND, OR, and NOT conditions—determine the relationship between the input and output patterns. Electronic logic gates use truth tables, which are essentially binary logic tables.

6.1 Logical Operations

AND, OR, and NOT logic gates govern virtually all computer operations, even arithmetical operations, through manipulation of bits. NOT is a *unary* operator (one input and one output). The following is a truth table for NOT:

in	out
0	1
1	0

NOT is generally known as COMplementing. The NOT or COM replaces each 1 with a 0 and each 0 with a 1. The COM of 0110 is 1001. The AND and OR are binary operators (two inputs and one output). Below are AND and OR truth tables.

in1	in2	AND
0	0	0
0	1	0
1	0	0
1	1	1

in1	in2	OR
0	0	0
0	1	1
1	0	1
1	1	1

The result of 1 AND 0 is 0. The result of 1 OR 0 is 1. If we read 1 as "true" and 0 as "false," we can interpret these logical operations very much as we would their English language equivalents. For example, a compound sentence using the conjunction AND is true only if each clause is true, while a compound sentence using OR is true if either clause is true. A binary operator has four output states, each of which can be 1 or 0. Therefore, we have 16 possible output patterns (0000 to 1111), all of which we can construct using AND, OR, and NOT logic operators. Because of this not all 16 patterns are assigned unique operator names. Patterns that are used quite frequently have unique names. NAND and NOR, meaning NOT AND and NOT OR, for example, are complements of the operators. XOR is an exclusive OR. Thus, OR is generally referred to as Inclusive OR, IOR. (DECSYSTEM-20 recognizes either OR or IOR.) EQV, the equivalent operator, is the complement of XOR. Below are truth tables for these common operators.

in1	in2	AND	NAND	OR	NOR	XOR	EQV
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

As mentioned above, all patterns can be constructed from the operators AND, IOR and NOT. The XOR operator is equivalent to the "complement of in1 ANDed with in2, the result of this is IORed with in1 ANDed with the complement of in2"—that is,

in1 XOR in2 = ((NOT in1) AND in2) IOR (in1 AND (NOT in2))

You can use the logical operations to set bits (make them 1's) or clear bits (make them 0's). A bit AND 0 will clear it. One way to make a number

even is to clear its least significant bit (LSB). If the target number is in an ac, then

```
zero:   0
...
AND ac, zero
...
```

will set the LSB to 0, along with all the other bits! Thus, ZERO is the wrong *mask*, something used to set and/or clear bits in another word. Instead you need a word of 1's in all bits except for bit 35, which should have a 0. The directive .B will change the radix on one line; it instructs the assembler to read what follows the .B as a binary number, independent of any RADIX directive. (Two similar directives, .O and .D, direct the assembler to read the next symbol in octal and decimal, respectively.) The following will clear the LSB without changing the other bits in ac.

```
LSB:    .B11111111111111111111111111111110
...
AND AC, LSB
...
```

You can simplify mask construction by using a complementing operation.

```
LSB:    1
...
ANDCM AC, LSB
...
```

This opcode complements the operand (memory) and then ANDs it to ac. To clear bits 33 and 35, use

```
MASK:   .B101   ; or MASK: 5
...
ANDCM AC, MASK
...
```

Use the IOR operation to set specific bits. The statements below will set bits 32 and 35 without changing the other bits in ac.

```
MASK:   .B1001;   or MASK: 9
...
IOR AC, MASK
...
```

Using the same MASK,

```
IORCM AC, MASK
```

guarantees all bits but 32 and 35 are set. (The value of bits 32 and 35 will not necessarily be 0; their values will remain what they were before the operation.) To set the ac to a specific pattern, you could use a MOVE operation; however, when dealing with logic relations, use the opcode SET*x* (set to).

```
PATTERN:  some ones and zeros
...
MOVE AC, PATTERN
```

is the same as

```
SETM AC, PATTERN
```

This statement means "set the ac to the value in PATTERN." (The M in SETM does not make the flow from left to right as in MOVEM.) The following copies ac to HERE, set to the value of the ac the memory.

```
SETAM AC, HERE
```

We will mention two more SET*x* instructions (see Appendix B for other logic opcode modes). To clear a location, use

```
SETZ AC,        ;set the ac to zero
...
SETZM  HERE     ;set here to zero
```

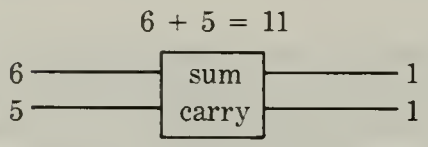
To fill a location with 1's, use

```
SETO AC,        ;set the ac to ones
...
SETOM HERE      ;set here to ones
```

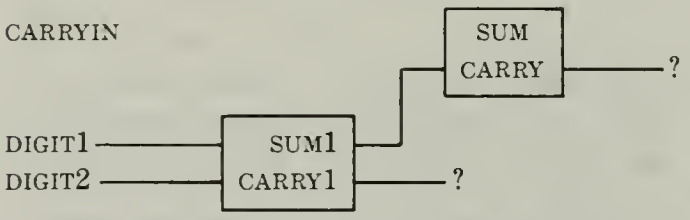
The reader is encouraged to glance at the opcode table in Appendix B and examine the various logic opcode modes.

6.2 Arithmetic and Logic

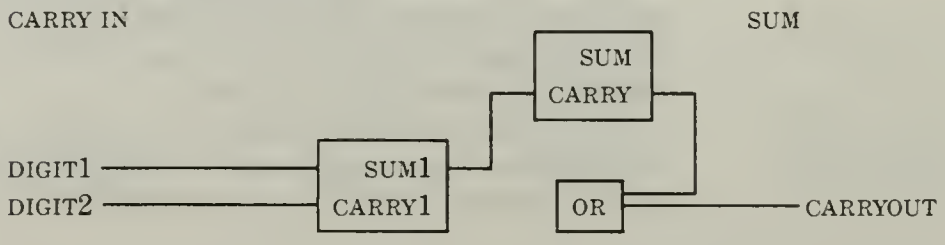
Addition using logic operations is analogous to a black box with two inputs and two outputs. Each input line will have a single digit, and the black box will add the digits and place the sum on one of the output lines and a possible carry on the other output line. For example, when adding 6 and 5, the result is a sum of 1 and a carry of 1.



In general, addition involves three digits—the two digits you are adding plus a possible previous carry. The previous carry is called a *carryin*. For example, let's add 26 and 35: first, add the 6 and 5, yielding a 1 plus a carry. Next, add the 2 and 3, yielding a 5. Then add 5 to the previous carry, yielding 6. The final answer is 61. To find the second digit of the answer, we performed two additions involving the three digits 2, 3, and 1. Because of the carryin, two black boxes, or *half-adders*, are necessary, as shown in the figure below.



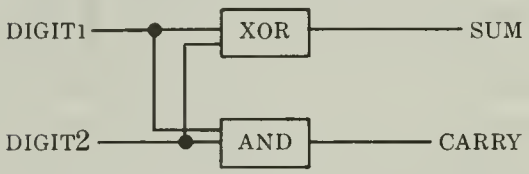
Adding SUM1 of the digits to the carryin gives the sum of the three digits. What about a possible resulting carry, a carryout? No matter what radix you are working in, adding pairs of digits and a possible carry will never generate a carry greater than 1. Consider the base-10 problem of 9 plus 9 plus a carry. The answer is 9 plus a carry of 1. Carries are either 0 OR 1. A carry can come from adding digits (for example, 8 plus 4), OR from the digit SUM1 and a carryin (for example, 9 plus 1), but never from both places. Below is a *full adder*.



The logic gate OR is part of the full adder. The binary addition table below helps illustrate the half-adders in the black boxes.

Digit1	Digit2	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

This table shows that the result of adding a binary 0 to a binary 1 is a sum of 1 and a carry of 0. As we saw in the table of logic operators in this chapter, the sum column above is the same as an exclusive OR, XOR, and the carry column is the same as an AND. A half-adder consists of an exclusive OR and an AND gate:



An assembly version of the full adder follows.

```
SETM SUM1, DIGIT1       ;put digit1 in first half-adder
SETM CARRY1, DIGIT1
XOR SUM1, DIGIT2        ;form sum of digit1 and digit2
AND CARRY1, DIGIT2      ;form carry

SETM SUM, SUM1          ;put first sum in second half-adder
SETM CARRY, SUM1
XOR SUM, CARRYIN        ;add previous sum1 and carryin
AND CARRY, CARRYIN      ;get resulting carry
IOR CARRY, CARRY1       ;or the carries
```

To add an N bit binary number to another N bit binary number, N full adders are necessary—in other words, a full adder for each bit. Given this reasoning, we would need N versions of the above program in assembly! (Yes, easier ways do exist.) Binary addition, and therefore any radix addition, is possible using logic gates.

To subtract, the computer needs a way to form a signed number. As you recall, to form a signed number you must take the complement of the number and add 1. Binary system complementing occurs via a NOT gate:

the complement of a 1 is 0 and vice versa. To form the negative value of an N bit number, you must use a NOT gate for each bit and then add 1. The assembly version follows.

```
SETCM AC, NUM      ;not each bit
ADDI AC, 1         ;add 1
```

The accumulator now has the negative of the number in NUM. As you learned in Chapter 5, subtraction becomes addition upon generation of the signed representation.

What about multiplication? Multiplication is essentially repetitive addition. But *shifting* provides an alternative. To multiply 5 times 10, we could add 5 to itself 10 times, but we generally put a 0 on the right of the number. The original number then shifts one position to the left ($5 \times 10 = 50$). In any radix, shifting a number to the left is the same as multiplying the number by the radix. In binary, shifting a number one place to the left is the same as doubling the number. Assembly language has three ways to shift the contents of an accumulator: logical, arithmetical, and rotational shifts. The sign of the effective address determines the direction of the shift, which can be either to the left or to the right. The magnitude of the effective address determines the number of bits shifted. The three shifts control what happens to the vacated bit position when the bit pattern in a word shifts. A Logical Shift (LSH) fills the vacated bit position with a 0. The arithmetical shift (ASH) to the right copies the sign bit (bit 0) into bit 0 and bit 1. The Arithmetical Shift to the left fills bit 35 with a 0, and bit 1 is lost. This method prevents negative numbers from shifting into positive numbers. The sign bit never changes with an arithmetical shift. The final shift, a ROTational shift (ROT), is a cyclic operation that "connects" bits 0 and 35. Left rotational shifts copy bit 0 into bit 35; right shifts copy bit 35 into bit 0. No bits are lost in a rotational shift; it simply rearranges them. Table 6-1 illustrates these shift patterns.

TABLE 6-1. Shifting Table

Shift	*Left (fill bit 35 with)*	*Right (fill bit 0 with)*	*Example*
Logical	0	0	LSH AC, AMT
Arithmetical	0	previous 0th	ASH AC, AMT
Rotational	bit 0	bit 35	ROT AC, AMT

The following will multiply the contents of NUM by 4:

```
ASH NUM, 2      ;shift left 2 positions
```

Thus, (00010) becomes (01000). To divide the contents of NUM by 8, use

```
ASH NUM, -3
```

in which case (01101) becomes (00001). Note that a 3-bit remainder is lost: 24/8 = 25/8 . . . = 31/8. MASKing NUM with a 7, binary 111, is the first step in finding the remainder.

```
SETM REMAINDER, NUM     ;copy NUM to remainder
ANDI REMAINDER, 7       ;mask off all
                        ;but the last
                        ;three bits
ASH NUM, -3             ;divide by 8
```

The quotient is in NUM, and the remainder is in REMAINDER. Because the effective address is always calculated first and that value does the shifting,

```
MOVEI 2,  3
ASH NUM,  (2)
```

is the same as "ASH NUM, 3." (Examples of LSH and ROT will come later.)

Several tricks are possible when you multiply by a nonbinary number. To multiply by 15, note that 15 is 16 − 1, so NUM*15 is the same as "NUM*(16 − 1) = NUM*16 − NUM."

```
SETM AC, NUM
ASH  AC, 4          ;multiply by 16
SUB  AC, NUM        ;subtract num
```

In base 2, one cannot hope to survive on tricks, however; a more general algorithm is necessary. The normal way to multiply is to start with the right-most bit and to right justify each partial product with the multiplying bit. For example,

```
100
101
---
  100
   0
100
-----
10100
```

You can also start with the left bit and then left justify.

```
100
101
---
100
 0
  100
-----
10100
```

Every bit of the multiplier contributes a partial product, which is either 0 or a copy of the multiplicand. Each partial product shifts left before the addition to find the product. A multiplication algorithm follows.

```
ans: = 0
bit: = value of left-most bit of the multiplier
for i: = 1 to numberofbitsinaword do
begin
  shift ans 1 position to the left
  if bit is 1 then add multiplicand
  shift multiplier one to the left
  bit: = value of left-most bit of the multiplier
end
```

The following assembly version multiplies 5 by 15.

```
        TITLE EXAMPLE 14     ;multiplying via shifting
        SEARCH MONSYM
        RADIX 10
        ANS=5
        COUNTER=6
        MULP=7               ;multiplier ac
        MULC=8               ;multiplicand ac
START:  MOVEI COUNTER,36     ;number of bits
        MOVEI MULP, 5
        MOVEI MULC, 15
        SETZ ANS,

FOR:    LSH ANS, 1
        SKIPG MULP           ;test sign bit
          ADD ANS, MULC
        LSH MULP, 1
        SOJG COUNTER, FOR
        MOVEI 1,,PRIOU       ;set up tty
        MOVEI 3, 10
        MOVE 2, ANS
        NOUT%
         HALTF%
        HALTF%
        END START
```

The bit doing the multiplying always shifts to the zero-bit position, which is also the sign-bit position. If the zero-bit bit is set, then the computer can treat the number as negative. If the zero-bit is cleared, then it can treat the number as positive. Testing of the bit was done using those facts. The interested reader can try division.

6.3 ASCII Bit Packing and Unpacking

Packing in programming means placing more than one piece of information in a memory word. As you learned earlier, you can send a message to the terminal by packing the message via ASCIZ and then setting up the PSOUT% with the unimaginative HRROI opcode. In this section, we will simulate the PSOUT% via PBOUT%s and use a shift opcode to accomplish the unpacking. For example, to unpack the message

```
WORD: ASCIZ/ABCDE/
```

recall that each character uses 7 bits. In location WORD, the letter A uses bits 0 to 6, the letter B uses bits 7 to 13, and so on. PBOUT% sends the right-most 7 bits (29–35) in accumulator 1 to the terminal. The program below uses LSH with an index-effective addressing mode to allow use of a loop to shift and PBOUT%. Note that the statement has no RADIX directive; therefore, octal is the assumed radix. Recall also that ^D is an assembly directive that tells the assembler to treat the following number as a decimal number. This directive supersedes any RADIX directive, but only for the number following ^D.

```
        TITLE EXAMPLE 15     ;unpacking via LSH
        SEARCH MONSYM
WORD:   ASCIZ/ABCDE/
START:  MOVNI 2,^D36         ;ac 2 get decimal -36
        MOVEI 3,5            ;5 characters in WORD
OUT:    ADDI 2,7             ;each shift is 7 less
        MOVE 1, WORD
        LSH 1,(2)            ;shift character into
                             ;bits (29-35)
        PBOUT%
        SOJN 3, OUT          ;if not 0 get next char
        HALTF%
        END START
```

Let us now repeat the problem, using a ROT shift, which is more efficient than LSH for this example.

```
        TITLE EXAMPLE 16     ;unpack via ROT
        SEARCH MONSYM
WORD:   ASCIZ/ABCDE/
START:  MOVEI 3,5            ;5 chars
        MOVE 1, WORD
ROUT:   ROT 1,7              ;rot bits (0-6) to
                             ;(29-35)
        PBOUT%
        SOJN 3, ROUT
        HALTF%
        END START
```

Now that we have an idea how to unpack a location, let us examine packing. The assembly directive ASCIZ will pack for us, but the directive B is also worth noting. For instance, the following example places an octal 33 right justified at bit 9.

```
BEXAM:  33B9
```

The directive valueBn logically shifts the "value" in front of the B to the *n*th bit, where *n* is always a decimal (0–35). The following example illustrates one of the common uses of the B directive:

```
CRLF: 15B6 + 12B13
```

This statement places a 15 right justified at bit 6 and adds it to a 12 that is right justified at bit 13. The bit positioning prevents overlap between the 15 and the 12. If we were to unpack this message, a carriage return (ASCII 15) would result, followed by a line feed (ASCII 13). One more example:

```
WORD: 101B6 + 102B13 + 103B20 + 104B27 + 105B34
WORD:   ^D65B6 + ^D66B13 + ...
```

Except for the trailing null word, this example is the same as WORD: ASCIZ/ABCDE/. Below is an algorithm for packing characters from the terminal.

```
get char
while char not carriage return do
  begin
        shift char
        or it with previous chars
        decrement shift
  end
```

Because parentheses designate indexed addressing, use angle brackets to signify arithmetic grouping. Thus, 3*<4 + 2> is the same as 3*6. An assembly version of packing follows.

```
      TITLE EXAMPLE 17          ; packing chars
      SEARCH MONSYM
MESS1:ASCIZ/type in up to 4 chars end with a cr/
WORD: Z                         ;same as BLOCK 1
CR:   15                        ;ASCII of carriage return

       BYTE=7                   ;seven bits to a char
START:HRROI 1, MESS1
      PSOUT%
      MOVEI 4, ^D<36-BYTE>      ;ac 4 get a decimal 29
GETEM:PBIN%
      CAMN 1, CR
       JRST LF                  ;if a cr then get
                                ;line feed
      LSH 1,(4)                 ;shift ch to left
      IORM 1, WORD              ;put ch in memory
      SUBI 4, BYTE              ;decrement amt shifting
      JRST GETEM
```

```
LF:    PBIN%                    ;get the line feed
       HRROI 1, CRLF            ;send a cr lf to terminal
       PSOUT%
       HRROI 1, WORD            ;send word back to
       PSOUT%                   ;terminal
       HALTF%
CRLF:  15B6+12B13
       END START
```

(*Note:* MONSYM defines a symbol BYTE, but the computer searches local symbol tables first; thus, it takes the meaning of BYTE from the program, not MONSYM.) Establishing the four-character limit guarantees a null byte in a word. PSOUT%, which unpacks the input message, needs a null for termination.

To pack or unpack more than five characters, another counter is necessary. In this case, we must increment the memory word after every five characters, as illustrated below.

```
MOVE 1, WORD(COUNTER)
```

The next chapter discusses opcodes for manipulating strings.

6.4 Data Base Bit Packing and Testing

In this section, we will look at a nonarithmetic application of bit manipulation. Assume we have a data base containing the following information about a group of people: male/female, married/single, dead/alive, and programmer/nonprogrammer. We could ask the following question of this data base: How many single, alive males can program? The program seeking this information would have to know how the data are assigned to each record (person). A high-level language could use a double-subscript variable, with the first subscript identifying a person and the second defining properties of the person. The array would be as follows.

base(person, property)

where property = 1 might refer to sex, property = 2 to marital status, and so on. The following would assign person1 as male, single, dead, and a nonprogrammer, for example.

```
base(person1,1) = 1   ;(male)
base(person1,2) = 0   ;(single)
base(person1,3) = 1   ;(dead)
base(person1,4) = 0   ;(nonprogrammer)
```

The following algorithm could then help us answer our original question.

```
counter: = 0
person: = 1
while not end of data base do
  begin
    if base(person,1) = 1
      then if base(person,2) = 0
        then if base(person,3) = 0
          then if base(person,4) = 1
            then counter: = counter + 1;
  get next person
end
```

This approach has two drawbacks, however. First, it uses too much computer memory space; each record property takes up a memory location. Second, the counting program contains many IF/THEN statements, which create a long program that is difficult both to read and write. Let us rethink the problem, remembering that we can use individual bits in a computer word. One word can contain a complete record (an individual), with bit 35 signifying sex, bit 34 signifying marital status, and so on. (This method is possible in some high-level languages.) To load a record, we use the logical command IOR.

```
SETZ AC,          ;clear the ac
IOR AC, ^B0101    ;set bits 33 and 35
```

(Though IOR AC, 5 would also work, the ^B notation is clearer.) Once the data are set, we can place them into the data base:

```
MOVEM AC, BASE(PERSON)
```

where base is the location of the first record and the ac referenced as person indexes into the data base. Now to count the number of single, alive, male programmers, we must form a mask representing the question:

```
MASK: ^B1001
```

and loop over the following code:

```
MOVE AC, BASE(PERSON)     ;get a record
CAMN AC, MASK
  ADDI COUNT, 1
```

The following mask will reveal the number of dead people in the data base (bit 33):

```
DEAD:    ^B0100
```

and then loop over the following code.

```
MOVE AC, BASE(PERSON)
AND AC, DEAD
SKIPE AC
  ADDI COUNT, 1
```

This example checks whether a bit is set. To see whether a bit is cleared, let us write the code that counts all the single people in the data base.

```
SINGLE:  ^B0010
...
MOVE AC1, SINGLE
MOVE AC, BASE(PERSON)
ANDCM AC1, AC
SKIPE AC1
  ADDI COUNT,1
```

ANDCM establishes the complement of the value in ac. If the bit is now set, we can identify that person as single.

An opcode is available to test bits. We will discuss only two versions of the many TEST opcodes, TDNE and TDNN.

```
TDNx AC, MASK
```

When x is E, the computer skips the next instruction if all the bits of ac referred to in the MASK are equal to 0. If x is N, the computer skips the next instruction if not all the MASKed bits of ac are 0 (at least one bit is a 1). For a single bit, TDNN instructs the computer to skip if the bit is set and TDNE instructs to skip if the bit is cleared. The following illustrates how to test if a person is single.

```
MOVE AC, BASE(PERSON)
TDNN AC, ^B010
ADDI COUNT, 1
```

The properties of the preceding data base are so far simple yes/no or set/clear conditions. Let us add the property of age. How many bits are necessary to keep track of age? Since each bit is a power of 2, 7 bits are adequate since they can represent ages from 0 to 127. Let us then use bits 25 to 31 for the age information. (If we had only this one multibit field, the right-most 7 bit would be a better choice.) Suppose we want to find out how many 18-year-olds (binary 10010) are in the data bank. We might try ANDing with the following mask:

```
AGE:     ^B100100000
```

However, this approach counts all ages using bits 28 and 30—for example, 19, 22, 23, and so on—which do not even include all ages greater than 17. To work with numbers, it is better to shift the number field to the right and use the arithmetical commands. To search for 18-year-olds, then, the age field must shift right four positions. We can use LSH since the sign bit isn't important. To consider only the age field, we can mask out all other bits with the following.

```
AGEMASK:     ^B11111110000
```

(This mask only makes sense if bits 0 to 24 contain information.) The following code will count the number of 18-year-olds in the data base.

```
MOVE AC, BASE(PERSON)
AND AC, AGEMASK
LSH AC, -4
CAIN AC, ^D18
 ADDI COUNT, 1
```

To find the number of males who are 18 years old or older, first test bit 35 to see if the record represents a male. If the record is a male, then check age as above using CAIL AC,18 rather than CAIN AC, 18.

You can determine the information in a record by looking at each bit to see if it is set or cleared. A rotational shift is one way to perform this test.

```
          BITTEST: 1
          ...
          MOVE TEST, BITTEST
          MOVE AC, BASE(PERSON)
          TDNE AC, TEST                ;bit cleared
           JRST MALE                   ;no
          PRINT FEMALE                 ;yes
          JRST ROT
MALE:     PRINT MALE
ROT:      ROT TEST, 1
          TDNE AC, TEST
           JRST MARRIED
          PRINT SINGLE
          JRST ROT1
MARRIED:  PRINT MARRIED
ROT1:     ...
```

However, a simpler way is available to find out what a record contains. We know that 17 combinations are possible, the 4 bits (32 to 35) and the age. The age of the person is not the challenge; rather, the other 16 possibilities are. Use bits 32 to 35 as a pointer into a *jump table* that signifies the messages to be printed. For example,

```
            MOVE AC, BASE(PERSON)
            ANDI AC, ^B1111    ;get just bits 32-35
            JRST MESSAGE(AC)
            ...
   MESSAGE  JRST MESS0000
            JRST MESS0001
            JRST   MESS0010
            ...
MESS0000: PRINT FEMALE,SINGLE,ALIVE,NONPROGRAMMER
MESS0001: PRINT MALE,SINGLE,ALIVE,NONPROGRAMMER
...
MESS1111: PRINT MALE,MARRIED,DEAD,PROGRAMMER
```

The extra jump allows for the possibility that the printed message may take up more than one memory location, in which case the indexing MESSAGE(AC) would no longer point to the correct message. The JRST commands take up only one word; therefore, indexing is not misaligned. (The alert reader will have noted that the JRST labels are more than six characters and that the first six characters are not unique! These labels must change to prevent all the JRSTs from pointing to the same location!) The word *print* is pseudocode. A way to deal with messages is still necessary. Byte pointers are the solution; we will discuss their use in the next chapter.

6.5 Summary

The setting and clearing of any bit in assembly allows for better utilization of a machine word by the program. Traditionally, high-level languages concentrated on portability and therefore sacrificed bit operations. Recently developed languages, like PASCAL, allow for indirect bit manipulation via the set data type. Even newer languages like ADA allow for more direct bit manipulation via low-level packages.

We have now covered the five basic operations of any computer language: input, output, data transfer, arithmetic, and logic. The operations so far have dealt with full words or bits. The next chapter discusses operations that work with user-defined groups of bits.

6.6 Exercises

1. Write a program to PBIN% a character. After the character is accepted, set bit 30 and PBOUT% the result. What happens when you PBIN% a capital letter? A digit? (*Note:* It will help if this program is restartable.)
2. Write a program to PBIN% a character. After the character is accepted, clear bit 29 and PBOUT% the result. Answer the questions in exercise 1.
3. Write a program that will accept an integer (radix 10); then examine each binary bit in the number. If the bit is set, print the word HIGH, followed by a carriage return. If the bit is not set, print the word LOW, followed by a carriage return. (This program is an example of a parallel-to-serial conversion.)
4. Write a program to PBIN% a character. Count the number of bits set in the word. If the number of bits is odd, set bit 28. If the number of bits is even, clear bit 28. (This program illustrates *even parity:* an even number of bits is always set, and bits to the left of 28 are not used when PBIN%ing. The parity bit is available for checking errors when transferring information over phone lines.)
5. Write a program that accepts a number from the terminal. If the number sets bit 30, then treat bits 30 to 35 as an octal number and display this value.
6. This chapter presented a logic equation that expressed XOR in terms of AND, IOR, and NOT. Verify this equation by accepting two binary words (NIN% using base 2) and perform the AND, IOR, and NOT operations yielding the XOR of the words. Display your result (NOUT%

using base 2). Using the same words, do a simple XOR and display the result. The results should be the same.

7. Accept a character from the terminal. If the character is a 1, 2, or 3, then index into the appropriate position in a jump table that points to locations that print ONE, TWO, or THREE.

CHAPTER

7

Transferring Data Using Pointers

In the last chapter we examined packing and unpacking, which deal with a contiguous group of bits. These bits come from or are placed in certain positions within a memory location. We learned that shifting is necessary to deal with the next group of bits. For the DECSYSTEM-20, a group of contiguous bits is a *byte*. (In microcomputers, a byte is always 8 bits.) ASCII is a 7-bit byte. In this chapter, we will examine some opcodes that help with byte packing and unpacking. These opcodes set up and use a pointer to reference a particular byte.

7.1 Byte Pointers

The assembly directive POINT deals with the three aspects of computer words we have been discussing: the size of the byte, its position within a location into which it is placed or copied, and the memory location referenced:

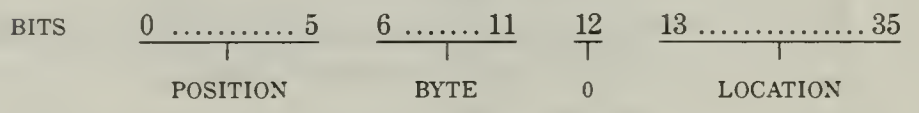
```
POINT BYTE, LOCATION, POSITION
```

(POSITION is optional, as you will see below.) You can think of POINT as setting up an imaginary pointer that points to the right-most bit of the

byte being referenced. The assembler encodes POINT as follows: LOCATION, which identifies the memory location of the string, uses bits 13 to 35 (indirect, index, and memory). The value of BYTE—interpreted as the decimal number of bits in the byte—is coded into bits 6 to 11. POSITION uses bits 0 to 5.

The user places in POSITION the decimal value of the bit that will contain the imaginary pointer. The assembler places in bits 0 to 5 the octal value of the number of bits to the right of the imaginary pointer. If the imaginary pointer is pointing to bit 3 (POSITION = 3), then the octal number 40 would be in bits 0 to 5 (32 bits are to the right of the pointer). Bit 12 is set to 0.

POINT Machine Code



Below is a standard use of POINT:

```
MESSAGE:    ASCIZ/strins/

PTR:        POINT 7,MESSAGE
```

Since no position value is present in POINT, the imaginary pointer defaults by pointing to the bit left of the 0 bit. Thirty-six bits are right of the pointer; therefore bits 0 to 5 contain octal 44. The instruction designates the byte size as 7; therefore, bits 6 to 11 contain a 7. Since the operand uses no indirect or index addressing, bits 13 to 17 are set to 0. Bits 18 to 35 contain the value of the symbol MESSAGE. The machine code in location PTR is 440700MESSAGE. Note that

```
POINT 7, MESSAGE,0
```

codes as 430700MESSAGE, while

```
POINT 7, MESSAGE
```

codes as 440700MESSAGE. To reference the first character of a string, the byte pointer must move to the right 1 byte. The opcode IBP (Increment the Byte Pointer) will effect this shift.

```
IBP PTR
```

PTR must be a location that contains a byte pointer. The IBP not only increments the position byte counter (bits 0 to 5), but also changes bits 13 to 35 to reference the next word once the original word referenced has no more room for a byte. IBP also resets the position counter to the first byte and changes the values in the location PTR. After the first IBP command, the value in PTR is 350700MESSAGE. After the sixth IBP, the value in PTR is 350700MESSAGE + 1. The opcode LDB (LoaD a Byte) will copy a PTR byte into an accumulator; the operand must be a byte pointer.

```
LDB AC, PTR
```

This command loads the byte to the left of the imaginary pointer into ac. We can combine incrementing and loading into one opcode.

```
ILDB AC, PTR
```

To copy a byte from an accumulator to a position within a memory location, use the opcode DPB (DePosit a Byte).

```
DPB AC, PTR
```

Again, we can combine incrementing and depositing in the opcode IDPB.

Given the combined forms, the computer always increments first and then loads or deposits. The example below uses pointers. Since incrementing changes the value of a pointer, we must save the original pointer so that the characters accepted from the terminal can be returned to the terminal. This example uses PSOUT%, which requires a pointer in accumulator 1, but without HRROI. (We will discuss HRROI further later.)

```
        TITLE EXAMPLE 18                ;byte pointer
        SEARCH MONSYM
WORD:   Z                               ;typed in chars go here
PTR1:   POINT 7,WORD
MESS:   ASCIZ/type in up to 4 chars
/                                       ;note, embedded cr
PTR2:   POINT 7,MESS
START:  MOVE 1, PTR2
        PSOUT%
        MOVE 2, PTR1
CHARIN: PBIN%
        CAIN 1,15                       ;was it a cr
         JRST LF                        ;yes, then get lf
        IDPB 1,2                        ;no, then store byte
        JRST CHARIN                     ;get next char
LF:     PBIN%                           ;get lf
        MOVE 1, PTR1                    ;reload pointer
        PSOUT%                          ;show chars
        HALTF%
        END START
```

Since location WORD is saving only one word, a problem arises if the user types more than four characters. After the computer receives the fifth character, the next incoming characters will start to replace the program code. (We set the limit to 4 so that a null is present for PSOUT%.)

The flow of data is not only between the program and the terminal, however. Remember that a program can rearrange the data within it. The following lines of code change all *e*'s in a string pointed to by PTR to *k*'s.

```
        MOVEI 2, "k"
        MOVE 3, PTR
LOOP:   ILDB 1, 3
        JUMPE 1, DONE       ;null char?
        CAIN 1, "e"         ;no, then is it an e
          DPB 2,3           ;yes, then change to k
        JRST LOOP           ;get next char
DONE:   ...
```

Byte pointers can simulate the fine string capabilities of BASIC as well. To count the number of characters in a string, for example, you can use the following instructions:

```
        SETZ LENGTH,
        MOVE 2, PTR         ;get a pointer
LOOP:   ILDB 1,2            ;get a char
        JUMPE 1, DONE       ;null char?
        ADDI LENGTH, 1      ;no, then count char
        JRST LOOP
DONE:   ...
```

To simulate the functions RIGHT$ or MID$, the opcode ADJBP (ADJust the Byte Pointer) is helpful. This opcode allows the computer to move the imaginary pointer anywhere within the string. To use ADJBP, you must place the relative byte offset in an accumulator and a pointer in the operand field.

```
MOVEI AC, 3
ADJBP AC, PTR
```

These lines will place in the accumulator a copy of PTR advanced 3 bytes. PTR remains unchanged, and negative numbers would back up the pointer.

Let us define a function RHT$ with two arguments, a string and a number. The number designates the starting positions of a new string so that

all characters from this position to the end of the original string move to the new string. The lines below demonstrate a RHT$. (*Note:* RHT$ is not the same as RIGHT$ in BASIC.)

```
A$="ABCDEFG"
B$=RHT$(A$,5)
After the above line is executed
B$ would be "EFG"
```

An assembly version of RHT$ follows.

```
            TITLE EXAMPLE 19        ;RIGHT$
            SEARCH MONSYM
SOURCE:     BLOCK 5                 ;save room for 25 chars
PTR:        POINT 7,SOURCE
MESS1:      ASCIZ/ type in a string /
PTR1:       POINT 7, MESS1
MESS2:      ASCIZ/ starting position /
PTR2:       POINT 7,MESS2
START:      MOVE 1, PTR1
            PSOUT%
            MOVE 2, PTR             ;this pointer has to
                                    ;be saved
LOOP:       PBIN%
            CAIN 1, 15              ;cr?
             JRST LF
            IDPB 1, 2
            JRST LOOP

LF:         PBIN%                   ;get lf
            MOVE 1, PTR2
            PSOUT%
            MOVEI 1, .PRIIN         ;set up NIN%
            MOVEI 3, 12             ;since RADIX not
                                    ;use, base is octal
            NIN%                    ;get starting position
             ERJMP ERROR
            SUBI 2, 1               ;an increment is done
                                    ;before char loaded
                                    ;therefore sub 1
            ADJBP 2, PTR            ;move byte pointer
            MOVE 1, 2               ;set up PSOUT%
            PSOUT%
            HALTF%
ERROR:      MOVE 1, PTRE
            PSOUT%
            HALTF%
PTRE:       ASCIZ * I/O ERROR *
            HALTF%
            END START
```

MID$ has three parameters: a source string, a starting position, and the number of characters.

MID$(string,start,numofchars)

MID$ functions as follows:

```
A$="ABCDEFG"
B$=MID$(A$,3,2)
After the above line is executed
B$ would be "CD"
```

MID$ is similar to RHT$ except that MID$ needs another counter to count the number of characters. You can print these characters via PBOUT%. (If you were to use PSOUT%, you would need to place a null character in the position "start + numofchars" via ADJBP.)

7.2 Literals

A byte pointer requires two locations: the location storing the directive POINT and the address of the data. *Literals* help the assembler pick these locations. The directive square brackets, [ ], requests a literal from the assembler. Consider the following lines, which do not use literals.

```
HERE:     ASCIZ/message/
PTR:      POINT 7, HERE
          MOVE 1, PTR
```

Using literals, the lines become

```
HERE:     ASCIZ/message/
          MOVE 1, [POINT 7, HERE]
```

The assembler chooses a location and places the bracketed code in that location. The value of this location then replaces the literal in the original instruction. You can nest literals also.

```
MOVE 1,[POINT7,[ASCIZ/message/]]
```

Literals are not restricted to pointers. For example, MOVEI AC, 3 yields the same result as MOVE AC,[3]. Literals are also helpful with monitor calls that return +1 or +2, as illustrated below.

```
        ...
NOUT%
  ERJMP [MOVE 1,[POINT 7,[ASCIZ /error message /]]
        PSOUT%
        JRST somewhere]
here if NOUT% successful
```

The "here if . . ." appears to be the fourth line after the NOUT% rather than the second line as required by the +2 monitor return. The reason is that the assembler places only the address of the literal next to the ERJMP and moves the rest of the code to the literal pool. The literal request thus allows the code to appear linear. One of the few restrictions for literal nesting is on breaking up an instruction; you cannot place the opcode and ac on one line and the operand on another. Also, MACRO will not show the literal pool unless you include the assembly directive LIT in the program. Only those literals before the LIT will appear. To have MACRO show all the literals, place the LIT immediately before END.

Another assembly directive that helps identify locations is #, which creates a variable symbol. Generally you should use a label to define any symbol you use. For example,

```
A:    Z
. . .
MOVEM 2, A
```

However, directive # makes it unnecessary to establish the symbol's location.

```
MOVEM 2, A#
```

The # informs the assembler to create the symbol A—that is, assign the name A to a location. Future references to A need not have the trailing directive #. As you can see, it makes identification of locations very similar to identification of variables in high-level languages.

7.3 Reading from the Terminal

PSOUT% is a glorified PBOUT%. RDTTY%, "read from the terminal," is more than a glorified PBIN%, however. RDTTY% permits line editor capabilities, for example, delete and control keys. RDTTY% is similar to a BASIC INPUT or FORTRAN READ statement. It uses three accumulators: Accumulator 1 holds a pointer indicating where the input string will be stored; in other

words, it sets aside an area of memory for incoming data. Accumulator 2 designates the maximum number of characters acceptable to prevent incoming data from overflowing the allotted storage area. (Accumulator 2 can do other things, as you will see.) RDTTY% ends either when the computer receives the maximum number of characters (not counting deletes or other line editor commands) or when the user types a carriage return. The carriage return and line feed also go into the storage space (if the count in ac 2 allows it). Accumulator 3 works in conjunction with the line editor control R. It contains a pointer, which references the message to be sent to the terminal when control R is used. If you do not want this feature, you must clear accumulator 3. RDTTY% uses a +1, +2 return. A simple setup of RDTTY% follows.

```
          SIZE=6
INPUT:    BLOCK SIZE              ;6 words or
                                  ;30 ASCII chars
PTR:      POINT 7, INPUT
          ...
          MOVE 1, PTR
          MOVEI 2, SIZE*5-1       ;leave room for null char
                                  ;used by PSOUT%
          SETZ 3,                 ;no control R facility
          RDTTY%
            ERJMP ERROR
```

Because of its line editor features, RDTTY% is the preferred way to accept input from the terminal. To input numbers from the terminal, RDTTY% works in conjunction with NIN%. NIN% accepts ASCII characters from a source and converts the ASCII string into a number. The contents of accumulator 1 determine the source of NIN%. Only when ac 1 contains .PRIIN is the terminal the source. In general, ac 1 should contain a pointer to the source. RDTTY% will place ASCII data into the program; NIN% will then take the ASCII data from the program and convert them to a number. This method of accepting numbers allows the user line editor capabilities. The program below accepts a base 10 number supplied by the user, then converts it to another base supplied by the user.

```
                TITLE EXAMPLE 20            ;base converting via RDTTY%
                SEARCH MONSYM
                SIZE=6
HERE:           BLOCK SIZE                  ;input string here
PTR:            POINT 7, HERE
BASE:           Z                           ;input base
START:          MOVE 1,[POINT 7, [ASCIZ/number?/]]
                PSOUT%
                MOVE 1, PTR                 ;set up RDTTY%
                MOVEI 2, SIZE*5-1
                SETZ 3,
                RDTTY%
                  ERJMP [ MOVE 1,[POINT 7,[ASCIZ/ error /]]
                                PSOUT%
                                HALTF% ]
                MOVE 1, [POINT 7, [ASCIZ/ new base/]]
                PSOUT%
                MOVE 1, [POINT 7,BASE]      ;set up RDTTY%
                MOVEI 2,4                   ;two digits and cr, lf
                RDTTY%
                  ERJMP [MOVE 1,[POINT 7,[ASCIZ/ base err/]]
                               PSOUT%
                               HALTF%]
                MOVE 1, [POINT 7, [ASCIZ / answer /]]
                PSOUT%
                MOVE 1, [POINT 7,BASE]      ;set up NIN% to
                MOVEI 3, 12                 ;get base
                NIN%
                  ERJMP [ MOVE 1,[POINT 7,[ASCIZ/ error /]]
                                PSOUT%
                                HALTF% ]
                MOVEM 2, BASE               ;save number
                MOVE 1, [POINT 7, HERE]     ;set up NIN%
                NIN%                        ;get number
                  ERJMP [HALTF%]            ;lazy
                MOVEI 1, .PRIOU             ;set up NOUT%
                MOVE 3, BASE                ;get base
                NOUT%
                  Z                         ;even lazier
                HALTF%
                JRST START                  ;restartable
                LIT                         ;show literal table
                END START
```

7.4 Comparing Strings—I

In this section, we will discuss how to store strings and how to search for a particular string. For the examples below, we will use the strings *aabc*, *acdda*, and *aa*. We will use RDTTY% to accept the strings, and will save

the pointer for each string before accepting it so that we can reference the string again. Below is one way to approach this problem.

```
        STR=6                       ;ac 6 is a counter
        COUNTER=7                   ;ac 7 counts num of strings
        NUMSTR=max number of strings
        BIG=NUMSTR*average num of char per string
HOLD:   BLOCK BIG/5                 ;all strings saved here
PTR:    POINT 7, HOLD
SAVE:   BLOCK NUMSTR                ;pointers to beginning
                                    ;of each string saved here
        ...
        MOVEI STR, 3                ;3 strings in
                                    ;this example
        SETZ COUNTER,
        MOVE 1, PTR                 ;set up RDTTY%
        MOVE 2, BIG
        SETZ 3,
LOOP:   MOVEM 1, SAVE(COUNTER)      ;save pointer
        RDTTY%
          ERJMP ERROR
        ADDI COUNTER, 1             ;count string
        SOJG STR, LOOP              ;any more?
        ...
```

The strings and pointers (shown by the dots below) are stored as follows.

```
HOLD:   .a  a   b   c   cr
        lf  .a  c   d   d
        a   cr  lf  .a  a
        cr  lf  0   0   0

SAVE:   440700 HOLD
        350700 HOLD+1
        170700 HOLD+2
```

If we were to use PSOUT%, we would need a null at the end of each string. The following adds a null and compacts the data by removing the carriage return and line feed.

```
          NULL=0
          MOVEI 5, NULL
          set up a RDTTY%
LOOP:     MOVEM 1,SAVE(COUNTER)
          RDTTY%
            ERJMP ERROR
          MOVNI 4, 1              ;put a -1 in ac 4
          ADJBP 4, 1              ;backup byte ptr
          MOVEM 4, 1              ;put byte ptr
                                  ;back in ac 1
          DPB 5, 1                ;put a null on
                                  ;top of cr
          ADDI COUNTER, 1
          SOJG STR, LOOP
```

The data and pointers now are as follows.

```
HOLD:   .a    a    b    c    0
        .a    c    d    d    a
        0     .a   a    0    0

SAVE:   44070 HOLD
        44070 HOLD+1
        35070 HOLD+2
```

Knowing how the strings are stored, let us find a specific string referenced by FIND.

```
FIND:     POINT 7, [ASCIZ/string/]
```

The following algorithm checks each string character by character. The last string typed in is the first string of the data bank checked.

```
nomatch:    get next string pointer
            set up FIND pointer
loop:       get char from FIND
            get char from data bank string
            if  findchar <>bankchar then goto nomatch
                else if findchar=0 then found
                        else goto loop
```

As soon as the characters are not the same, the computer retrieves the next pointer. When it finds a match, the computer checks the character to see if it is the null character signifying the end of the FIND string. The null character can also indicate the end of the data bank string because of

the match before the check. If a match exists but the character is not a null, then the computer will get the next character. The assembly version of the string check follows.

```
                MOVE STR, COUNTER       ;number of strings + 1
NOMATCH:        SOJL STR, NOTFOUND      ;any strings left?
                MOVE 1, SAVE(STR)       ;yes, then get pointer
                MOVE 2, FIND            ;get find pointer
LOOP:           ILDB 3,1                ;get a data char
                ILDB 4,2                ;get a find char
                CAME 3,4                ;are the chars the same
                  JRST NOMATCH          ;no
                CAIE 4, 0               ;null char?
                  JRST LOOP
                here if string found
```

With a pointer, the size and position stay in the left half of a word and the address generally stays in the right half. However, opcodes that "split" information into half words can help in manipulating data.

7.5 Half-Word Opcodes and Monitor Control Bits

Half-word opcodes, as suggested by their name, allow for half-word manipulation. The *left half word* contains bits 0 to 17, and the *right half word* contains bits 18 to 35. Half-word opcodes have the form H*sdfm*, where *s* and *d* signify the source and destination of the half word and can have value L (left) or R(right)—for example, HRR*fm*, HLR*fm*, HRL*fm*, and HLL*fm*. The *f* is optional, designating the possible effects to the contents of the other half word of the destination. The *f* position can contain no letter, meaning "no changes"; O, meaning "fill with 1's"; and Z, meaning "fill with 0's." (Another possible letter, E, is beyond the scope of our discussion). The *m* represents the addressing mode, determining the source and destination of the half word. It can be one of the standard letters: space, I, or M. The source is not always the operand; M indicates that the source is the accumulator, for example. Two assembly directives for half words are double comma (,,) and XWD. All three lines below represent the same instruction.

```
HALF: a,,b
HALF: XWD a,b
HALF: aB17+bB35
```

They all place *a* right justified in bits 0 to 17 and *b* right justified in bits 18 to 35 in the location HALF.

Consider the examples of half words in Table 7-1. Examples always begin with AC, which contains C,,D, and HALF, which contains A,,B. The 0 in the last example comes from the immediate mode of addressing, I, in which the left half of the operand always becomes 0.

Now we have enough information to discuss the opcode that sets up PSOUT%, HRROI. HRROI places all 1's in the left half of a word and an address of an ASCIZ string in the right half. PSOUT% is set up to accept this as an ASCII pointer. (No logical reason exists for this function of PSOUT%; it simply works!)

Half words also help set *control bits* for some monitor calls. The NOUT% monitor call uses control bits to format a number to be printed. It sets the control bits in the left half of accumulator 3. With no control bits, NOUT% will left justify a number without restricting the number of digits in the printing field (of course, the number must fit in an accumulator). Given this format, the user can pick the width of the field—that is, the maximum number of digits to be printed. The width is right justified in bits 11 to 17. Once a width is specified, the program will ignore digits that do not fit in the field. To prevent overflow, you must set bit 4, thereby expanding the field to the right if too many digits appear. To recognize overflow without expanding the field, you must also set bit 5. Too many digits will then cause stars (*) to print in the field as warning that the number is too large. Normally, a field justifies numbers on the left. To justify numbers on the right, you must set bit 2; this allows a choice for the leading fill character. If you set bit 3, the leading fill will be 0's; if you clear bit 3, the leading fill will be spaces. Two other control bits are not directly related to the width bits 11 to 17. To print an unsigned number, 777777777777 instead of −1, set bit 0. If you want a sign in all cases, set bit 1. If you set bit 1, a plus sign (+) will print out for positive numbers. The negative sign (−) will print as usual. (See Table 7-2.)

TABLE 7-1. Half-Word Examples

Code	*Result* *AC*	*Half*
HLL AC, HALF	A,,D	A,,B
HLLM AC, HALF	C,,D	C,,B
HLR AC, HALF	C,,A	A,,B
HLRM AC, HALF	C,,D	A,,C
HLLO AC, HALF	A,,777777	A,,B
HLROM AC, HALF	C,,D	777777,,C
HLLZ AC, HALF	A,,0	A,,B
HLRI AC, CB17+DB35	0,,C	A,,B

TABLE 7-2. NOUT% Control Bits (Left Half of AC 3)

Bit	*Symbol*	*Comment*
0	NO%MAG	print all numbers as unsigned
1	NO%SGN	show the plus sign
2*	NO%LFL	right justify
3*	NO%ZRO	leading fills are 0's (not spaces)
4*	NO%OOV	expand on overflow
5*	NO%AST	if bit 4 set, print * on overflow
11–17		field width

*These bits need data in bits 11 to 17.

The monitor defines the symbols NO%*xxx*, which you must generally use with literals. A few examples follow.

```
MOVE 3, [^D8,,^D10]          field width 8 char, radix 10
HRLI 3, 400000               print as an unsigned number
HRRI 3, ^D10                 decimal radix
MOVE 3, [NO%OOV+<^D9>B17]    field width 9, expand overflow
HRLI 3, 160012               field width 10, expand overflow,
                             right justify, fill with 0's
```

RDTTY% also has a number of control bits, which are in the left half of accumulator 2. (We will mention only a few.) As with a BASIC INPUT or a FORTRAN READ statement, the assembly program waits until RDTTY% ends—in other words, until the characters are in the input buffer. If set, bit 7 eliminates waiting, which is handy when you must minimize decision time or when a game calls for two players. Bit 5 monitors the use of the delete key. Setting bit 5 returns the user to the program if he or she tries to delete past the beginning of the new input. Setting bit 10 converts all lower case letters to upper case letters. Table 7-3 provides examples of RDTTY% control bits.

TABLE 7-3. RDTTY% Control Bits (Left Half of AC 2)

Bit	*Symbol*	*Comment*
5	RD%RND	trap if user tries to delete too much
7	RD%RIE	do not wait for carriage return
10	RD%RAI	convert all lower case to upper
	MOVE 2, [RD%RIE,,1]	allow for one character; do not wait if input buffer is empty
	HLL 2,[1B5]	trap if user deletes too much

7.6 Comparing Strings—II

In this section, we will compare machine words, five characters at a time. In the process, we will be able to review past chapters. We have four parameters for this approach to examining strings: NUMSTR, the NUMber of STRings in the data bank; AVELEN, the AVErage LENgth of a string; BYTE size; and BYTEMK, the BYTE MasK. By using BYTE as a parameter, we are not restricted to ASCII. The program setup follows.

```
        TITLE EXAMPLE 21          ;finding a string by
                                  ;comparing 5 chars at a time
        SEARCH MONSYM
        NUMSTR=3                  ;number of strings in data bank
        AVELEN=^O20               ;average length of a string
        BYTE=7                    ;bits in char
        BYTEMK=^B1111111          ;used to get one byte
        COUNTER=7                 ;ac 7 counts num of strings
        CPWORD=^O36/BYTE          ;chars per word
        NOCHARS=NUMSTR*AVELEN/CPWORD
HOLD:   BLOCK NOCHARS             ;all strings stored here
PTR:    POINT BYTE,HOLD           ;first pointer
SAVE:   BLOCK NUMSTR              ;data bank pointers
FIND:   POINT BYTE, LOOK4         ;string to be found
LOOK4:  BLOCK AVELEN
```

Once the user has supplied the three parameters (NUMSTR, AVELEN, and BYTE), the program calculates the necessary storage area. The next part of the program will accept the data bank strings from the terminal and store them in HOLD. Because the comparison proceeds one word at a time, the storage method must be different from the character-by-character method. A string must start at the beginning of a new word, which calls for initializing the pointer to the beginning of a word. To initialize, the program must increment the right half of the pointer, the address part, and then change the left half to 440700. Also, we cannot use a null to signify the end of a string as we did when working with single characters. Because the RDTTY% command keeps track of the number of characters in a string, we can use this number rather than a sentinel to count words and to help locate a particular string. Instead of examining all the strings in the data bank, the computer will examine only those strings that are the same length as the string sought. When it finds a string of similar length, it will compare the machine words of the string to the machine words of the string sought. Each match will reduce the number of char-

acters by 5, assuming ASCII. When the number is less than 0, the string ends.

Where will we store the number of characters in the string? Though we could use a separate block of memory to hold the length of each string, another way is to store the number at the beginning of the string! This method limits the number of characters in a string, for the binary value of the number must fit into one byte. For an ASCII byte (7 bits), this number would be 127 characters, which is not heavily restrictive. Now when the program compares the first word of a string with the first word of the sought string, it must compare two elements, the length of the strings and the first four characters! To store a number in the string, the byte pointer of the string increments once, leaving the first byte blank. After the user has typed the string and the computer has calculated the number of characters, the computer must set the pointer to the first location and deposit the number. How is the number of characters calculated? As RDTTY% accepts each character, accumulator 2 decreases by 1. The difference between the initial value of ac 2 and the value at the end of RDTTY% is almost the number of characters in the string (RDTTY% sets some control bits in the left half of ac 2 upon completion). The program must not consider control bits when subtracting. The instruction MOVNI 2,(2) enables it to ignore the control bits and still retrieve the desired number. The immediate mode, I, causes it to ignore the left half. Completion of this instruction places the negative value of the right half of ac 2 in ac 2, which the program then adds to the original number minus 2 (to avoid counting the carriage return and line feed). Backing up the pointer removes the carriage return and line feed; inserting a null allows PSOUT% to be used. Index offset addressing, SAVE + NUMSTR(COUNTER), eliminates one counter, setting COUNTER to the negative value of NUMSTR and incrementing it. This addressing ensures that the program will always reference the first pointer first, even in the searching. The following code loads the data bank strings and the string to be found.

```
START:      MOVE 1,[POINT7,[ASCIZ/
            type in 3 strings /]]
            PSOUT%
            MOVNI COUNTER, NUMSTR
            MOVE 1, PTR              ;set up RDTTY%
            SETZ 3,                  ;no ^R
```

```
INLDDP:     MOVEM 1, SAVE+NUMSTR(COUNTER)
            IBP 1                   ;make room for num of chars
            MOVEI 2, BYTEMK         ;max num of chars
            RDTTY%
              ERJMP [HALTF%]        ;lazy
            MOVNI 4, 1              ;backing up
            ADJBP 4, 1              ;pointer, taken out cr lf
            DPB 3, 4                ;put in a null
            MOVNI 2,(2)             ;find number of
            ADDI 2,BYTEMK-2         ;chars typed minus cr lf
            MOVE 1, SAVE+NUMSTR(COUNTER)   ;get original pointer
            IDPB 2,1                ;put number in first byte
                                    ;now initialize pointer to beginning of
                                     next word
            HRR 1, 4                ;get last address
            ADDI 1, 1               ;set to next address
            HRLI 1,440700           ;set to beginning
            AOJL COUNTER, INLODP    ;any more strings?

                                    ;getting the FIND string
OVER:       MOVE 1,[PDINT 7, [ASCIZ / find? /]]
            PSOUT%
            MOVE 1, FIND
            IBP 1
            MDVEI 2, BYTEMK
            RDTTY%
              Z
            MOVNI 4, 1
            ADJBP 4, 1
            DPB 3, 4
            MDVNI 2,(2)
            ADDI 2,BYTEMK-2
            MOVE 1, FIND
            IDPB 2,1
```

The next part of the program will compare machine words. In order to retrieve full words, this program will change the size of the byte to octal 44, using one pointer for the FIND string and another for a DATA string. After the program loads each original pointer in an accumulator, the instruction "HRLI AC, 444400" will change the pointer so that it will appear at the beginning (44) of a full word (44). If the program finds a match between the first word of a string and the first word of the desired string, it will place the number of characters in the string in a counter. The counter will decrease by CPWORD and check to see whether the string is complete. If not, ILDB will retrieve the next word. This process continues until the end of the string or a "no match" condition.

```
            MOVNI COUNTER, NUMSTR
            SUBI COUNTER, 1         ;an add is done before each
                                    ;fetch, sub to compensate
                                    ;for the first add
BEGINF:     MOVE 1, FIND            ;get pointer
            HRLI 1, 444400          ;change byte size to a word
            ILDB 3, 1               ;put the first full word in ac 3
NOMATCH:    AOJE COUNTER, DONE
            MOVE 2, SAVE+NUMSTR(COUNTER)   ;get next data ptr
            HRLI 2, 444400          ;change byte size to a word
            ILDB 4, 2               ;put word in ac 4
            CAME 3,4                ;same words
              JRST NOMATCH          ;no, then get next ptr
            MOVE 5,@FIND            ;yes, get number of chars
            ROT 5, BYTE             ;rotate left bits into right
            ANDI 5, BYTEMK          ;set all other bits to zero
            SUBI 5, CPWORD-1        ;looked at first CPWORD-1 chars
MORE:       CAIG 5, 4               ;at sentinel?
              JRST FOUND            ;yes
            ILDB 3,1                ;no, then get next word of FIND
            ILDB 4,2                ;next word of string
            CAME 3,4                ;same?
              JRST BEGINF           ;no, start over
            SUBI 5,CPWORD           ;sub CPWORD chars
            JRST MORE               ;get next word
FOUND:      MOVE 1, [POINT 7, [ASCIZ/ found /]]
            PSOUT%
            MOVE 1, SAVE+NUMSTR(COUNTER)  ;get data ptr
            IBP 1                   ;don't show num of chars
            PSOUT%
            HALTF%
DONE:       MOVE 1, [POINT 7, [ASCIZ/ did not find /]]
            PSOUT%
            HALTF%
            LIT
            END START
```

A simple JRST OVER after a HALTF% will not make this program restartable. Instead, we need to clear all previous FIND characters. If we were not to clear a first string of *abcde*, for example, the program would look for a second string, say 12 by searching mistakenly for 12*cde*!

In the program above, the code to load a string in the data bank is very similar to the code to load FIND. A high-level language would use a subroutine for this purpose. We will learn more about subroutines in the next chapter.

7.7 Summary

Historically, computers were of two types: scientific number crunchers, and business, data-processing machines (string manipulators). However, in recent years (post 1964) we have witnessed the arrival of the general-purpose computer, one designed to function on a program of commands with a view to solving a host of problems related to data-processing. With a general-purpose computer it is necessary to have methods whereby the user can set the logical size of information even though the physical unit is fixed by the machine word. This capability allows for infinitely better utilization of memory.

Armed with this byte capability and the better I/O operations introduced in this chapter, we are now ready to work with data structures.

7.8 Exercises

1. Write a program that will print the numerical positions of the letter A in a string. Use a 7-bit byte pointer to gain access to each character. You can write the string into the program via ASCIZ.
2. Write a program that simulates a RDTTY% by looping over a PBIN%. Since the monitor traps control characters, use 1 to delete a character, 2 as a control U, and a carriage return to terminate the input loop. After accepting the message, print it. (ADJBP is helpful here.) For example,

```
AB1CD1E
ACE
```

3. Write a program that simulates a MID$ function. The program should accept a string and two numbers and then print the partial string.
4. Set up an RDTTY%/NIN% to accept two 5-bit binary numbers. IOR the numbers; then NOUT% the result in a field of width 5, with leading 0's.
5. Place two strings (via ASCIZ) at two different locations in a program. Have the program move the strings to another area, packing the second string after the first string. Use a single pointer to display the concatenation.

CHAPTER

8

Stacks and Subroutines

A *stack* is a block of memory set aside for data. As such, it is a data structure and a way to reference data. The number of memory locations reserved is the *depth* of the stack. (Depth is actually a dynamic quantity signifying the number of elements presently on the stack. But because of the way the DECSYSTEM-20 operates, we will define depth as the maximum number of elements allowed on the stack.) Placing a value on the stack is *pushing*. Copying a value from the stack is *popping*. The last element pushed onto a stack is on the *top* of the stack. For a normal stack, the present top element is the only element that can be popped. A stack is a last in, first out (LIFO) data structure. It has finite depth: An OVERFLOW error occurs if a new element is pushed onto the stack when it is full. An UNDERFLOW error occurs with the application of a pop to an empty stack.

8.1 Stacks on the DECSYSTEM-20

Any accumulator can keep track of a stack; it then becomes a *stack accumulator*. The right half of the stack accumulator contains the address of the top. Pushing an element onto the stack causes the right half of the

stack accumulator to increment and place the element at the new top. Popping an element causes the right half of the stack accumulator to decrement. The left half of the stack accumulator keeps track of the number of elements associated with the stack; the computer monitors this value for either an underflow or an overflow. More specifically, any time the sign bit (bit 0) changes, a trap is set. The computer cannot monitor both types of errors (underflow and overflow) at the same time. To monitor for an underflow error, program a 0 in the left half of the stack accumulator. With each push, both halves of the stack accumulator increment. Each pop decrements both halves of the stack accumulator, signifying that one fewer element is in the stack and assigning the new top of the stack. Since the left half starts at 0, indicating that no elements are on the stack, an underflow error occurs any time a pop decrements a 0. The system will trap this error and stop. Below is an example of a stack setup for underflow.

```
          STK=17            ;ac 17 will be used
                            ;as a stack
          DEPTH=50
STACK:    BLOCK DEPTH       ;save depth many words
          ...
          MOVE STK, [0,,STACK-1]
```

The right half of the stack is set at STACK–1 because a push always increments and then deposits. The first push increments STACK–1, yielding STACK, the location of the beginning of the stack.

The usual way to monitor for an overflow error is to place the negative value of the stack depth in the left half of the stack accumulator. Since each push increments both halves, overflow occurs when the left half crosses from negative to 0, and the system will trap this error and stop. A minor difficulty arises, however, in that the last element allowed by the depth will cause a trap! Thus, if you want to push five elements, you must set the depth to 6 so that the trap will not activate when you place the fifth element on the stack. Below is an example of a stack setup for overflow.

```
          STK=17
          DEPTH=50
STACK:    BLOCK DEPTH
          ...
          MOVE STK, [-DEPTH,, STACK-1]
```

The assembly directive IOWD sets up a stack for overflow monitoring.

```
MOVE STK, [IOWD DEPTH,STACK]
```

After the above instruction, STK would contain −DEPTH,,STACK − 1. We could push 47 (octal) elements onto this stack; we could place a fiftieth element on the stack, but the system would trap.

Let us now repeat our simulation of NOUT% via PBOUT%, this time using a stack.

```
quotient gets number
while quotient not zero
  begin
    divide quotient by divisor
    push remainder
    quotient gets quotient
  end
count gets number of remainders
    (from the left half of stack ac)
for i:=1 to count
  begin
    pop a digit
    show digit
  end
```

The assembly version follows.

```
        TITLE EXAMPLE 22        ;NOUT% via PBOUT%
                                ;using a stack
                                ;note radix not used
                                ;this program could be
                                ;improved by using a RDTTY%
                                ;to accept an output
                                 base and any number
                                ;also, the extra ASCII offset
                                ;for characters above 9
                                ;could be added

        SEARCH MONSYM
        QUO=5
        REM=6
        COUNT=7
        STK=17
        DEPTH=20
STACK:  BLOCK DEPTH
BASE:   ^D8                     ;out base is octal
NUM:    ^D68                    ;number to be converted
```

```
START:    MOVEI STK, STACK      ;or MOVE STK,[0,,STACK]
          MOVE QUO,NUM
WHILE:    JUMPE QUO, GETCT
          IDIV QUO, BASE
          PUSH STK, REM
          JRST WHILE
GETCT:    HLRM STK, COUNT       ;get number of pushes
          JUMPE COUNT, DONE     ;check for a zero
FOR:      POP STK, 1            ;put value in ac 1
          ADDI 1, "0"           ;ASCII offset
          PBOUT%
          SOJG COUNT, FOR
DONE:     HALTF%
          END START
```

Placing elements on a stack via a push does not mean we can only remove them via a pop. For instance, if we loaded the stack using PUSH STK, ELEMENT, we could use indexing to retrieve data via the instruction MOVE AC, STACK(INDEX).

Stacks are also helpful in (pre)sorting. The program below sorts data into even and odd numbers. It uses two stacks, one for the even numbers and one for the odd numbers. We need not test to see if a number is even or odd; instead, we can use the least significant bit (bit 35) to designate the stack accumulator by copying bit 35 into the PUSH instruction's ac, bit 12. This program appears to be rewriting itself. It could be dangerous! An eXeCuTe opcode, XCT, can help avoid changing the code in a procedure. This opcode allows the program to jump out of the flow of instructions, execute the one instruction at the effective address of the XCT, and then return to the instruction after the XCT. (See the discussion of reentrant code in the next section.)

```
          TITLE EXAMPLE 23      ;stack sort
          SEARCH MONSYM
          DEPTH=10              ;number of data pieces
DATA:     0                     ;the data to be sorted into
          1                     ; even and odd
          2
          3
          4
          5
          6
          7
          COUNT=5               ;ac 5 is used as a counter
STACK0:   BLOCK DEPTH           ; stack for even numbers
STACK1:   BLOCK DEPTH           ; stack for odd numbers
EVEN:     Z                     ; number of even numbers
PTR:      POINT 1, HERE,12      ; choose a stack
HERE:     PUSH ,3               ; note ac determined by PTR
```

```
START:      MOVE 0, [0,,STACK0-1]  ;set up even stack0
            MOVE 1, [0,,STACK1-1]  ;set up odd stack1
            SETZ COUNT,
LOOP:       MOVE 3, DATA(COUNT)    ; place a number into ac 3
            DPB 3, PTR             ;place bit 35 of ac 3
                                   ;into bit 12 of HERE
            XCT HERE               ;do the instruction at HERE
                                   ;using an XCT makes this
                                   ;reentrant, see next section
            ADDI COUNT, 1
            CAIE COUNT, DEPTH
              JRST LOOP
                                   ;show the even numbers
            HRROI 1, [ASCIZ/
              the even numbers are
/]
            PSOUT%
            HLRM 0, EVEN           ;left half of stack pointer
                                   ;contains number of pushes
            SETZ COUNT,
            MOVEI 3, 12            ;set up NOUT%
SHOW:       MOVEI 1, .PRIOU
            MOVE 2, STACK0(COUNT)
            NOUT%
              HALTF%
            HRROI 1, [12B6+15B13]  ;CR, LF
            PSOUT%
            ADDI COUNT, 1
            CAME COUNT, EVEN
              JRST SHOW
            HALTF%
            END START
```

To sort at a finer level than even and odd, you can use a 3-bit byte and eight stacks. This method will return the partially sorted data to the data area after each presort, with the next 3 bits of each data word available for the next presorting. This process will eventually sort all the data.

A few additional opcodes are designed to work with the half-word structure of the stack accumulator. AOBJ*x* means "Add One to Both halves and Jump on *x*," where *x* can be P for positive or N for negative (bit 0). Another opcode, ADJSP (ADJust the Stack Pointer), allows you to change the position of the top of the stack. Like ADJBP, this instruction can move the stack pointer using the value in the operand.

8.2 Subroutines

Subroutines, also known as procedures, are a very important concept in programming. They allow you to write blocks of code once for multiple use. More importantly, they allow you to use a top-down structure in

tackling problems. You can think of a program in broad concepts (procedures), which you can refine (more procedures). BASIC uses "GOSUB label" and RETURN to handle subroutines. FORTRAN uses "CALL name" and RETURN. These instructions call for the system to save a value (the next line) for use upon RETURN. The programmer does not become involved in this process. In assembly, however, the user must make room for the value. The value is the program counter, PC word, or simply PC. It contains the address of the next instruction to be executed. (The system takes care of incrementing the PC after each FETCH.)

The simplest method for calling a subroutine is a Jump to SubRoutine, or "JSR label." This instruction saves PC in the location "label" and assigns PC the value "label + 1."

```
(PC)          JSR LABEL              ;call,PC get label+1
(PC+1)
              ...
label:         Z                     ;places PC+1 here
label+1        execute this line
              ...
              JRST @label            ;return
```

The assembler generates the value of (PC) and (PC + 1). (See the discussion of MACRO in Chapter 5.) The return is an indirect jump using "label." The fact that you write the value of the PC into the subroutine places some restrictions on its use. Namely, by changing part of the subroutine (changing the Z to the PC value), the subroutine is "nonreentrant." A *reentrant* subroutine does not change any opcode values within the routine. Thus, reentrant subroutines are important when more than one user is calling the routine. By not allowing opcodes to change, users can share routines without interfering with each other. Reentrant subroutines also have meaning for a single user when interrupts are possible, as you will see in Chapter 11. A second restriction is that storing the PC (the return address) in a fixed location means that the subroutine is "nonrecursive." A *recursive* subroutine can call itself. (Neither BASIC nor FORTRAN have this capability though some other languages, like PASCAL and LISP, do.) Example 25 on page 00 is a recursive routine.

Parameter passing is another problem with the simple call JSR. JSR is more similar to GOSUB in BASIC than to "CALL name(parameter list)" in FORTRAN. Thus, values to be passed must be in accumulators or in explicitly labeled locations. Below is one way to add two numbers via JSR, for example.

```
        NUM1:   value1
        NUM2:   value2
        ...
        JSR ADDER
        ...
ADDER:  Z
        MOVE AC, NUM1
        ADD AC, NUM2
        JRST @ADDER
        ...
```

Though an alternative would be to use one label with NUM2 referenced by NUM1 + 1, you would then need at least one label (or an ac with indexing, among other options). If you needed to add many pairs of numbers, referencing the pairs would become a mess. In this case the JSR option is not helpful.

Another method of calling a subroutine is Jump and Save the Program counter, JSP. This opcode has the following form.

```
JSP AC, LABEL
```

This opcode saves the program counter in the ac and executes the subroutine located at LABEL. The original contents of the ac are lost. Because the subroutine doesn't have to reserve a space for the PC, it is reentrant. It can also be recursive, but with restrictions. If the subroutine is to call itself, it must use a different ac for every call, which can lead to confusing RETURNs. Passing parameters is easier with a JSP than with a JSR. A program can place the parameters after the JSP and fetch by indexing off the JSP's ac.

```
        JSP AC, LABEL
        value1
        value2
        ...
LABEL:  MOVE AC1, 0(AC)     ;copy value1 into AC1
        ADD  AC1, 1(AC)     ;add value2 to AC1
        ...
```

Remember that after the system fetches instruction JSP, it increments the program counter and places the value in the ac. Therefore, after the computer fetches JSP, the saved PC references value1. If no parameters are passed, then JRST @AC would be a return. But, since an ac stores the PC, a faster and preferred method is JRST (AC). If *N* parameters are passed, the return is JRST N(AC).

The most commonly used subroutine call is a PUSH Jump, or PUSHJ. This call uses a stack to save the PC word and then jumps to the subroutine.

```
PUSHJ STK, LABEL
```

STK is set up as a normal stack accumulator. A POP Jump, or POPJ STK, will accomplish a return. Because of the LIFO structure of a stack, this method of calling allows for nesting subroutines as well as recursion. It also allows passing parameters but is not as label-free as JSP. Effective use of PUSH and POP can make parameter passing interesting. (To see how FORTRAN handles subroutines and parameters, see Appendix D.)
Table 8-1 summarizes the subroutines we have been discussing. Below is a (long) program that illustrates these calls. The program is an "improved" version of example 8, the "calculator" program. This program allows for parentheses and an error routine to check for correct use of parentheses. The program will accept a string like 3+(5*2), yielding 13. The same string without parentheses would yield the answer of 16 (3 plus 5 is 8; 8 times 2 is 16). The first part of the program below is the initialization.

```
        TITLE EXAMPLE 24            ;nested parentheses
        SEARCH MONSYM
        AMT=20                      ;5 times number of chars
                                    ;in an expression to be
                                    ;evaluated
PTR:    POINT 7, STRING             ;store the expression
STRING: BLOCK AMT                   ;here
        AC=6                        ;used in JSP (GETEM)
        NUM=7                       ;used as a counter
        ER=10                       ;used in JSP (ERROR)
        STK=17                      ;stack ac
        DEPTH=40                    ;depth of stack
STACK:  BLOCK DEPTH
        OP=11                       ;ac used to hold operators
TEMP1:  Z                           ;scratch pad area
TEMP2:  Z
```

TABLE 8-1. Subroutine Calls

	Method 1	*Method 2*	*Method 3*
Call	JSR LABEL	JSP AC, LABEL	PUSHJ STK, LABEL
Return	JRST @LABEL	JRST (AC)	POPJ STK,
Reentrant?	No	Can be	Can be
Recursive?	No	Limited	Can be
Parameter Passing	Nothing special	Index off ac	Use stack

And below is the program.

```
START:  MOVE STK, [0,,STACK] ;set up stack
        JSR GETEM            ;get the string expression
        JSP AC, CHECK        ;check parentheses
        PUSHJ STK, EVAL      ;evaluate expression
        PUSHJ STK, SHOWEM    ;show answer
        HALTF%
          JRST START         ;restartable
```

The subroutine to read the expression GETEM uses RDTTY%.

```
GETEM:      Z
            HRROI 1, [ASCIZ/type in an expression /]
            PSOUT%
            MOVE 1, PTR
            MOVEI 2, AMT*5
            SETZ 3,
            RDTTY%
             Z
            JRST @GETEM
```

The next procedure, CHECK, reads the string and checks for two possible parentheses errors. NUM, the number of left parentheses minus the number of right parentheses, helps monitor these errors. When the system has read the complete string, NUM should be equal to 0—that is, the number of open parentheses equals the number of closed parentheses. Also, since an open parenthesis must come before a closed parenthesis, NUM should never be negative. A negative indicates an error in the string. Below is the CHECK procedure.

```
procedure check
num:=0
error:=false
get char
while char <> carriage return do
  begin
    if char='(' then increment num
    if char=')' then begin
                       decrement num
                       if num < 0 then error
                       end
    get char
  end
if num<> 0 then error
return
```

The assembly routine follows.

```
CHECK:    SETZ NUM,              ;net amt of ( and )
          MOVE 1, PTR            ;get the string
WHILE:    ILDB 2,1               ;get a char
          CAIN 2,15              ;CR is end of string
            JRST WEND
          CAIE 2, "("            ;is char a (
            JRST RIGHT           ;no
          AOJA NUM, WHILE        ;yes, increase ( count
RIGHT:    CAIE 2, ")"            ;is char a )
            JRST WHILE
          SOJGE NUM, WHILE       ;count cannot be negative
ERR:      JSP ER, ERROR
          1                      ;value of error
          SETZ NUM,              ;so a continue is possible
          JRST WHILE
WEND:     CAIN NUM, 0            ;check if same num of ( and )
            JRST (AC)            ;return
          JSP ER, ERROR
            2                    ;value of error
          JRST (AC)              ;return
```

If an error occurs, JSP calls the ERROR procedure, which consists essentially of error messages. A jump table within the error procedure determines the message to be printed. The value (parameter) directly after the JSP call helps locate the correct JRST in the table; it passes the value to ac 3, which becomes an index into the jump table.

```
ERROR:    MOVE 3, (ER)           ;get parameter
          JRST ERROR+1(3)
          JRST ERROR1            ;jump table
          JRST ERROR2
          JRST ERROR3
          HRROI 1, [ASCIZ/ something is wrong in the
                                 jump table/]
          HALTF%
ERROR1:   HRROI 1, [ASCIZ/ a ) came before a ( /]
            JRST ERRET
ERROR2:   HRROI 1, [ASCIZ/ not the same number of ( and
          )/]
          JRST ERRET
ERROR3:   HRROI 1, [ASCIZ/ invalid operation /] ;see
          below
          JRST ERRET
ERRET:    PSOUT%                 ;show message
          HALTF%                 ;stop
            JRST 1(ER)           ;allows a continue
```

The error procedure uses a continue to enable the user to discover other errors once a first error is revealed. Since this procedure is not very sophisticated, the value of the continue is questionable. However, it helps give you an idea of how a computer program scans a line, a process known as *syntax checking*.

If the procedure reveals no errors, then the system can safely evaluate the expression using the EVAL procedure. EVAL is very similar to the procedure illustrated in example 8, but it has the added ability to handle parentheses. When it reads an open parenthesis, the program stores the present value of the calculation and the current operation. When it reads a closed parenthesis, it retrieves the previous stored value and operation. The system evaluates the previous value and the current value based on the retrieved operation. An open parenthesis can only occur in expectation of a number; a closed parenthesis can only occur in expectation of an operator. Therefore, before reading a number, the system must check a single character to see if it is an open parenthesis. If it is, it places the previous values on the stack and restarts the program. If not, the system must back up the byte pointer one character and perform a NIN%. NIN% will terminate on the first nondigit. Unfortunately, because the system will still read the nondigit character, the pointer must back up one byte after the NIN%. When reading an operation, the system must check a character to see if it is a closed parenthesis. If not, it will read it as an operation and proceed to do the operation. If it is a closed parenthesis, the stack must be popped twice. The popped operator will operate on the present value of the expression and the popped value.

```
        procedure eval
again:  op:= '+'
        temp:=0
        while op <> '=' do
          get char
          if char = '(' then begin
                                push temp
                                push op
                                jump to again
                            end
          else getnum
          do op
          get char
          if char = ')' then begin
                                pop op
                                pop temp
                                do op
                            end
          else getop
```

The assembly version follows.

```
EVAL:       MOVE 1, PTR                 ;get expression
AGAIN:      MOVEI OP, "+"               ;initialize op
            SETZM TEMP1                 ;and TEMP1
;start while loop
WLOOP:      CAIN OP, "="
             POPJ STK,                  ;return
;get a char see if it is (
            MOVEM 1, TEMP2              ;save the pointer
            ILDB 4,1                    ;get a character
            CAIE 4,"("                  ; is it (
              JRST [MOVE 1, TEMP2       ;no, then restore pointer
                    JRST GETNUM]        ;and get the number
            PUSH STK, TEMP1             ;    yes, then save TEMP1
            PUSH STK, OP                ;         and the op
            JRST AGAIN                  ;         start over
GETNUM:     MOVEI 3, 12                 ;set up NIN%
            NIN%
             Z
            MOVNI 5, 1                  ;when reading digits
            ADJBP 5, 1                  ;the NIN% will read until it
            MOVEM 5, 1                  ;finds a non digit, then
                                        ;back up ptr one byte to
                                        ;get terminating non digit
FINDOP:     CAIN OP, "+"
             JRST PLUS
            CAIN OP, "-"
             JRST MINUS
            CAIN OP, "*"
             JRST MULT
            JSP ER, ERROR
             3                          ;bad opcode
            HALTF%                      ;can not continue
            JRST START                  ;restartable
PLUS:       ADDM 2, TEMP1               ;do op
             JRST GETOP
MINUS:      SUB 2, TEMP1                ;this is ac2 minus TEMP1
            MOVNM 2, TEMP1              ;now it is TEMP1 minus ac2
             JRST GETOP
MULT:       IMULM 2, TEMP1
                                        ;get the next op
GETOP:      ILDB OP,1
            CAIE OP, ")"
             JRST WLOOP
            POP STK, OP                 ;yes, then get old op
            POP STK, 2                  ;     and previous TEMP1
            JRST FINDOP                 ;     do operation
```

The last procedure, SHOWEM, is a simple NOUT%.

```
SHOWEM:     MOVEI 1, .PRIOU
            MOVE 2, TEMP1
            MOVEI 3, 12
            NOUT%
             Z
            POPJ STK,
            END START
```

The next example demonstrates recursion. The problem is to calculate N factorial ($N!$). Recall that $N! = N*(N-1)* \cdots *1$. For example, $3! = 6$, $4! = 24$. The definitions of 0! and 1! are both 1. You can use *iteration* to perform a factorial.

```
FACTOR=1
FOR I=N DOWNTO 1 DO
        FACTOR=I*FACTOR
```

To do this problem via recursion, define the factorial symbol (!) in terms of itself. The values of 0! and 1! are the only values defined without the factorial symbol. Using recursion, FACTOR becomes a function.

```
FACTOR(N)
 IF N<=1 THEN FACTOR=1
         ELSE FACTOR=N*FACTOR(N-1)
```

Note that the function FACTOR calls itself, which is its recursive element.

```
            TITLE EXAMPLE 25        ;factorial via recursion
                                    ;this program could be improved by
                                    ;using RDTTY%
                                    ;check for negatives
                                    ;too large a number
                                    ;and error routines
                                    ;as well as a fancier printout,
                                    ;like 3!=6
            SEARCH MONSYM
            STK=17
            DEPTH=20                ;half the stack is used for
STACK:      BLOCK DEPTH             ;returns, the other for numbers
                                    ;then with a depth of 20 the
                                    ;largest factorial is 20/2=10 or 8
TEMP1:      Z                       ;scratch area
```

```
START:  MOVE STK, [IOWD DEPTH, STACK]
        PUSHJ STK, GETNUM    ;get the number
        PUSHJ STK, FACTOR    ;calculate its factorial
        PUSHJ STK, SHOWAN    ;show the answer
        HALTF%
         JRST START

GETNUM: HRROI 1, [ASCIZ/ type in a positive number /]
        PSOUT%
        MOVEI 1, .PRIIN
        MOVEI 3, 12
        NIN%
         Z
        POPJ STK,
                             ;call this routine with
                             ;a number in ac 2
                             ;leave this routine with
                             ;an answer in ac 2
FACTOR: CAIG 2, 1            ;if N=1 then return
         POPJ STK,           ;return
        PUSH STK, 2          ;save ac 2 value
        SUBI 2, 1            ;construct N-1
        PUSHJ STK, FACTOR    ;call with N-1
        POP STK, TEMP1       ;get last ac 2 value
        IMUL 2, TEMP1        ;form N*FACTOR(N-1)
        POPJ STK,            ;return

                             ;call this with a number
                             ;to be printed in ac 2
SHOWAN: HRROI 1, [ASCIZ/ the factorial value is /]
        PSOUT%
        MOVEI 1, .PRIOU
        NOUT%
         Z
        POPJ STK,
        END START
```

8.3 External Subroutines

In this section, we will learn how to call subroutines that are *external* to the calling program. External procedures can be in a program or in a *library*, which is a collection of procedures (not a program). A library does not have a label following END. In order to execute a program with an external procedure, the computer (linker) must have the file containing the program and the file containing the external procedure. For example, if the file containing the program were PROG.MAC and the file containing the external procedure were EXTN.MAC, you would type the following at TOPS-20.

```
EX EXTN.MAC,PROG.MAC
```

The order of the files following EX is important only when a program contains more than one labeled END statement. If EXTN.MAC above is also a program, which program will the system execute? It will use the last labeled END statement to decide. If the external procedure is in a library, the order of the files following EX is not important. Once the system has collected the files, we have a potential problem with the assembler. For example, suppose we give the label EXSUB to the subroutine in EXTN.MAC. In the calling program PROG.MAC, JSP AC, EXSUB (or JSR or PUSHJ) would reference this routine. But PROG.MAC does not define EXSUB in its symbol table. We can get around this problem using the assembly directive EXTERN at the beginning of PROG.MAC.

```
EXTERN EXSUB
```

If more than one procedure is external to the program, you will need two or more EXTERNs, or EXTERN EXSUB1, EXSUB2, and so on. A symbol defined in a program is a *local symbol*. When the assembler is trying to identify a symbol, it checks the local symbol table first. If it does not find the symbol locally, it examines the external tables. Therefore, if a local symbol and an external symbol are symbolically the same—for example, they are both called SECOND—the assembler will use the local definition. (You may have considered using the SEARCH directive to bring forth the value of the external symbol. However, SEARCH deals with MACROs, not subroutines. See the next chapter.)

The file containing the external procedure raises another problem. In order to gain access from outside a file to a procedure in the file, you use the assembly directive ENTRY within the file containing the procedure. You must announce all procedures that can be accessed by other programs as in the example below.

```
ENTRY EXSUB
```

The value following ENTRY must be the starting label of the procedure. If more than one procedure is available, you must use more than one ENTRY, or ENTRY EXSUB1, EXSUB2, and so on. Below are three files. The first two are programs (EXAM26.MAC and EXAM27.MAC); the third is a library (EXAM28.MAC).

```
                TITLE EXAMPLE 26          ;simple program
                                          ;using external subs
                SEARCH MONSYM
                EXTERN SECOND, THIRD      ;THIRD is not defined in
                                          ;this program but
                                          ;SECOND is, see below
                ENTRY FIRST               ;FIRST is available to
                                          ;other programs
                STK=17
                DEPTH=10
STACK:          BLOCK DEPTH
START:          MOVEI 17, STACK
                HRROI 1, [ASCIZ/ in first program
  /]
                PSOUT%
                PUSHJ 17, SECOND
                PUSHJ 17, FIRST
                JSR THIRD
                HALTF%

FIRST:          HRROI 1, [ASCIZ/ sub in first program
  /]
                PSOUT%
                POPJ 17,
SECOND:         HRROI 1, [ASCIZ/ second is local
  /]
                PSOUT%
                POPJ 17,
                END START


                TITLE EXAMPLE 27          ;simple program
                                          ;using externals subs
                SEARCH MONSYM
                ENTRY SECOND
                EXTERN FIRST, THIRD
                STK=17
                DEPTH=20
STACK:          BLOCK DEPTH
START:          MOVE STK, [0,,STACK]
                HRROI 1, [ASCIZ/ in second program
  /]
                PSOUT%
                PUSHJ 17, FIRST
                PUSHJ 17, SECOND
                JSR THIRD
                HALTF%
```

```
SECOND:   HRROI 1, [ASCIZ/ sub in second program
 /]
          PSOUT%
          POPJ 17,
          END START


          TITLE EXAMPLE 28              ;a library
          SEARCH MONSYM
          ENTRY THIRD
THIRD:    Z
           HRROI 1, [ASCIZ/ sub in library
 /]
          PSOUT%
          JRST @THIRD
          END                           ;note no label
                                        ;following end
```

If we execute the files via

```
EX EXAM26.MAC,EXAM27.MAC,EXAM28.MAC
```

EXAM27 is the program (as distinguished from external subroutines), and the printout would be as follows.

```
in second program
sub in first program
sub in second program
sub in library
```

If we execute the files via

```
EX EXAM27.MAC,EXAM26.MAC,EXAM28.MAC
```

EXAM26 is the program, and the printout would be as follows.

```
in first program
second is local
sub in first program
sub in library
```

Note that the call and accumulator that do the calling also do the returning. In the above programs, PUSHJs using ac 17 referenced FIRST and SECOND. Thus, the returns must be POPJs using ac 17. In the library,

THIRD is set up for a JSR (because of the leading Z); therefore, JSR, not JSP or PUSHJ, should call it as well. Using the symbol STK in both programs will not create confusion (nor is it necessary to use it), because symbols are local to the program that defines them. However, the symbol must refer to the same ac in both programs. Programs can share symbols much as they share subroutines. A shared symbol is called a *global symbol*. To make a symbol global, use the directive INTERN in the file that defines the symbol. INTERN is similar to ENTRY. Use EXTERN in the file that wishes to use the symbol. The pair ENTRY/EXTERN initiates shared subroutines. The pair INTERN/EXTERN governs shared symbols and thus the values within the symbol locations. If the values to be shared are in accumulators, you need not use the directives. If too many values are to be shared or if the accumulators are needed for some other reason, you must use globals to share information.

One difficulty with the above method of using external subroutines is that you must type in the name of all files needed by the program: EX EXAM26, EXAM27, and so on. Why not place the names of extra files within the program? The assembly directive .REQUIRE (the leading period is necessary) allows you to do this. Of course, some conditions are necessary. All subroutines needed by a program must be in libraries (no label after the END statement), and all libraries must be compiled. When these conditions are met, type the names (no extension) of the libraries after the directive. Thus, to execute the program, you need only announce the program name; the linker will use the files following the directive .REQUIRE. For example, if you replaced the line EXTERN SECOND, THIRD in example 26 with EXTERN THIRD and then added the line .REQUIRE EXAM28, you would only need to type EX EXAM26.MAC to execute the program.

8.4 Coroutines

Main-line programs and subroutines have a master-slave relationship. A subroutine waits to be called by the main-line program. When called, the subroutine executes a block of code from beginning to end. Thus, subroutines exist in a dictatorship ruled by a main-line program. *Coroutines*, on the other hand, exist in a democracy. Much cooperation (interwoven calls) exists between coroutines, with no clear hierarchy. One way to handle multiple users on a single computer (processor) is to treat each user (process) as a coroutine. No process is a slave to another process. An alternative to running coroutines is to run routines sequentially; a new process cannot start until another process has finished. If the processes are efficiently

written, sequential processing is fine. However, a process with a great deal of I/O can lead to wasted time waiting for I/O. Coroutines enable you to execute one process while waiting for I/O on another.

Let us look at an example that uses two loops, or coroutines. Loop1 types a message requesting a number. Execution then transfers to loop2 where, for want of something simple to do, a counter counts down. When the counter gets to 0, control passes back to loop1, which then accepts a number. Once the system reads a typed number, it transfers control back to loop2, prints "thanks," and transfers control back to loop1. Figure 8-1 shows the time flow of the control schedule. Remember that coroutines *interleave* time, not memory. Thus, loop1 is in a contiguous block of memory that does not overlap with loop2's contiguous block of memory.

The macro program below illustrates code for this example. The counter in loop2 uses the number supplied by loop1. In a "real" problem, loop2 would be doing something more meaningful than counting down. The countdown number can be very large before you notice terminal response hesitation in jumps from one loop to another. (Because the DECSYSTEM-20 is a time-sharing system, this number is only a relative quantity.)

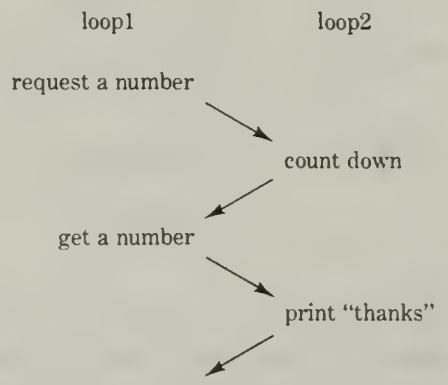


Figure 8-1. A Sample Coroutine

```
                TITLE EXAMPLE 29     ;coroutines
                SEARCH MONSYM
NUM:            Z
START:          HRROI 1, [ASCIZ/ STARTING
 /]
                PSOUT%

LOOP1:          HRROI 1, [ASCIZ/ TYPE IN A NUMBER
 /]
                PSOUT%
                JSR FROM1TO2
                MOVEI 1, .PRIIN      ;set up NIN
                MOVEI 3, 12
                NIN%
                 Z
                MOVEM 2, NUM
                JSR FROM1TO2
                JRST LOOP1
LOOP2:          MOVE 10, NUM
                SOJG 10, .       ;'block of NUM statements'
                JSR FROM2TO1
                HRROI 1, [ASCIZ/ THANKS
 /]
                PSOUT%
                JSR FROM2TO1
                JRST LOOP2

FROM1TO2:       LOOP1
                JRST @FROM2TO1

FROM2TO1:       LOOP2
                JRST @FROM1TO2

                END START
```

The period in SOJG 10,. indicates the present value of the location counter. (The SOJG instruction is the jump location.) JSR transfers from one coroutine to another. Recall that JSR captures the content of the program counter and stores it at the starting location of the called routine, which is the next instruction to be executed. This instruction is an indirect jump into the other coroutine. Initially, the "jump to" address is the first line of a coroutine loop. After each call, JSR updates this address with the latest PC value.

The coroutines discussed above are effectively independent of each other, for they do unrelated tasks. You can also use coroutines to help execute a single problem. For instance, let us consider one way to use coroutines to format text in a word processor. One coroutine could read in characters,

eliminating redundant blanks as well as blank lines, and place these characters into a buffer (a block of memory). Another coroutine could read from this buffer and write words to another buffer. Finally, a third coroutine could read the words and print them on lines of fixed width without breaking words. The routines cooperate in this way: Once the character buffer is full, the "read character" routine pauses, passing control to the "make word" routine. This routine constructs words from these characters and places them into the word buffer. If the word buffer fills, the "make word" routine passes control to the "print line" routine. If the word buffer does not fill, the "make word" routine passes control back to the "read character" routine. Figure 8-2 illustrates this process. Note that the JSR method of transferring control would not work in this example. The "word-making" coroutine can transfer to either the "read character" or "print line" coroutine, which would require two places for the "make word" program counter.

In the above examples, we have only needed to preserve the PC when transferring from one routine to another. However, regardless of the amount of (or lack of) cooperation, the routines are still independent processes so we must generally preserve more than the PC. Specifically, we must preserve any quantity of one coroutine that could be changed inadvertently by another coroutine. Typically we will need to preserve accumulators (and possibly some memory) as well as the PC. (The accumulators in example 29 did not contain state, or conditional, information, so we did not need to save their contents before transferring.) One way to preserve the necessary elements (and to solve the above JSR problem) is to create state tables. For each process, we must set aside a block of memory large enough to hold all quantities that might change. To simplify the problem, let us assume that we need to preserve only the PC and accumulator 17. Given a program of three coroutines, we must set aside a block of six words, for example, STATETAB:BLOCK 6. We give each routine a process identification number, such as 0, 2, and 4. We then add the process identification to STATETAB to find the state block for a coroutine. (These locations would already be loaded with the starting address of each coroutine.)

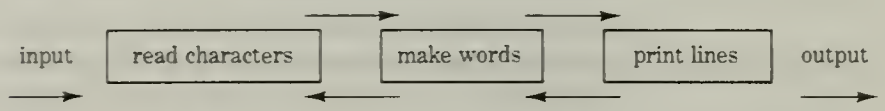


Figure 8-2. A Text-Formatting Coroutine

```
STATETAB:    COR1     ;start address routine one
              Z       ; cor1's ac 17 saved here
             COR2     ;start address routine two
              Z       ; cor2's ac 17 saved here
             COR3     ;start address routine three
              Z       ; cor3's ac 17 saved here
```

We can now refer to coroutines by their identification numbers. Thus, the following program would transfer from one routine (cor2 with an identification number of 2) to another (cor1 with an identification number of 0).

```
          MOVEI 1, 0                ;process id going to
          MOVEI 2,2                 ;process id presently
                                    ;in
          JSP 3, trans              ;place pc in 3 and
                                    ;goto trans
           return here to continue process 2
          ...
TRANS:    MOVEM 3, STATETAB(2)      ; put old pc in
                                    ;statetab
          MOVEM 17, STATETAB+1(2)   ; save ac 17
          MOVE 17, STATETAB+1(1)    ; get new 17 content
          JRST @STATETAB(1)         ;jump into process 1
```

If we needed to preserve more than one accumulator, STATETAB would expand and we would need more MOVE/MOVEMs. In the next chapter, we will examine an opcode (BLT) that helps transfer blocks of memory and/or accumulators.

8.5 Summary

One of the major contributions to the writing of software is the concept of top-down programming. With this approach a programmer first answers all "what" questions and then all "how" questions. Effectively "what" questions are the names (calls) of subroutines (procedures) and "how" questions are the actual subroutines. Programs can be thought of as made up of units or modules. Allowing these modules to be translated (assembled, compiled) separately (externally) allows for greater flexibility. If a module is improved, only that external module need be retranslated rather than the complete program.

Subroutines still do not offer enough flexibility. Regarding certain code optimizations, the translator does not interact with the subroutine. The next chapter will introduce MACROs. The MACRO can contain code that instructs the translator as to what is to be translated and how to translate it.

8.6 Exercises

1. Write a subroutine that will substitute a "new" character for an "old" character in a given string. Use JSP to call the subroutine, and place the three parameters—pointer to a string, new character, and old character—after the call location.
2. Write a subroutine that will specify the location of the left-most 1 (set bit) in a 36-bit DEC word. Call this routine, and display the position value.
3. Write a queue program. Call a PUT subroutine to place a number into the queue via a PUSH. Call a TAKE subroutine to remove an element from the queue. (Use another accumulator, and treat the stack block as an array.)

Example:

```
PUT or TAKE     P
IN VALUE        3
PUT or TAKE     P
IN VALUE        2
PUT or TAKE     T
OUT VALUE IS    3
PUT or TAKE     D
DONE
```

4. Set up an external subroutine library (LIB.MAC) containing a routine that will add two numbers. Use a stack to pass the numbers and their sum. Write a program (PROG.MAC) that sets up the stack, places the two numbers into the stack, and then calls the external routine. Upon returning, POP and display the answer. (*Note:* Since the data are behind the return address, the routine must first POP the stack and preserve this return value. Next POP the two data values. Perform the addition, PUSH the answer, PUSH the return address, and return with a POPJ.)
5. Write a routine that will accept three strings from the terminal. PUSH the pointers of each string onto a stack. When the last string has been typed in, POP each pointer and display the strings.
6. Sort a group of octal numbers using eight stacks as described in the text.
7. Write a recursive factorial routine that accepts numbers using RDTTY%/NIN% and checks for negative numbers (see example 26).

CHAPTER

9

Macros

We have discussed the equals-sign assembly directive that allows us to define a symbol. As you recall, the value of the symbol is generally a number, as in ac = 2. In this chapter, we will discuss user-defined symbols that are blocks of code rather than single numbers. We need *macros* for this purpose; like subroutines, macros are modules that are written once but can be used many times. Evoking macros is different from evoking subroutines, however. The assembler places the complete macro in the program at assembly time rather than jumping to the macro code whenever it needs a macro at run time. The length of a program depends on the number of times a macro is necessary. A macro demands more computer memory than does a subroutine, but it nonetheless saves time since the program need not save the PC and JUMP, among other elements. Macros also offer other advantages. For instance, they can use special assembly directives that are not available to subroutines; these directives can select the lines of code to encode.

9.1 Defining a Macro

Subroutines generally appear at the bottom of a program and are usually isolated from the rest of a program (typically by a HALTF%). This approach prevents the program from slipping into the subroutines. You must generally define a macro before evoking it, however, so you need to place it at the beginning of a program (similar to a procedure). Macros are identified by their names, and these names have restrictions much as labels do. The simplest approach is to use up to six alphanumeric characters, designating the first as a letter. (Macro names should not be the same as opcodes, as you will see.) You must use the following format in defining all macros.

```
DEFINE macname (darglist) <
            statements >
```

DEFINE is a reserved word. Macname is the name of the macro. The *darglist* (the *d*ummy *arg*ument *list*) is a list of the formal parameters the macro uses. This format is very similar to that of a FORTRAN subroutine or a PASCAL procedure. Parentheses delimit the darglist, and the angle brackets enclose the statements that are the body of the macro. Placement of the brackets is not important as long as they delimit the body of the macro; the opening bracket need not be on the same line as the darglist, and the closing bracket can appear by itself on the last line of the macro. Below is an explicit example of a macro.

```
DEFINE PLUS (A, B) <
        MOVE 1, A
        ADD 1, B >
```

Invoke PLUS as follows.

```
...
PLUS (3, HERE)
...
```

This macro places the sum of the contents of ac 3 and location HERE into ac 1. When the program invokes a macro, the assembler replaces the line PLUS (3, HERE) with all the lines in the macro.

```
...
MOVE 1, 3
ADD  1, HERE
...
```

Below is another macro example.

```
DEFINE PRINT (MESS)
<HRROI 1, [ASCIZ/ MESS /]
 PSOUT%
>
```

which we invoke as follows.

```
...
PRINT (HELLO THERE)
...
```

This macro sends the message HELLO THERE to the terminal. You can use a macro name within another macro as long as you have previously defined the nested macro. (It is also possible to define a macro within another macro, but we will not discuss this issue here.)

When the assembler scans a line and interprets symbols, it looks at macro definitions before opcode definitions. Thus, it is possible to (erroneously?) change the meaning of an opcode. For example, if we define the macro ADD as

```
DEFINE ADD (A,B)<
        IMUL A,B>
```

and use the symbol ADD as

```
...
ADD 2, 3
...
```

the assembler will replace ADD with the macro definition IMUL 2,3 rather than with the opcode ADD! To avoid possible confusion, avoid using opcode symbols to name macros. The following program uses some standard macros.

```
        TITLE EXAMPLE 30                    ; examples of macros
        SEARCH MONSYM
SAVE:   POINT 7, NUMBER
NUMBER: BLOCK 2

  DEFINE PRINT (MESS) <
        HRROI 1, [ASCIZ ? MESS ?]           ;? is used rather than /
        PSOUT% >                            ;so a / could be printed
```

```
            DEFINE INPUT (MESS, VALUE) <
            PRINT (MESS)                        ;nested macro
            MOVE 1, SAVE                        ;setting up RDTTY%
            MOVEI 2,10                          ;for 8 characters
            SETZ 3,                             ;no message on ^R
            RDTTY%
             ERJMP  [PRINT (RDTTY% i/o
                    error)
                    HALTF%]
            MOVE 1, SAVE                        ;set up NIN%
            MOVEI 3,12
            NIN%
             ERJMP  [PRINT (NIN% i/o error)
                    HALTF%]
            MOVEM 2, VALUE>                     ;save input in value

  DEFINE NUMOUT(VALUE) <
            MOVEI 1,.PRIOU                      ;set up NOUT%
            MOVEI 3,12
            MOVE 2, VALUE
            NOUT%
             ERJMP  [PRINT (NOUT% i/o error)
                    HALTF%]
            >

PROG:       INPUT(FIRST NUMBER , FIRST#)
            INPUT(SECOND NUMBER, SECOND#)
            PRINT (THE SUM IS )
            MOVE 4, FIRST
            ADD 4, SECOND
            NUMOUT(4)
            PRINT (
)                                               ;carriage return
            HALTF%
            END PROG
```

If we were to use ac 1 instead of ac 4 to do the additions in the program above, we would introduce an error. Because the macro NUMOUT uses ac 1 (as well as acs 2 and 3), NUMOUT(1) would not work. Since we use acs 1, 2, and 3 with monitor calls, we can avoid possible errors by using them only for this purpose. If you need to use all the acs in a program, you should copy (transfer) the contents of any ac used by the procedure to a memory location before invoking the procedure. When the procedure is complete, you must transfer the original contents back to the ac.

BLT (BLock Transferring) is an opcode that copies. It will transfer *N* machine words from a location, say FIRST, to another location, say HERE. Place the label FIRST in the left half of the ac used with BLT and the label

HERE in the right half. The operand is the last address in the transferred section, HERE + N − 1. You must be careful not to transfer the ac used in the BLT instruction when transferring the contents of acs. For example, to transfer the contents of the first five acs (0 to 4) to HERE, use the following code.

```
MOVE 5, [0,,HERE]
BLT 5, HERE+4
```

This example could have used any ac but 0, 1, 2, 3, 4 with the BLT. The following macros save and return acs.

```
DEFINE SAVE<
MOVEM 4, HERE+4        ;save BLT ac
MOVE 4, [0,,HERE]
BLT 4,HERE+3>          ;move other acs

DEFINE RETURN<
MOVE 4,[HERE,,0]       ;get acs except BLT's
BLT 4,3
MOVE4,HERE+4>          ;get last ac
```

Note that the user must supply a BLOCK labeled HERE. Also, note that MOVE freed ac 4 so that BLT could use it. If a macro and a program use the same acs, the first line of the macro should be SAVE and the last line RETURN.

```
DEFINE name...<
SAVE
...
RETURN>
```

Three assembly directives offer options in listing a program containing macros: XALL, SALL, and LALL. The default directive is XALL. When listing a program that contains a macro, the line calling the macro is followed by a circumflex (^), the macro code, and another circumflex. XALL lists only those lines that generate code. SALL lists only the title of the macro, with no code. LALL lists all information within the macro. To change from XALL to SALL or LALL, simply include one of these directives (SALL or LALL) in the beginning of a program.

9.2 Created Symbols and Default Values

Since the system expands macros at every place they are called, a problem arises if a macro contains a standard label: with expansion of the macro, the label references a different location! However, label locations must be unique (used only once). To handle this problem, the assembler can create symbols. A percent sign before a formal parameter in the dummy argument list will be the clue to the assembler to create a symbol. When a macro is called and a value is not passed to a created symbol, the assembler will assign a value to the symbol. The value it assigns will be different for each call.

```
        DEFINE LABEL(A, %B)<
        JRST %B
%B:     MOVEI 1, A>

        ...
        LABEL(3)
        LABEL(4)
        LABEL(5,HERE)
        ...
```

Expanding this example yields the following.

```
LABEL(3)^
                    JRST ..0001
..0001:              MOVEI 1, 3^
LABEL(4)^
                    JRST ..0002
..0002:              MOVEI 1, 4^
LABEL(5,HERE)^
                    JRST HERE
HERE:                MOVEI 1, 5^
                    ...
```

Created symbols begin with dots. As discussed earlier, all your labels should begin with a letter to avoid confusion.

Macro parameters can have default values other than null. For instance, expansion of a macro invoked with a simple LABEL would be as follows.

```
            ...
LABEL^
            JRST ..0003
..0003:     MOVEI 1, 0^
            ...
```

To choose a default value other than null, you must place the default value, enclosed in angle brackets, next to the formal parameter in the darglist.

```
        DEFINE LABEL(A<2>,%B)<
        JRST %B
%B:     MOVEI 1, A>
```

A macro referenced as

```
...
LABEL
...
```

would expand as follows.

```
                ...
LABEL^
                JRST ..0001
..0001          MOVEI 1, 2^
                ...
```

With more than one formal parameter, the interpretation of default becomes hazy. Consider the following example.

```
DEFINE LABEL2 (A<2>,B<3>,%C)<
        JRST %C
%C:     MOVE A,B>
```

Used by itself, LABEL2 makes it clear that all parameters are missing, so the program can use their default values.

```
LABEL2^
                JRST..0001
                ..0001 MOVE 2,3^
```

LABEL2(5) would indicate that a value is passed to the first parameter, the other two parameters are missing, and their default values are used. Below is the expansion.

```
LABEL2(5)^
                    JRST     ..0002
                    ..0002   MOVE 5, 3^
```

If we were to use the expression LABEL2(,4), the expansion would be as follows.

```
LABEL2( ,4)^
                JRST    ..0003
                ..0003  MOVE    ,4^
```

The first parameter does not assume its default value because the system does not treat it as missing. The last call is the same as LABEL2(0,4). Only parameters that are truly missing, rather than just skipped over, have default values. We place created symbols (for example, %C) last in the dummy argument list for this reason.

9.3 Parameter Passing and Special Pseudo-Ops

Parentheses are not really necessary when calling a macro. For example, you can invoke the macro PRINT either with PRINT(HELLO) or with PRINT HELLO.

```
DEFINE  PRINT (mess) <
        HRROI 1, [ASCIZ/ mess/]
        PSOUT%>
```

To pass parentheses to a macro, simply repeat the parentheses.

```
PRINT ((HELLO))
```

results in the message

```
(HELLO)
```

The use of parentheses in calling a macro is subject to a number of rules and exceptions to the rules. Thus, the simplest approach is to use parentheses only to pass parentheses (as above), spaces, or CRLF.

In the macros we have discussed so far, we have treated the parameters individually. For instance, we could use the following to accept a number.

```
DEFINE NUMIN(VAR,BASE<12>)<
        MOVEI 1, .PRIIN
        MOVEI 3, BASE
        NIN%
         HALTF%
        MOVEM 2, VAR>
```

And we would call this macro with NUMIN A#. To accept two numbers, we would have to use NUMIN A# and NUMIN B#.

In a high-level language, an INPUT or READ is not restricted by the number of variables to be accepted. BASIC uses INPUT A or INPUT A,B which would call the input function as many times as the variables following INPUT. A *pseudo-op* called IRP (Indefinite number of REpeats) allows a macro to perform a similar operation. It takes this form.

```
IRP parameter, <code>
```

You can use IRPs only within macros. You must also group together all parameters that will use "<code>" and pass them to the macro as a single parameter. IRP will then repeat the "<code>" for each parameter in the group. An example will help.

```
DEFINE  ADDEM (A) <
        MOVEI 1, 0
        IRP A, < ADD 1, A>>
```

This macro adds a group of variables. To call this macro with more than one parameter, you must group the parameters within angle brackets.

```
ADDEM <3,4,5>
```

The angle brackets indicate that just one parameter is being passed to ADDEM. IRP repeats the "<code>" for each variable separated by commas in that one parameter. The call to the macro, not IRP, removes the angle brackets. ADDEM expands as follows.

```
ADDEM<3,4,5>^
        MOVEI 1, 0
        ADD 1, 3
        ADD 1, 4
        ADD 1, 5^
```

Grouping parameters brings out an interesting aspect of macro calling. Consider the following macro.

```
DEFINE READ(A, BASE<12>)<
        MOVEI 1, .PRIIN
        MOVEI 3, BASE
        NIN%
         HALTF%
        MOVEM 2, A>
```

Using this macro, which has two parameters, and the previous PRINT macro, we can define a new macro.

```
DEFINE  READEM(mess, var)<
        PRINT mess
        IRP var, <READ var> >
```

We can invoke READEM with

```
READEM type in two numbers, <a#,b#>
```

This statement would print a message and then assign the input values to *a#* and *b#*, evaluating the numbers in decimal because of the default parameter BASE. How can we change BASE in the READ macro? In other words, how does IRP handle two (or more) parameters? One reasonable but incorrect guess would be to call READEM with

```
READEM type in two numbers, <<a#,5>,<b#,6>>
```

The attempt is to read *a#* in base 5 and *b#* in base 6. However, this method passes <*a#*,5>,<*b#*,6> to IRP, which in turn passes <*a#*,5> to READ. Now the angle brackets indicate that READ has only one parameter, and it will try to do a MOVEM, which does not make sense.

```
MOVEM 2, a#,5
```

Remember that IRP does not remove brackets; only calls to macros will remove brackets. So we must write a macro that will remove the brackets.

```
DEFINE  REMOVE(stuff)< stuff>
```

REMOVE will not change its argument, but it will remove brackets because a call to a macro removes one set of brackets (if they are present). With this fact in mind, we can rewrite the IRP line in READ as

```
IRP var, <read (REMOVE (var))>
```

This line assumes that the system will call REMOVE and then READ. Not true! Instead, READ thinks it has a parameter, namely "REMOVE(var)," which it will expand as

```
MOVEM 2, REMOVE(<a*,5>)
```

which expands as

```
MOVEM 2, a*,5
```

If a macro is the argument of another macro, the calling moves from left to right and not from the innermost nest outward. The correct intermediate macro, then, is

```
DEFINE  COMB (mac, stuff) <mac stuff>
```

COMB removes a COMma and a set of Brackets. The IRP line of READEM is replaced with

```
IRP var, < COMB ( READ, var)>
```

Now when IRP receives <a#,5>, the steps are

```
COMB( READ, <a*,5>)
```

yielding

```
READ a*,5
```

With COMB, calls to READEM would be

```
READEM type in two numbers, <a*,b*>
```

or

```
READEM type in two numbers, <<a*,5>,<b*,6>>
```

A better use of the above macro concept would be to treat the numbers 5 and 6 as file channels rather than as bases (see Chapter 10). READEM would then be a macro that could read variables from different files. Neither FORTRAN nor BASIC has a statement allowing multidevice reads. We will see more of this concept in the next chapter.

The last special pseudo-op we will consider is *concatenation*. Concatenation allows us to pass variable prefixes, roots, and/or suffixes to macros.

The example below has a variable suffix. The program will pass a suffix, the letter E, to JUMP, yielding JUMPE. To signify suffix concatenation, an apostrophe appears before a formal parameter within the body of the macro (not in the darglist). The program below searches a data bank for all occurrences of VALUE.

```
        TITLE EXAMPLE 31                    ;concatenation
        SEARCH MONSYM
DATA:   EXP 3,4,1,3,2,5
        NUM=6                               ;6 pieces of data
        VALUE=3                             ;value to be compared

        DEFINE COMPAR(this,that,rel,%LABEL,%DONE)<
        MOVE 4, that
        SUB  4, this
        JUMP'rel 4, %LABEL                  ;concatenating
        JRST %DONE
%LABEL: MOVEI 1, .PRIIN
        MOVEI 3, 12
        MOVE 2, that
        NOUT%
         HALTF%
        HRROI 1, [1586+12813]               ;crlf
        PSOUT%
%DONE:  >

START:  MOVNI 5, NUM
        MOVEI 10, VALUE                     ;compare this
AGAIN:  MOVE 6, DATA+NUM(5)
        COMPAR (10, 6,E)                    ;show all VALUEs
        AOJL 5, AGAIN
        HALTF%
        END START
```

To find all the numbers greater than or equal to VALUE, you can call COMPAR(10,6,GE). Concatenation makes it unnecessary to write separate programs to examine different aspects of the same data. An apostrophe after the formal parameter in the body of the macro allows for a prefix variable. Apostrophes before and after the formal parameter in the body of the macro will pass a root.

9.4 Universals—External Macros

This section explains how to make macros available to more than one program. To call "external" macros, the macros must be in a *universal* file and not in another program. You must write a universal file in the editor; an example is MYSTUF.MAC. MYSTUF is a collection of commonly used macros. To signify that a file is universal, use the keyword UNIVERSAL instead of TITLE. As in other files, universals must end with END, with no label following END. After editing the file, the system must compile it. To compile, write the following at TOPS-20.

```
@COMP MYSTUF.MAC
```

The compiler creates a file MYSTUF.UNV (note the extension), which makes the universal file available to other programs. Every time you edit a universal, you must remember to compile the new version. To use a macro that is in MYSTUF, a program must include the directive SEARCH MYSTUF. You could also write it as SEARCH MONSYM, MYSTUF. If the assembler cannot find a symbol in its local table, it will search universal files in the order in which they appear after the directive SEARCH. When executing a program, you need not link universal files. For example, if a program PROG.MAC includes the line SEARCH MYSTUF, the following simple line will execute it.

```
@EX PROG.MAC
```

The following line will *not* work.

```
@EX PROG.MAC,MYSTUF.UNV
```

If you recompile a universal file (add something new), you also need to recompile any program that used the universal.

```
@EX PROG.MAC/COMP
```

Universal files should use macros like PRINT, INPUT, READ, and so on. Some macros are also available in a system universal file titled MACSYM.

9.5 Summary

Proper utilization of macros can save a programmer much time. The customization permitted with macros allows the programmer to design his/her own software language. Once the macros are written for their purpose of multiple usage, the programmer is then free to concentrate more on the essence of an algorithm rather than on the lower level details of the hardware. Not all assembly languages have macro capabilities; without macros usage of system facilities becomes very difficult, however.

In the next chapter we will demonstrate how macro can be used to communicate with secondary storage units via files.

9.6 Exercises

1. Using macros, design a "language" with the following tokens: PRINT, SHOW, GET, and IF. The syntax of the language is:

```
PRINT mess                  ; mess is an ASCII string
SHOW var                    ; var is a numerical variable
GET var                     ; accepts a number from the tty
                            ;and places it in var
ADD ans, var1, var2         ; adds number var1 to var2
                            ;placing their sum in ans
IF var, less, eq, great     ; an arithmetic jump
                                    jump to less if var <0
                                    jump to eq if var=0
                                    jump to great if var >0
```

   Write a program that will accept (GET) two numbers, add them, and print (SHOW) the phases NEG, ZERO, or PLUS based on the value of the sum of the numbers.

2. Write a macro that simulates the MID$ function in BASIC. (In other words, pass to the macro MID$ a pointer and two numbers. The pointer points to a string, the first number is the beginning position of the partial string, and the second number is the length of the partial string.

```
Example:    P ---> THE MOON IS BLUE
            MID$(P,4,3) = MOO
```

3. Write a macro that will accept a string pointer and return the number of characters in the string being pointed to.
4. Write a macro SQUARE(in, out). This macro places the square of the parameter "in" into the parameter "out." Place this macro in a universal file. Write a program that invokes this macro.
5. Write a macro that accepts a string pointer and a character. The macro will return the numerical position where the character occurs in the string. If the character is not in the string, return a 0.

CHAPTER

10

Files

A *file* is a data structure. Unlike an array, it does not have a declared length; theoretically, it can be of infinite length. Files that are executable are called *programs*. Other files that simply contain data are called *data files*. Data files allow data to be independent of programs. More interestingly, files make data available to many programs. To use a file, a program must establish a connection with it through a Job File Number, JFN. The program must also establish the direction(s) of the flow of data (to the program and/or from the program). A high-level language would establish the flow of data with one of the following statements.

```
RESET(filevar, filename)
```

or

```
OPEN filename AS channel, mode
```

In the first example, RESET implies that the flow of data is from the file to the program. The computer assigns the JFN "filevar" to the previously

established file referenced in "filename." In the second example, OPEN implies that a file is to be referenced. It designates the file by "filename." The user picks the JFN by the integer value of the variable "channel." The value of "mode" determines the flow of data. Once the file is established, the JFN becomes its reference. For instance, READ or WRITE plus a JFN will perform I/O.

```
READ(filevar, variables)
```

or

```
WRITE #channel, variables
```

Assembly language uses similar operations. The monitor call GTJFN% (GeT a Job File Number) establishes the link between the program and the file. Another monitor call, OPENF%, (OPEN the File) links a file to a program. Once a file is open, all references to it use the JFN.

10.1 Files in Assembly

The DECSYSTEM-20 establishes a job file number using either a long form or a short form. We will discuss only the short form here since it is sufficient for most users. GTJFN% uses two accumulators, and it returns +1, +2. Before assigning the JFN you must specify a number of possibilities using *flags*. These flags, which define a file as a NEW file, an OLD file, and so on, go in accumulator 1. Table 10-1 lists the flags that we will deal with in this chapter.

TABLE 10-1. Some GTJFN% Flags

Symbol	*Bit*	*Meaning*
GJ%SHT	17	Short form. This flag must be set.
GJ%NEW	2	Create a file; if the file exists, the GTJFN% will return +1.
GJ%OLD	3	The file must exist.
GJ%FOU	0	Updating a file (new generation); if the file does not exist, the program will create it.
GJ%FNS	16	The terminal supplies the name of the file.

Below is a typical flag setup.

```
MOVE 1, [GJ%SHT+GJ%OLD]
```

Ac 2 references the name of the file. You may refer to the file name in two ways: If the terminal is to supply the file name at run time, then you set up accumulator 2 as follows.

```
MOVE 2, [.PRIIN,,.PRIOU]
```

You must set the flag GJ%FNS when using the terminal to supply the file name. (This flag can reference other JFNs as well, though we will not discuss this use here.) The second way to supply the file name is to pack the name in the program and reference it using a byte pointer. This method, illustrated below, does not use the GJ%FNS flag.

```
PTR: POINT 7, NAME
...
NAME: ASCIZ/filename/
...
MOVE 2, PTR
```

The string between the slashes must be an exact match—that is, use no extra spaces. Once the accumulators are set up, you can execute GTJFN%. Don't forget the +1, +2 return. If the GTJFN% is successful, it will place the value of the JFN in accumulator 1. You will then need to MOVE the JFN since you will need ac 1 for further use.

Having established a JFN, you must fix a direction to it using the monitor call OPENF%. OPENF% also uses two accumulators; place the JFN of the file to be opened in accumulator 1 and designate the direction(s) of the data flow by flags in accumulator 2. (See Table 10-2.)

TABLE 10-2. Some OPENF% Flags

Symbol	*Meaning*
OF%RD	Read the file.
OF%WR	Write to the file.
OF%APP	Append the file.

You must place the byte size of the data in bit 5 of ac 2. To read an ASCII file, you should use this command.

```
MOVE 2, [OF%RD+7B5]
```

After you have set up both accumulators, you can execute the monitor call OPENF%, which returns +1, +2.

Once a file is open, you can transfer data using monitor calls (similar to PBIN% and PBOUT%) that exchange a data byte between a file and a program. The simplest calls are BIN% and BOUT%. These calls use two accumulators: a JFN in ac 1, and the data byte to be transferred in ac 2.

Once a program has used a file, you need to disconnect the file from the program. The monitor call CLOSF% (CLOSe the File) places the JFN in accumulator 1 and then returns +1, +2. (We need not discuss the flags available for CLOSF% here.)

In dealing with files, good programming practice is to begin all programs with the monitor call RESET%. This call closes any existing open files and also initializes the program's address space. Below is a program that will create a file, getting data for the file from the terminal.

```
        TITLE EXAMPLE 32                    ;creating a file
        SEARCH MONSYM
JFN:    Z                                   ;save JFN here
        AMT=5                               ;5 pieces of data
START:  RESET%
        MOVEI 4, AMT                        ;set up a counter
        HRROI 1, [ASCIZ/ new file name /]
        PSOUT%

        MOVE 1, [GJ%SHT+GJ%FOU+GJ%FNS]      ;set up GTJFN%
        MOVE 2, [.PRIIN,,.PRIOU]
        GTJFN%
         HALTF%
        MOVEM 1, JFN                        ;save JFN

                                            ;set up OPENF%
        MOVE 1, JFN
        MOVE 2,[OF%WR+7B5]                  ;writing in ASCII
        OPENF%
         HALTF%

        HRROI 1, [ASCIZ/ type in 5 chars /]
        PSOUT%
        HRROI 1, [12B6+15B13]               ;crlf
        PSOUT%
```

```
LOOP:   PBIN%                           ;get a char
                                        ;set up BOUT%
        MOVE 2, 1                       ;put char in ac 2
        MOVE 1, JFN                     ;get JFN
        BOUT%
        SOJG 4, LOOP


                                        ;close file
        MOVE 1, JFN
        CLOSF%
         HALTF%
        HALTF%
        END START
```

Execution of this program will print the prompt "new file name" at the terminal. If you type JUNK.DAT, the system will create a file with that name. Next, the prompt "type in 5 chars" will appear at the terminal. You can then press any five keys, as illustrated below.

```
new file name  JUNK.DAT
type in 5 chars
ABCDE
```

After typing the fifth key, the next prompt will be @ at TOPS-20. Since the file is in ASCII, you can examine the file with the TOPS-20 command TYPE JUNK.DAT. The file contents, ABCDE, will then appear at the terminal. You can also use other TOPS-20 commands—like EDIT—on the file.

The next example is a program that reads an existing file. This time the name of the file is in the program. Placing the string JUNK.DAT in the program will cause the terminal to display five characters from that file.

```
        TITLE EXAMPLE 33                        ;reading a file
        SEARCH MONSYM
        AMT=5
JFN:    Z
START:  RESET%
        MOVEI 4, AMT

                                                ;set up GTJFN%
        MOVE 1, [GJ%SHT+GJ%OLD]
        MOVE 2, [POINT 7,[ASCIZ/Junk.dat/]]
        GTJFN%
         HALTF%
        MOVEM 1, JFN

                                                ;set up OPENF%
        MOVE 1, JFN
        MOVE 2, [OF%RD+7B5]
        OPENF%
         HALTF%

LOOP:   MOVE 1, JFN
        BIN%                                    ;get a char
        MOVE 1,2
        PBOUT%                                  ;show char
        SOJG 4, LOOP

                                                ;close file
        MOVE 1, JFN
        CLOSF%
         HALTF%
        HALTF%
        END START
```

The programs above transferred ASCII. To transfer numbers, the byte size in OPENF% should be 44 (decimal 36) since each number uses a full word. NIN% will read the numbers into the program from the terminal, and BOUT% will transfer them to a file. Similarly, BIN% (byte size 44) will read numbers from a file, and NOUT% will transfer them to the terminal. A file created with a byte size other than 7 bits will not be readable via the TOPS-20 command TYPE.

10.2 Macros and Byte Files

Several macros can help handle files. The first macro we will discuss uses the GTJFN% call. This call can designate the file name with flags or by returning a JFN. Because the number of flags can vary, an IRP is necessary. The assembly directives IFIDN and IFDIF are used to designate the file.

These directives compare strings: IF the strings are IDeNtical, the assembler uses the code following IFIDN; IF they are DIFferent, it uses IFDIF. These directives take the following forms:

```
IFIDN <string1><string2>, <code>

IFDIF <string1><string2>, <code>
```

Generally, one string is fixed and the other is a parameter of a macro. Since all flags begin with GJ%, suffix concatenation will pass the rest of the flag.

```
DEFINE GETJFN(JFN,FLAG,IO<TTY>)<
     IFIDN <IO><TTY>,<HRROI 1, [ASCIZ/file name/]
                      PSOUT%
                      MOVE 2, [.PRIIN,,.PRIOU]
                      SETZ 1,
                      IOR 1, [GJ%FNS+GJ%SHT]>
     IFOIF <IO><TTY>,<MOVE 2, [POINT 7,[ASCIZ/IO/]]
          SETZ 1,
          IOR 1, [GJ%SHT]>
     IRP FLAG, <IOR 1, [GJ%'FLAG]>
     GTJFN%
        ERJMP [PRINT get JFN error
          HALTF%]
     MOVEM 1, JFN>
```

The following are examples of calls to GETJFN.

```
GETJFN(var,old)

GETJFN(var,new,junk.dat)
```

The first example indicates that "var" contains the JFN, that the file exists, and that the terminal supplies the name of the file. The second example shows that "var" contains the JFN, that the file will be created, and that the file name is "JUNK.DAT." Below is a macro for OPENF%.

```
DEFINE FOPEN(JFN,FLAG,WORD<7>)<
        MOVE 1, JFN
        MOVE 2, [<WORD>B5]
        IRP FLAG,<IOR 2,[OF%'FLAG]>
        OPENF%
         ERJMP [PRINT OPENF% error
                HALTF%]
        >
```

The examples below are calls to FOPEN.

```
FOPEN(var,rd)

FOPEN(var,<rd,wr>,44)
```

In the first example, "var" contains the JFN, the file is open for reading, and the byte size is 7 bits. In the second example, "var" contains the JFN, the file is open for reading and writing, and the byte size is a full word.

You can use the above macros to construct larger macros. For instance, the following example establishes a reading link to a file that already exists.

```
DEFINE REREAD(var,filename)<
        GETJFN(var,old,filename)
        FOPEN(var,rd)>
```

The file name is "filename" and the JFN "var" handles all references to the file. Here is a macro for writing to a file.

```
DEFINE REWRIT(var,filename)<
        GETJFN(var,fou,filename)
        FOPEN(var,wr)>
```

The "fou" flag will create a new generation and instruct the system to ignore any previous versions. Below is a simple macro to close a file.

```
DEFINE CLOSE(var)<
        MOVE 1, var
        CLOSF%
         PRINT closing error
        >
```

Let's repeat the examples above, creating and reading a file, using macros. The macros are in a universal file MYSTUF. The examples above used a specific number of characters (five). In general, however, because you will not know the number of bytes in a file, an end-of-file marker is available. This marker is similar to a null: Not all nulls are end-of-file markers, but all end-of-file markers are nulls. The monitor call GTSTS% (GeT file STatuS) will signify if a null is a null or an end-of-file marker. GTSTS% uses two accumulators: accumulator 1 contains the JFN of the file to be checked;

upon completion of GTSTS%, accumulator 2 contains the status of the file. GTSTS% always returns +1. You can use several flags with GTSTS%, but we will use only bit 8, GS%EOF, the end-of-file bit. Setting bit 8 indicates the end of the file. The following example creates a file and places in it five characters from the terminal; it then closes the file, reopens it, and reads it. The example uses GTSTS% to signify the end of the file rather than to count characters.

```
        TITLE EXAMPLE 34              ;using reread and rewrit
        SEARCH MONSYM,MYSTUF
        AMT=5                         ;read in 5 chars

START:  RESET%
        MOVEI 5, AMT
        REWRIT(JFN#,JUNK.DAT)         ;create the file
        PRINT type in 5 chars

LOOP:   PBIN%                         ;get a char
        MOVE 2,1                      ;set up BOUT%
        MOVE 1, JFN
        BOUT%                         ;write the char to the file
        SOJG 5, LOOP                  ;more chars?
        CLOSE(JFN)                    ;close the file
        CRLF                          ;send a crlf to terminal

        REREAD(JFN,JUNK.DAT)          ;reopen the file
AGAIN:  MOVE 1, JFN                   ;set up BIN%
        BIN%                          ;get a char
        SKIPN 2                       ;char a null
         JSR CHECK                    ;yes, then eof?
        MOVE 1,2                      ;set up PBOUT%
        PBOUT%
        CRLF                          ;place a crlf between chars
        JRST AGAIN                    ;keep reading file

CHECK:  Z                             ;subroutine eof,
                                      ;ac 1 must contain JFN
        GTSTS%                        ;get file status
        TLNE 2, (GS%EOF)              ;bit 8 set?
          JRST EOF                    ;yes, goto eof routine
        SETZ 2,                       ;no, return ac 2 to null
        JRST @CHECK                   ;return

EOF:    CLOSE(JFN)

        HALTF%
        END START
```

Because a program is an ASCII file, we could open a program file and alter it. For example, we could edit the file by changing all E's to I's. First we use BIN% to instruct the computer to read each character in the file. However, when BIN% reads an E, we cannot simply BOUT% an I to the file because the file has a byte pointer. Each BIN% increments and loads a byte from the file. Therefore, when BIN% reads the E, the file pointer passes it, meaning that BOUT% will simply place an I over the character following E. A temporary file is one way to handle this problem. To open the file, we BIN% a character from the first file to the program. If the character is not an E, we BOUT% the character to the temporary file. If the character is an E, we BOUT% an I to the temporary file. After the system has completely read the original file, we must close both files. At TOPS-20, we *rename* the temporary file as the original file. This approach is typical in handling sequential access files.

Random byte access uses JSYS, RIN%, and ROUT% to control the file pointer. These calls use three accumulators: accumulator 1 contains the JFN, accumulator 2 handles byte transfer, and accumulator 3 contains the position of the byte being referenced within the file. RIN% and ROUT% return +1 and only work with disk files (not with tapes, for example). To edit a file with RIN% and ROUT%, you simply increment the file pointer, RIN% a byte, and check to see if it is an E. If the byte is an E, then you ROUT% an I. The I will then replace the E because the file pointer still points to the E position. If the byte is not an E, then you must increment the file pointer and RIN% the next byte. This process continues to the end of the file. To use this approach, you must open a file for reading and writing. The following macro, UPDATE, will accomplish this task.

```
DEFINE UPDATE(var,filename)<
        GETJFN(var,old,filename)
        FOPEN (var,<rd,wr>) >
```

The example below programs the concepts we have been discussing.

```
            TITLE EXAMPLE 35          ;random access files
                                      ;change all E's to I's
            SEARCH MONSYM, MYSTUF
JFN:        Z
OLDLET:     "E"
NEWLET:     "I"
START:      RESET%
            MOVNI 3,1                 ;set file pointer to -1
            UPDATE(JFN,TTY)
```

```
LOOP:   MOVE 1, JFN
        ADDI 3,1                ;increment file pointer
        RIN%                    ;get a byte
        SKIPN 2                 ;is it a null
         JSR CHECK              ;yes, then check eof
        CAMN 2, OLDLET          ;is byte an E?
         JSR FIX                ;yes, then change it
        MOVE 1,2
        PBOUT%                  ;show byte to tty
        JRST LOOP               ;keep fetching bytes
FIX:    Z                       ;changing bytes
        MOVE 2, NEWLET
        ROUT%
        JRST @FIX

CHECK:  Z                       ;subroutine eof
        GTSTS%                  ;get file status
        TLNE 2, (GS%EOF)        ;bit 8 set?
          JRST EOF              ;yes, goto eof routine
        SETZ 2,                 ;no, return ac 2 to null
        JRST @CHECK             ;return
EOF:    HALTF%
        END START
```

10.3 Summary

Executable files are known as programs, whereas data files merely contain data, thereby allowing data to be independent of programs. From an operating-system point of view, files can be considered as fundamental units—either program, data, or device (TTY). An assembly file perspective allows for operations on files in more finely tuned detail than is possible in high-level languages. Presently, however, there is a trend in high-level languages (MODULA-2, ADA) to build into the language the more precise file operations that are available in assembly.

In the next chapter we will see how assembly deals with files and devices in a real-time situation.

10.4 Exercises

1. Using the system editor, create a file and place three numbers in it. Write a program that will access this file and display its contents.
2. Write a program that will append three numbers to an existing file. Write the program to supply the name of the file and the three numbers at run time.

3. Using the system editor, create two files. Write a program that will merge the files into a new file (simply place one after the other). Use the JSYS DELF% to delete the other two files. DELF% takes a JFN in accumulator 1 and returns +1, +2.
4. Using the system editor, create two files containing numbers sorted in ascending order. Write a program that will merge the numbers into one sorted file.

   Example: file1 3 8 10
   file2 2 7 9
   file3 2 3 7 8 9 10

5. Write a simple editor program with a substitute command to replace one character with another. Determine the file to be edited at run time.

CHAPTER

11

Interrupts

In a number of situations, you may want to allow a program to receive interruptions, or *interrupts*, from various sources, such as the terminal or other parts of the operating system. This chapter discusses the uses of interrupts and how you can set up procedures for a program to inform the operating system of software interrupts.

11.1 An Analogy

To understand interrupts, let us consider the analogy of how a teacher might handle interruptions in a classroom of polite students. If a student raises his/her hand while the teacher is lecturing, the teacher has several options. He or she can stop in the middle of a thought and acknowledge the student. He or she can finish a point of discussion and then pause to recognize the student's request. Or the teacher can ignore the student and continue lecturing. Generally, a teacher will handle a student interrupt in the second way—by completing a thought and then recognizing the student. This approach allows the teacher a natural break in the lec-

ture and easy continuation after he or she has dealt with the student's request. (Of course, this method is not practical if the student is trying to tell the teacher that the room is on fire, in which case the message justifies interrupting a train of thought.)

If a second student attempts to interrupt while the teacher is responding to the first student, the situation is more complicated. The teacher can ignore the second student, finish with the first student, and then acknowledge the second student. Or, at an appropriate point, the teacher can postpone his or her response to the first student, deal with the second student, and then return to the first student. The teacher will take the latter approach if the second student has higher *priority* in some sense than the first student.

Let's look at computer interrupts using this analogy. Think of the teacher as your program and the students as peripheral devices (TTY, card readers, and so on) and/or other procedures. Actually, you should think of your program as a blind teacher who cannot see any raised hands! You should also imagine a "big brother," the operating system, standing in the corner of the classroom watching the students and helping the teacher. The blind teacher tells big brother under what conditions he/she will accept interrupts. (Until now, none of your programs has instructed the operating system to accept interrupts.) The operating system must control your program and watch for interrupts simultaneously. It accomplishes this dual function by checking before fetching each instruction in your program to see if anything wishes to interrupt your program. If it finds no interrupts, it proceeds to fetch and execute an instruction. It will continue to check for interrupts before each instruction. If an interrupt occurs, the system checks whether your program is set up to handle the interrupt requesting service. If your program is not accepting interrupts, the system must decide what to do. The system will always acknowledge certain interrupts, such as a reference to a nonexisting location, or an illegal operation. If your program is not prepared to handle these interrupts (*panic interrupts*), its continuation would cause a serious error. Therefore, the system takes over, generally by stopping the program and reporting an error. This interrupt is not analogous to a student's yelling "fire," however, which would be a *hardware interrupt*. We are talking about a *software interrupt*, which can occur only after completion of an instruction, not in the middle. The system handles the "fire"-type (hardware) interrupts as well as (software) panic interrupts (such as a bad address reference) and (software) nonpanic interrupts (such as end of a file).

If you allow interrupts to occur, you must know how to handle them. Interrupts can come from many different sources. Reasons for interrupts include: illegal instruction, end of file, stack overflow, and a particular key from the terminal. Generally, each source of an interrupt demands a specific procedure. You must supply these procedures in your program.

In some cases, *polling* may be an alternative to interrupts: After finishing a point, the blind teacher could ask each student, one at a time, if he or she has any questions. This method demands much more time to get through a (boring) lecture.

11.2 Setting up Interrupts

Each source of an interrupt connects to your program via a channel (see Table 11-1). The system has 36 channels, with some channels preassigned to a specific type of interrupt (for example, end of file is assigned to channel 10).

Each interrupt that occurs on a channel has associated with it a procedure referenced by your program. An *interrupt channel table* establishes this connection by setting aside 36 consecutive locations, one for each channel. All 36 words must have data. The right half of each location

TABLE 11-1. Software Interrupt Channel Assignments

Channel	*Symbol*	*Meaning*
0–5		Assignable by user program
6	.ICAOV	Arithmetic overflow
7	.ICFOV	Arithmetic floating point overflow
8		Reserved for DEC
9	.ICPOV	Pushdown list overflow (panic channel)
10	.ICEOF	End-of-file condition
11	.ICDAE	Data error file condition (panic channel)
12–14		Reserved for DEC
15	.ICILI	Illegal instruction (panic)
16	.ICIRD	Illegal memory read (panic)
17	.ICIWR	Illegal memory write (panic)
18		Reserved for DEC
19	.ICIFT	Inferior process termination
20	.ICMSE	System resources exhausted (panic channel)
21		Reserved for DEC
22	.ICNXP	Nonexistent page reference
23–35		Assignable by user program

contains the starting address of the procedure for the corresponding interrupt channel. The words corresponding to the channels that your program is not going to monitor contain a 0. The left half of these 36 words establishes the priority of an interrupt in case it is interrupted by another interrupt. You must be concerned with three levels of priority: 1, 2, and 3, with 1 the highest priority. If your program is handling a priority 2 interrupt and a level 3 interrupt occurs, the level 3 interrupt must wait until completion of level 2. If a level 1 occurs while your program is servicing a level 2, the system will interrupt the level 2 and then return to it when the level 1 is completed. If two interrupts of the same level occur, the system will service them on a "first come, first served" basis.

An interrupt will cause the system to save the present contents of the program counter so that it can execute a "return" (if desired) after completion of the interrupt. The program must set aside three locations (words)—one location for each priority level—for the PC when an interrupt occurs. The *priority level table* references these locations. The priority table also shows where to store a PC during an interrupt. Therefore, six table locations deal with priority—the three table locations and the three locations referenced by the table.

To better understand how the channel table and the level table work with interrupts, let's discuss an example. Assume that the CHaNnel TABle starts at location CHNTAB and the LEVel TABle starts at location LEVTAB. Assume also that the program is running and an interrupt occurs on channel 10, indicating, for example, that the last piece of data has been read from a file. The operating system checks whether your program will handle this interrupt. If so, it retrieves the word at location "CHNTAB + interrupt channel." The left half of this word indicates that the priority level for this interrupt is 2. The system then saves the present PC in the second position referenced by the priority level table—the location pointed to by "LEVTAB + priority − 1." It places the right half of "CHNTAB + interrupt channel," the starting address of the procedure handling the interrupt, in the PC. If another interrupt occurs while servicing this interrupt, the operating system retrieves the appropriate word from the CHNTAB. If the left half of this word has a higher priority, say level 1, than the present position of the PC (somewhere within the level 2 interrupt procedure), the system saves it in the address referenced by the first position in the priority level table. It places the right half of the CHNTAB word in the PC, the starting address of this level 1 procedure. When the level 1 interrupt has ended, the program picks up in the middle of the previous level 2 interrupt by placing the contents pointed to by the first position

of the priority level table, location @LEVTAB, in the PC. When this process ends, the system places the contents of the second position in the priority level table into the PC, and the program continues from the point at which the level 2 interrupt occurred.

You must let the system know the locations of the interrupt tables by using the monitor call SIR% (Software INTERrupt). This call uses two accumulators. The DECSYSTEM-20 allows more than one process (program) to run at the same time. You must identify the process, or FORK, that will handle interrupts and place this identification in ac 1. For now, only the present program will handle interrupts, so you use .FHSLF (Fork Handle SELF). SIR% needs the location of the priority level table in the left half of ac 2 and the channel table address in the right half of ac 2. It returns +1.

```
MOVEI 1, .FHSLF
MOVE 2, [LEVTAB,,CHNTAB]
SIR%
```

SIR%'s job is to inform the operating system where the interrupt tables are located. The next step is to enable the interrupt system. Using our classroom analogy, consider the fact that interrupts can come from many directions (students, public address system, and so on) and that the teacher may not wish to be interrupted by anyone or anything at certain points in a lecture. (A program might ignore all nonpanic interrupts in order to continue accepting data from a device about to shut down.) Or at times, the teacher will ignore only the public address system but acknowledge other interruptions. (A program might temporarily ignore interrupts from one device, such as the keyboard, but accept interrupts from other devices.) Thus, given the many sources of interrupts, it is nice to be able to shut down all interrupts or only certain channels with relative ease, and assembly JSYS exist for this purpose. The commands EIR% and DIR% (Enable INTERrupts, Disable INTERrupts) govern interrupts; the commands AIC% and DIC% (Activate Interrupt Channel, Deactivate Interrupt Channel) govern channels. In addition, ATI% and DTI% (Activate Terminal Interrupt, Deactivate Terminal Interrupt) will assign the terminal to an interrupt channel if necessary.

After SIR%, you may need an EIR% and an AIC% and possibly an ATI% to complete an interrupt setup. In using these directives, you must observe a hierarchy: SIR%, EIR%, AIC%, ATI%. In other words, an AIC% will not take effect until execution of an EIR%. An ATI% will not take effect until execution of an AIC%. The EIR%/DIR% use one accumulator, ac 1, which

must contain the fork to be enabled or disabled. EIR% and DIR% return +1. Here is an example of an EIR%.

```
MOVEI 1, .FHSLF
EIR%
```

To set up DIC% or AIC%, you need two accumulators: ac 1 contains the fork and ac 2 the channel(s) to be (de)activated. The following lines would disable channels 6 and 9.

```
MOVEI 1, .FHSLF
MOVE 2, [1B6+1B9] or MOVE 2, [1B<.ICAOV>+1B<.ICPOV>]
DIC%
```

DIC% and AIC% return +1. ATI%, which allows for terminal interrupts, uses only one accumulator. Your program places the ASCII value of the key that will cause the interrupt in the left half of ac 1 and an assignable channel number in the right half. Assignable channel numbers are 0 to 5 and 23 to 35. Table 11-2 lists the codes and conditions used in terminal interrupts.

Don't forget, AIC% must at some time activate the channel used by ATI%. ATI% returns +1. Below is an example assigning a control A to channel 0. DTI% works in a similar manner.

```
MOVE 1, [1b1,,0]  or MOVE 1, [.TICCA,,0]
ATI%
```

The following program will constantly send the letter A to the TTY. When a control E is typed the program will then be interrupted, print "got interrupt," and stop. A delay LOOP avoids overburdening the terminal with A's. This program also uses the assembly directive REPEAT:

REPEAT number of times, <code>

This directive instructs the computer to repeat the value between the angle brackets, code, a specified number of times.

```
        TITLE EXAMPLE 36                        ;simple interrupt
                                                ;sends the letter A
                                                ;to tty until a control
                                                ;E is typed
        SEARCH MONSYM
CHNTAB: 2,,CNTE                                 ;channel table
        REPEAT ^D35,<0>                         ;fill rest of table with 0
LEVTAB: 0                                       ;no level 1
        PC2                                     ;point to level 2
        0                                       ;no level 3
PC2:    Z                                       ;save PC here
COUNT:  ^D100000

START:  RESET%
        MOVEI 1, .FHSLF                         ;set up SIR%
        MOVE 2, [LEVTAB,,CHNTAB]
        SIR%
        MOVEI 1, .FHSLF                         ;set up EIR%
        EIR%
        MOVEI 1, .FHSLF                         ;set up AIC%
        MOVE 2, [1B0]
        AIC%
                                                ;set up a terminal
                                                ;interrupt
        MOVE 1,[5,,0]                           ;control E channel 0
        ATI%
BEGIN:  MOVE 2, COUNT                           ;set up counter
LOOP:   SOJG 2, LOOP                            ;simple delay loop
        MOVEI 1, "A"                            ;send letter A to
        PBOUT%                                  ;tty
        JRST BEGIN

                                                ;here on a control E
CNTE:   HRROI 1, [ASCIZ/ got interrupt/ ]
        PSOUT%
        HALTF%
        END START
```

For the program above, only the first MOVEI 1, .FHSLF is necessary.

11.3 Interrupts and Macros

Some macros are available to help you set up interrupts. Let's design a macro for interrupts that uses four formal parameters: a procedure address, a priority value, a channel, and an optional TTY key. For instance, example 36 used a procedure address of CNTE, established a priority level of 2,

TABLE 11-2. Terminal Codes and Conditions

Code	*Symbol*	*Character or Condition*
0	.TICBK	Control @
1	.TICCA	Control A
2	.TICCB	Control B
3	.TICCC	Control C
4	.TICCD	Control D
5	.TICCE	Control E
6	.TICCF	Control F
7	.TICCG	Control G
8	.TICCH	Control H
9	.TICCI	Control I
10	.TICCJ	Control J
11	.TICCK	Control K
12	.TICCL	Control L
13	.TICCM	Control M
14	.TICCN	Control N
15	.TICCO	Control O
16	.TICCP	Control P
17	.TICCQ	Control Q
18	.TICCR	Control R
19	.TICCS	Control S
20	.TICCT	Control T
21	.TICCU	Control U
22	.TICCV	Control V
23	.TICCW	Control W
24	.TICCX	Control X
25	.TICCY	Control Y
26	.TICCZ	Control Z
27	.TICCEX	ESC key
28	.TICRB	Delete key
29	.TICSP	Space
30	.TICRF	Dataset carrier off
31	.TICTI	Typein (any key)
32	.TICTO	Typeout (any key)
33–35		Reserved

used the channel 0, and used a key, control E. Below is a call that will set up an interrupt.

```
INTER(CNTE,2,0,E)
```

If you need an end-of-file interrupt, you could set it up as follows.

```
INTER(EOF,2,10)
```

To make this problem manageable, assume that CHNTAB and LEVTAB are the labels for the interrupt tables. You, as the user, are responsible for setting up the following (although more relaxed requirements are possible).

```
CHNTAB:    BLOCK ^036
LEVTAB:    PC1
           PC2
           PC3
PC1:       Z
PC2:       Z
PC3:       Z
```

The macro INTER contains the following macros:

- FIXCHN places the necessary word in the channel table.
- SETEIR enables interrupts.
- SETAIC activates a channel.
- SETKEY performs an ATI%. The computer will call this macro only if a key parameter is present.

(We will discuss the SIR% part of this macro later.)

FIXCHN has three parameters: the procedure address, the priority, and a channel.

```
DEFINE FIXCHN(PROC,PRI,CHAN)<
        MOVE 1, [PRI,,PROC]
        MOVEI 2, ^D'CHAN
        MOVEM 1, CHNTAB(2)
        >
```

Note that the program concatenates the using channel value with ^D. This value indexes into the correct position within the CHNTAB block.

The macros SETEIR and SETAIC are straightforward.

```
DEFINE SETEIR(FORK<,FHSLF>)<
        MOVEI 1, FORK
        EIR%
          erjmp [print EIR% error]>

DEFINE SETAIC(CHAN,FORK<,FHSLF>)<
        MOVEI 1, FORK
        MOVE 2, [1B'CHAN]
        AIC%
         erjmp [ print AIC% error ]>
```

Note again how concatenation connects the channel value to the assembly directive B. (Though both EIR% and AIC% return +1, an ERJMP will exe-

cute only in the event of an error.) The SETKEY macro uses concatenation, both in suffix and root mode. An apostrophe on both sides of the variable "key" places the key in quotes, which instructs the assembler to transform it to its ASCII value. Subtracting octal 100 from the ASCII value converts the key into a control key value. (In other words, subtracting octal 100 from the octal value of A, or 101, yields 1, which is an ASCII control A.)

```
DEFINE SETKEY(KEY,CHAN)<
        HRLZI 1, "'KEY'"-100
        HRRI 1, ^D'CHAN
        ATI%
         erjmp [print ATI% error ]>
```

We can now define INTER. Note that a default value is assigned to the key parameter. The program calls SETKEY only if the parameter key is not the default value. Calling SETKEY is accomplished by using an IFDIF.

```
DEFINE INTER(PROC,PRI,CHAN,KEY<>)<
        FIXCHN(PROC,PRI,CHAN)
        SETEIR
        SETAIC(CHAN)
        IFDIF <KEY><>,<SETKEY(KEY,CHAN)> >
```

To set up more than one interrupt at a time you need to IRP INTER; because INTER has more than one parameter, you must use the macro COMB (see section 9.4). The macro INTERRUPT can now establish interrupts. Because of IRP, INTERRUPT will have only one parameter, which in turn will contain the four values needed by INTER. INTERRUPT calls SIR% via a macro RDTAB. SIR% informs the operating system where to find CHNTAB and LEVTAB. The system will not read these tables until an interrupt occurs.

```
DEFINE RDTAB <
        MOVEI 1, .FHSLF
        MOVE 2, [LEVTAB,,CHNTAB]
        SIR%
         erjmp [print SIR% error ]>

DEFINE INTERRUPT (value)<
        RDTAB
        IRP value, <COMB INTER, value>>
```

Here are examples of calls to INTERRUPT.

```
INTERRUPT <<CNTE,2,0,E>>

INTERRUPT <<CNTE,2,0,E>,<EOF,2,10>>
```

The first call is the same as example 36. The second sets up two interrupts: one is the same as example 36, the other is a priority-level-2, end-of-file interrupt on channel 10.

The program in the following example will access a file and send the file to the TTY. An end-of-file interrupt will stop it, and a control E while the file is being printed will cause the message "got interrupt" to appear. The program will continue from the point of interruption because of the monitor call DEBRK%. DEBRK% is a return from interrupt.

```
            TITLE EXAMPLE 37          ;two interrupts
                                      ;end of file or
                                      ;control E

            SEARCH MONSYM,MYSTUF
CHNTAB:     BLOCK ^D36
LEVTAB:     0                         ;no level 1
            PC2                       ;point to level 2
            0                         ;no level 3
PC2:        Z                         ;save PC here

START:      RESET%
            INTERRUPT<<CNTE,2,0,E>,<EOF,2,10>>
            REREAD(JFN ,TTY)

                                      ;the following loop
                                      ;is stopped by a
                                      ;system generated
                                      ;end of file interrupt
LOOP:       MOVE 1, JFN
            BIN%
            MOVE 1,2
            PBOUT%
            JRST LOOP

                                      ;here on a control E
CNTE:       HRROI 1, [ASCIZ/
                    got interrupt
/ ]
            PSOUT%
            DEBRK%                    ;return from interrupt
```

```
                                        ;eof routine
EOF:        HRROI 1, [ASCIZ/ end of file /]
            PSOUT%
            CLOSE(JFN)
            HALTF%
            END START
```

We have used two methods to handle the end of an interrupt procedure: halting the program after the interrupt, and returning to the exact position of the interrupt within the program. Another possibility is to return to the program, but not to the location of the interrupt. This method is desirable when reading a file. Using interrupts, you need not check whether the byte is a null and then check to see whether it is the end of a file, as described in Chapter 10. Instead, the interrupt system can check for the end of a file. An interrupt will then cause the program to go to the end-of-file procedure. DEBRK% handles a return from interrupt by placing in the PC the contents of the address pointed to by the priority table. The interrupt routine simply changes the contents of this location priority level table and then DEBRK%S.

```
                            ;end-of-file interrupt
MOVEI 2, NEXT
MOVEM 2, @LEVTAB+1          ;assumed level 2
                            ;or MOVEM 2, PC2
DEBRK%
```

Now when an end-of-file interrupt occurs, the return from interrupt will go to NEXT rather than to the line where the interrupt occurred.

A time-sharing system will also return to a location other than the point of interruption. After a fixed amount of time, the system will stop one process temporarily, preserve its program counter, fetch another PC, and continue the process. The monitor call TIMER% will set a timed interrupt. It can cause an interrupt at a specific time or after an elapsed time. TIMER% uses the first three accumulators and returns +1, +2. The left half of accumulator 1 contains the process handle (.FHSLF), and the right half contains a function code. (We will discuss only the elapsed time code, code 1, here.) Accumulator 2 contains the time (or amount of time) to generate the interrupt. Accumulator 3 contains the software interrupt channel number. In the program below, loop FIRST continuously prints the message "in first program." After 2000 milliseconds, TIMER% generates an interrupt and the terminal prints the message "time is up." Next, the system exchanges program counters and transfers control to the loop

SECOND. This loop prints the message "in second program" for 2 seconds. After 2 seconds, the system generates an interrupt, prints "time is up," and then transfers control back to loop FIRST.

```
        TITLE EXAMPLE 38          ;scheduler
        SEARCH MONSYM,MYSTUF
CHNTAB: BLOCK ^D36
LEVTAB: 0
        PC2
        0
PC2:    Z
OLD:    SECOND
START:  RESET%
        PRINT STARTING
        CRLF
        INTERRUPT <<NEXT,2,0>>
        JSR SCHED                 ;set up TIMER%
FIRST:  PRINT IN FIRST PROGRAM
        CRLF
        JRST FIRST

SECOND: PRINT IN SECOND PROGRAM
        CRLF
        JRST SECOND


NEXT:   PRINT TIME IS UP
        CRLF
        JSR SCHED                 ;reset TIMER%
        MOVE 10, PC2              ;exchange old and new PC
        EXCH 10, OLD
        MOVEM 10, PC2
        DEBRK%


SCHED:  Z
        MOVE 1, [.FHSLF,,1]       ;this process,
                                  ;elapsed time

        MOVEI 2, 2000             ;2 seconds
        MOVEI 3, 0                ;channel zero
        TIMER%
         Z
        JRST @SCHED
        END START
```

If an interrupt routine must use accumulators, you must make sure that your program saves the original contents of the accumulators. Typically

the first line of an interrupt routine is a block transfer preserving acs. The last line is another block transfer returning the original values to the acs before calling DEBRK%.

11.4 Summary

Interrupts are asynchronous breaks within a program system capable of allowing for a resumption of the program at that exact point at some later moment. They allow a computer to control real-time situations. Some assembly interrupts include stack overflow, illegal instruction and end of file. Interrupt capabilities are strongly connected with computer hardware, whereas a high-level language, because of its need for portability, does not generally possess the ability to "get its hooks into the hardware." However, with the present interest in, and the inherent need for interrupts in, embedded systems, there is a trend towards supplying high-level languages (MODULA-2, ADA) with interrupt ability.

11.5 Exercises

1. Write a program that will search linearly through a file for a character. The program should supply both the file and the character at run time. If the character is present, print its numerical position. If the character is not present, trap the end-of-file interrupt and print "not found."
2. Set up a stack and the necessary conditions to trap a stack pushdown overflow. Test your program by using PUSH too often.
3. Set up a TIMER% interrupt so that the bell will ring every 30 seconds while a program is running. (For instance, insert the TIMER% routine into your program for exercise 1.)
4. Write a program that continuously writes a message to the screen. Set up a keyboard interrupt so that the program will stop whenever the user presses a key (.TICTI).
5. The JSYS TIME% is a millisecond clock. It places a monotonically increasing number in ac 1 (it also places a 1000 in ac 2). Write a program that will display the duration of an event. The event will start when the user presses a key, which should cause a keyboard interrupt that accesses TIME%. Store the value that is returned in ac 1. Stop the event when the user presses any key. This action should also cause an interrupt and call TIME%. The difference in the values of TIME% is the number of milliseconds between keystrokes.

APPENDIX

A

ASCII Codes for I/O

Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	[Remarks]
0	000	000	NUL	Null, tape feed. Control shift P.
1	001	001	SOH	Start of heading [SOM, start of message]. Control A.
1	002	002	STX	Start of text [EOA, end of address]. Control B.
0	003	003	ETX	End of text [EOM, end of message]. Control C.
1	004	004	EOT	End of transmission; shuts off TWX machines and disconnects some data sets. Control D.
0	005	005	ENQ	Enquiry [WRU, "Who are you?"]. Triggers identification ("Here is . . .") at remote station if so equipped. Control E.
0	006	006	ACK	Acknowledge [RU, "Are you . . .?"]. Control F.
1	007	007	BEL	Rings the bell. Control G.
1	008	010	BS	Backspace. Control H.
0	009	011	HT	Horizontal tab. Control I.
0	010	012	LF	Line feed. Control J.
1	011	013	VT	Vertical tab. Control K.

Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	[Remarks]
0	012	014	FF	Form feed to top of next page. Control L.
1	013	015	CR	Carriage return to beginning of line. Control M.
1	014	016	SO	Shift out; change character set or change ribbon color to red. Control N.
0	015	017	SI	Shift in; return to standard character set or color. Control O.
1	016	020	DLE	Data link escape [DCO]. Control P.
0	017	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	018	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	019	023	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	020	024	DC4	Device control 4 (stop), turns punch or auxiliary off. Control T (AUX OFF).
1	021	025	NAK	Negative acknowledge [ERR, error]. Control U.
1	022	026	SYN	Synchronous idle [SYNC]. Control V.
0	023	027	ETB	End of transmission block [LEM, logical end of medium]. Control W.
0	024	030	CAN	Cancel [S_0]. Control X.
1	025	031	EM	End of medium [S_1]. Control Y.
1	026	032	SUB	Substitute [S_2]. Control Z.
0	027	033	ESC	Escape, prefix [S_3]. Control shift K.
1	028	034	FS	File separator [S_4]. Control shift L.
0	029	035	GS	Group separator [S_5]. Control shift M.
0	030	036	RS	Record separator [S_6]. Control shift N.
1	031	037	US	Unit separator [S_7]. Control shift O.

Figures				Upper Case				Lower Case			
Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character[10]
1	032	040	SP[1]	1	064	100	@[4]	0	096	140	`[11]
0	033	041	!	0	065	101	A	1	097	141	a
0	034	042	"	0	066	102	B	1	098	142	b
1	035	043	#[2]	1	067	103	C	0	099	143	c
0	036	044	$	0	068	104	D	1	100	144	d
1	037	045	%	1	069	105	E	0	101	145	e
1	038	046	&	1	070	106	F	0	102	146	f
0	039	047	'[3]	0	071	107	G	1	103	147	g
0	040	050	(	0	072	110	H	1	104	150	h
1	041	051	)	1	073	111	I	0	105	151	i
1	042	052	*	1	074	112	J	0	106	152	j
0	043	053	+	0	075	113	K	1	107	153	k
1	044	054	,	1	076	114	L	0	108	154	l
0	045	055	-	0	077	115	M	1	109	155	m
0	046	056	.	0	078	116	N	1	110	156	n
1	047	057	/	1	079	117	O	0	111	157	o
0	048	060	0	0	080	120	P	1	112	160	p
1	049	061	1	1	081	121	Q	0	113	161	q
1	050	062	2	1	082	122	R	0	114	162	r
0	051	063	3	0	083	123	S	1	115	163	s
1	052	064	4	1	084	124	T	0	116	164	t
0	053	065	5	0	085	125	U	1	117	165	u
0	054	066	6	0	086	126	V	1	118	166	v
1	055	067	7	1	087	127	W	0	119	167	w
1	056	070	8	1	088	130	X	0	120	170	x
0	057	071	9	0	089	131	Y	1	121	171	y
0	058	072	:	0	090	132	Z	1	122	172	z
1	059	073	;	1	091	133	[[5]	0	123	173	{
0	060	074	<	0	092	134	\[6]	1	124	174	:[12]
1	061	075	=	1	093	135	][7]	0	125	175	}[13]
1	062	076	>	1	094	136	^[8]	0	126	176	~[14]
0	063	077	?	0	095	137	_[9]	1	127	177	DEL[15]

[1]Space.
[2]£ on some (non-DEC) units.
[3]Accent acute or apostrophe (') before 1965, but used until recently on DEC units.
[4] 1965–67, but never on DEC units.
[5]Shift K.
[6]~ 1965–67, but never on DEC units. Shift L.
[7]Shift M.
[8]Circumflex— ↑ before 1965, but used until recently on DEC units.

[9]Underscore— ← before 1965, but used until recently on DEC units.

[10]Codes 140–173 first defined in 1965. For a full ASCII character set, the monitor accepts codes 140–176 as lower case. For a character set that lacks lower case, the monitor translates input codes 140–174 into the corresponding upper-case codes (100–134) and translates both 175 and 176 into 033, escape. Early versions of the monitor used 175 as the escape code and translated both 176 and 033 to it.

[11]Accent grave—@ 1965–67, but never on DEC units.

[12]Control character ACK before 1965; — 1965–67, but never on DEC units. Vertical bar may or may not have gap depending on font design, but generally does not on DEC units.

[13]Unassigned control character (usually ALT MODE) before 1965. Code generated by ALT MODE key on most DEC units.

[14]Control character ESC before 1965; | 1965–67, but never on DEC units. Code generated by ESC key on some DEC units.

[15]Delete, rub out (not part of lowercase set).

APPENDIX

B

DECSYSTEM-20 Opcodes

ALGEBRAIC REPRESENTATION

Boolean	**In-out**
Byte manipulation	**Program control**
Fixed-point arithmetic	**Shift and rotate**
Floating-point arithmetic	**Stack**
Full-word data transmission	**Test, arithmetic**
Half-word data transmission	**Test, logical**

AC — The accumulator address in bits 9–12 of the instruction word (represented by *A* in the instruction descriptions).

AC + *N* — The address *N* greater than AC, except that accumulator addresses wrap around from 17, e.g., AC + 3 is 1 if AC is 16.

E — The result of the effective address calculation. When E is an address it has the number of bits appropriate to such use—depending on the type of processor, whether local or global, etc. E is 18 bits unsigned when used as a half-word operand, mask, or output conditions; 9 bits signed when used as a scale factor or shift number; and 18 bits signed when used as an offset. For any signed quantity, the sign is always bit 18.

E_R	The in-section part of E (the right 18 bits).
E_L	The section-number part of E (those bits, if any, at the left of bit 18).
$E+N$	The address N greater than E, with a wraparound, where appropriate, from a section value of 777777 without changing the section number.
PC	The 30-bit or 18-bit program counter; the symbol also represents the contents of PC when used as the source of an address.
PC+1	The address produced by adding 1 to the in-section part of PC with a wraparound from 777777 (the section number does not change).
(X)	The word contained in register X.
$(X)_L$	The left half of (X).
$(X)_R$	The right half of (X).
$(X)_S$	The word contained in X with its left and right halves swapped.
A_n	The value of bit n of the quantity A.
A,B	A 36-bit word with the 18-bit quantity A in its left half and the 18-bit quantity B in its right half (either A or B may be 0).
(X,Y)	The contents of registers X and Y concatenated into a double word operand.
$(X–Y)$	The contents of registers X to Y concatenated into a multiword operand.
$((X))$	The word contained in the register addressed by (X), i.e., addressed by the word in register X.
$A \rightarrow B$	The quantity A replaces the quantity B (A and B may be half words, full words or double words). For example, $(AC) + (E) \rightarrow (AC)$ means the word in accumulator AC plus the word in memory location E replaces the word in AC.
(AC)(E)	The word in AC and the word in E.
$\wedge \vee \sim$	The Boolean operators AND, inclusive OR, exclusive OR, and complement.
$+ - \times \div \|\|$	The arithmetic operators for addition, negation or subtraction, multiplication, division and absolute value (magnitude).

Square brackets are used occasionally for grouping, but when they enclose an arithmetic computation they represent the "largest integer contained in" the enclosed quantity. With respect to values of their terms, the equations for a given instruction are in chronological order.

Full-Word Data Transmission

EXCH	250	$(AC) \leftrightarrow (E)$			
MOVE	200	$(E) \rightarrow (AC)$	MOVS	204	$(E)_S \rightarrow (AC)$
MOVEI	201	$0,E \rightarrow (AC)$	MOVSI	205	$E,0 \rightarrow (AC)$
MOVEM	202	$(AC) \rightarrow (E)$	MOVSM	206	$(AC)_S \rightarrow (E)$
MOVES	203	*If* AC ≠ 0: $(E) \rightarrow (AC)$	MOVSS	207	$(E)_S \rightarrow (E)$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
MOVN	210	$-(E) \rightarrow (AC)$	MOVM	214	$\|(E)\| \rightarrow (AC)$
MOVNI	211	$-[0,E] \rightarrow (AC)$	MOVMI	215	$0,E \rightarrow (AC)$
MOVNM	212	$-(AC) \rightarrow (E)$	MOVMM	216	$\|(AC)\| \rightarrow (E)$

MOVNS	213	$-(E) \rightarrow (E)$ *If* AC $\neq$ 0: $(E) \rightarrow (AC)$	MOVMS	217	$\lvert(E)\rvert \rightarrow (E)$ *If* AC $\neq$ 0: $(E) \rightarrow (AC)$
XMOVEI	415	*If not local AC reference:* $E \rightarrow (AC)$ *If local AC reference:* $1,E \rightarrow (AC)$			
DMOVE	120	$(E,E+1) \rightarrow (AC,AC+1)$	DMOVEM	124	$(AC,AC+1) \rightarrow (E,E+1)$
DMOVN	121	$-(E,E+1) \rightarrow (AC,AC+1)$	DMOVNM	125	$-(AC,AC+1) \rightarrow (E,E+1)$
BLT	251	*Move* $E_R - (AC)_R + 1$ *words starting with* $((AC)_L) \rightarrow ((AC)_R)$			
XBLT	020	*Move* $\lvert(AC)\rvert$ *words (see page 2-8)* *If* $(AC) > 0$: *start with* $((AC+1)) \rightarrow ((AC+2))$ *and go up* *If* $(AC) < 0$: *start with* $((AC+1) - 1) \rightarrow ((AC+2) - 1)$ *and go down*			

Fixed-Point Arithmetic

ADD	270	$(AC) + (E) \rightarrow (AC)$	SUB	274	$(AC) - (E) \rightarrow (AC)$
ADDI	271	$(AC) + 0,E \rightarrow (AC)$	SUBI	275	$(AC) - 0,E \rightarrow (AC)$
ADDM	272	$(AC) + (E) \rightarrow (E)$	SUBM	276	$(AC) - (E) \rightarrow (E)$
ADDB	273	$(AC) + (E) \rightarrow (AC)(E)$	SUBB	277	$(AC) - (E) \rightarrow (AC)(E)$
IMUL	220	$(AC) \times (E) \rightarrow (AC)$*	MUL	224	$(AC) \times (E) \rightarrow (AC,AC+1)$
IMULI	221	$(AC) \times 0,E \rightarrow (AC)$*	MULI	225	$(AC) \times 0,E \rightarrow (AC,AC+1)$
IMULM	222	$(AC) \times (E) \rightarrow (E)$*	MULM	226	$(AC) \times (E) \rightarrow (E)$†
IMULB	223	$(AC) \times (E) \rightarrow (AC)(E)$*	MULB	227	$(AC) \times (E) \rightarrow (AC,AC+1)(E)$
IDIV	230	$(AC) \div (E) \rightarrow (AC)$ REMAINDER $\rightarrow (AC+1)$	DIV	234	$(AC,AC+1) \div (E) \rightarrow (AC)$ REMAINDER $\rightarrow (AC+1)$
IDIVI	231	$(AC) \div 0,E \rightarrow (AC)$ REMAINDER $\rightarrow (AC+1)$	DIVI	235	$(AC,AC+1) \div 0,E \rightarrow (AC)$ REMAINDER $\rightarrow (AC+1)$
IDIVM	232	$(AC) \div (E) \rightarrow (E)$	DIVM	236	$(AC,AC+1) \div (E) \rightarrow (E)$
IDIVB	233	$(AC) \div (E) \rightarrow (AC)(E)$ REMAINDER $\rightarrow (AC+1)$	DIVB	237	$(AC,AC+1) \div (E) \rightarrow (AC)(E)$ REMAINDER $\rightarrow (AC+1)$

*The high order word of the product is discarded.
†The low order word of the product is discarded.

DADD	114	$(AC,AC+1) + (E,E+1) \rightarrow (AC,AC+1)$
DSUB	115	$(AC,AC+1) - (E,E+1) \rightarrow (AC,AC+1)$
DMUL	116	$(AC,AC+1) \times (E,E+1) \rightarrow (AC-AC+3)$
DDIV	117	$(AC-AC+3) \div (E,E+1) \rightarrow (AC,AC+1)$ REMAINDER $\rightarrow (AC+2,AC+3)$

Floating-Point Arithmetic

FAD	140	$(AC) + (E) \rightarrow (AC)$	FADR	144	$(AC) + (E) \rightarrow (AC)$
FADL	141	$(AC) + (E) \rightarrow (AC,AC+1)$	FADRI	145	$(AC) + E,0 \rightarrow (AC)$
FADM	142	$(AC) + (E) \rightarrow (E)$	FADRM	146	$(AC) + (E) \rightarrow (E)$
FADB	143	$(AC) + (E) \rightarrow (AC)(E)$	FADRB	147	$(AC) + (E) \rightarrow (AC)(E)$

FSB	150	(AC) − (E) → (AC)	FSBR	154	(AC) − (E) → (AC)
FSBL	151	(AC) − (E) → (AC,AC + 1)	FSBRI	155	(AC) − E,0 → (AC)
FSBM	152	(AC) − (E) → (E)	FSBRM	156	(AC) − (E) → (E)
FSBB	153	(AC) − (E) → (AC)(E)	FSBRB	157	(AC) − (E) → (AC)(E)
FMP	160	(AC) × (E) → (AC)	FMPR	164	(AC) × (E) → (AC)
FMPL	161	(AC) × (E) → (AC,AC + 1)	FMPRI	165	(AC) × E,0 → (AC)
FMPM	162	(AC) × (E) → (E)	FMPRM	166	(AC) × (E) → (E)
FMPB	163	(AC) × (E) → (AC)(E)	FMPRB	167	(AC) × (E) → (AC)(E)
FDV	170	(AC) ÷ (E) → (AC)	FDVR	174	(AC) ÷ (E) → (AC)
FDVL	171	(AC) ÷ (E) → (AC) REMAINDER → (AC + 1)	FDVRI	175	(AC) ÷ E,0 → (AC)
FDVM	172	(AC) ÷ (E) → (E)	FDVRM	176	(AC) ÷ (E) → (E)
FDVB	173	(AC) ÷ (E) → (AC)(E)	FDVRB	177	(AC) ÷ (E) → (AC)(E)

		UFA	130	(AC) + (E) → (AC + 1) *without normalization*	
		DFN	131	− (AC,E) → (AC,E)	
		FSC	132	(AC) × 2^{E} → (AC)	
		FLTR	127	(E) *floated, rounded* → (AC)	
FIX	122	(E) *fixed* → (AC)		FIXR	126	(E) *fixed, rounded* → (AC)
		DFAD	110	(AC,AC + 1) + (E,E + 1) → (AC,AC + 1)	
		DFSB	111	(AC,AC + 1) − (E,E + 1) → (AC,AC + 1)	
		DFMP	112	(AC,AC + 1) × (E,E + 1) → (AC,AC + 1)	
		DFDV	113	(AC,AC + 1) ÷ (E,E + 1) → (AC,AC + 1)	

Boolean

SETZ	400	0 → (AC)	SETO	474	777777777777 → (AC)
SETZI	401	0 → (AC)	SETOI	475	777777777777 → (AC)
SETZM	402	0 → (E)	SETOM	476	777777777777 → (E)
SETZB	403	0 → (AC)(E)	SETOB	477	777777777777 → (AC)(E)
SETA	424	(AC) → (AC) [*no-op*]	SETCA	450	~ (AC) → (AC)
SETAI	425	(AC) → (AC) [*no-op*]	SETCAI	451	~ (AC) → (AC)
SETAM	426	(AC) → (E)	SETCAM	452	~ (AC) → (E)
SETAB	427	(AC) → (E)	SETCAB	453	~ (AC) → (AC)(E)
SETM	414	(E) → (AC)	SETCM	460	~ (E) → (AC)
SETMI	415	0,E → (AC)	SETCMI	461	~ [0,E] → (AC)
SETMM	416	(E) → (E) [*no-op*]	SETCMM	462	~ (E) → (E)
SETMB	417	(E) → (AC)(E)	SETCMB	463	~ (E) → (AC)(E)

AND	404	(AC) ∧ (E) → (AC)	ANDCA	410	~ (AC) ∧ (E) → (AC)
ANDI	405	(AC) ∧ 0,E → (AC)	ANDCAI	411	~ (AC) ∧ 0,E → (AC)
ANDM	406	(AC) ∧ (E) → (E)	ANDCAM	412	~ (AC) ∧ (E) → (E)
ANDB	407	(AC) ∧ (E) → (AC)(E)	ANDCAB	413	~ (AC) ∧ (E) → (AC) (E)
ANDCM	420	(AC) ∧ ~ (E) → (AC)	ANDCB	440	~ (AC) ∧ ~ (E) → (AC)
ANDCMI	421	(AC) ∧ ~ [0,E] → (AC)	ANDCBI	441	~ (AC) ∧ ~ [0,E] → (AC)
ANDCMM	422	(AC) ∧ ~ (E) → (E)	ANDCBM	442	~ (AC) ∧ ~ (E) → (E)
ANDCMB	423	(AC) ∧ ~ (E) → (AC)(E)	ANDCBB	443	~ (AC) ∧ ~ (E) → (AC)(E)
IOR	434	(AC) ∨ (E) → (AC)	ORCA	454	~ (AC) ∨ (E) → (AC)
IORI	435	(AC) ∨ 0,E → (AC)	ORCAI	455	~ (AC) ∨ 0,E → (AC)
IORM	436	(AC) ∨ (E) → (E)	ORCAM	456	~ (AC) ∨ (E) → (E)
IORB	437	(AC) ∨ (E) → (AC)(E)	ORCAB	457	~ (AC) ∨ (E) → (AC)(E)
ORCM	464	(AC) ∨ ~ (E) → (AC)	ORCB	470	~ (AC) ∨ ~ (E) → (AC)
ORCMI	465	(AC) ∨ ~ [0,E] → (AC)	ORCBI	471	~ (AC) ∨ ~ [0,E] → (AC)
ORCMM	466	(AC) ∨ ~ (E) → (E)	ORCBM	472	~ (AC) ∨ ~ (E) → (E)
ORCMB	467	(AC) ∨ ~ (E) → (AC)(E)	ORCBB	473	~ (AC) ∨ ~ (E) → (AC)(E)
XOR	430	(AC) ⊽ (E) → (AC)	EQV	444	~ [(AC) ⊽ (E)] → (AC)
XORI	431	(AC) ⊽ 0,E → (AC)	EQVI	445	~ [(AC) ⊽ 0,E] → (AC)
XORM	432	(AC) ⊽ (E) → (E)	EQVM	446	~ [(AC) ⊽ (E)] → (E)
XORB	433	(AC) ⊽ (E) → (AC)(E)	EQVB	447	~ [(AC) ⊽ (E)] → (AC)(E)

Shift and Rotate

ASH	240	(AC) × 2^E → (AC)	ASHC	244	(AC,AC + 1) × 2^E → (AC,AC + 1)
ROT	241	*Rotate* (AC) E *places*	ROTC	245	*Rotate* (AC,AC + 1) E *places*
LSH	242	*Shift* (AC) E *places*	LSHC	246	*Shift* (AC,AC + 1) E *places*

Arithmetic Testing

AOBJP	252	(AC) + 1,1 → (AC) *If* (AC) ≥ 0: E → (PC)			
AOBJN	253	(AC) + 1,1 → (AC) *If* (AC) < 0: E → (PC)			
CAI	300	*No-op*	CAM	310	*No-op*
CAIL	301	*If* (AC) < E: *skip*	CAML	311	*If* (AC) < (E): *skip*
CAIE	302	*If* (AC) = E: *skip*	CAME	312	*If* (AC) = (E): *skip*
CAILE	303	*If* (AC) ≤ E: *skip*	CAMLE	313	*If* (AC) ≤ (E): *skip*
CAIA	304	*Skip*	CAMA	314	*Skip*
CAIGE	305	*If* (AC) ≥ E: *skip*	CAMGE	315	*If* (AC) ≥ (E): *skip*

CAIN	306	*If* (AC) ≠ E: *skip*	CAMN	316	*If* (AC) ≠ (E): *skip*
CAIG	307	*If* (AC) > E: *skip*	CAMG	317	*If* (AC) > (E): *skip*
JUMP	320	*No-op*	SKIP	330	*If* AC ≠ 0: (E) → (AC)
JUMPL	321	*If* (AC) < 0: E → (PC)	SKIPL	331	*If* AC ≠ 0: (E) → (AC) *If* (E) < 0: *skip*
JUMPE	322	*If* (AC) = 0: E → (PC)	SKIPE	332	*If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*
JUMPLE	323	*If* (AC) ≤ 0: E → (PC)	SKIPLE	333	*If* AC ≠ 0: (E) → (AC) *If* (E) ≤ 0: *skip*
JUMPA	324	E → (PC)	SKIPA	334	*If* AC ≠ 0: (E) → (AC) *Skip*
JUMPGE	325	*If* (AC) ≥ 0: E → (PC)	SKIPGE	335	*If* AC ≠ 0: (E) → (AC) *If* (E) ≥ 0: *skip*
JUMPN	326	*If* (AC) ≠ 0: E → (PC)	SKIPN	336	*If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*
JUMPG	327	*If* (AC) > 0: E → (PC)	SKIPG	337	*If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*
AOJ	340	(AC) + 1 → (AC)	SOJ	360	(AC) − 1 → (AC)
AOJL	341	(AC) + 1 → (AC) *If* (AC) <0: E→(PC)	SOJL	361	(AC) − 1 → (AC) *If* (AC) < 0: E → (PC)
AOJE	342	(AC) + 1 → (AC) *If* (AC) = 0: E → (PC)	SOJE	362	(AC) − 1 → (AC) *If* (AC) = 0: → (PC)
AOJLE	343	(AC) + 1 → (AC) *If* (AC) ≤ 0: E → (PC)	SOJLE	363	(AC) − 1 → (AC) *If* (AC) ≤ 0: E → (PC)
AOJA	344	(AC) + 1 → (AC) E → (PC)	SOJA	364	(AC) − 1 → (AC) E → (PC)
AOJGE	345	(AC) + 1 → (AC) *If* (AC) ≥ 0: E → (PC)	SOJGE	365	(AC) − 1 → (AC) *If* (AC) ≥ 0: E → (PC)
AOJN	346	(AC) + 1 → (AC) *If* (AC) ≠ 0: E → (PC)	SOJN	366	(AC) − 1 → (AC) *If* (AC) ≠ 0: E → (PC)
AOJG	347	(AC) + 1 → (AC) *If* (AC) > 0: E → (PC)	SOJG	367	(AC) − 1 → (AC) *If* (AC) > 0: E → (PC)
AOS	350	(E) + 1 → (E) *If* (AC) ≠ 0: (E) → (AC)	SOS	370	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC)
AOSL	351	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) < 0: *skip*	SOSL	371	(E) − 1 → (E) *If* AC ≠0: (E) → (AC) *If* (E) < 0: *skip*
AOSE	352	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*	SOSE	372	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*

AOSLE	353	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ≤ 0: *skip*	SOSLE	373	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ≤ 0: *skip*
AOSA	354	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *Skip*	SOSA	374	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *Skip*
AOSGE	355	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ≥ 0: *skip*	SOSGE	375	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ≥ 0: *skip*
AOSN	356	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*	SOSN	376	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*
AOSG	357	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*	SOSG	377	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*

Logical Testing and Modification

TLN	601	*No-op*	TRN	600	*No-op*
TLNE	603	*If* $(AC)_L \wedge E = 0$: *skip*	TRNE	602	*If* $(AC)_R \wedge E = 0$: *skip*
TLNA	605	*Skip*	TRNA	604	*Skip*
TLNN	607	*If* $(AC)_L \wedge E \neq 0$: *skip*	TRNN	606	*If* $(AC)_R \wedge E \neq 0$: *skip*
TLZ	621	$(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZ	620	$(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZE	623	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZE	622	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZA	625	$(AC)_L \wedge \sim E \rightarrow (AC)_L$ *skip*	TRZA	624	$(AC)_R \wedge \sim E \rightarrow (AC)_R$ *skip*
TLZN	627	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZN	626	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLC	641	$(AC)_L \triangledown E \rightarrow (AC)_L$	TRC	640	$(AC)_R \triangledown E \rightarrow (AC)_R$
TLCE	643	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \triangledown E \rightarrow (AC)_L$	TRCE	642	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \triangledown E \rightarrow (AC)_R$
TLCA	645	$(AC)_L \triangledown E \rightarrow (AC)_L$ *skip*	TRCA	644	$(AC)_R \triangledown E \rightarrow (AC)_R$ *skip*
TLCN	647	*If* $(AC)_L \wedge E \neq 0$: skip $(AC)_L \triangledown E \rightarrow (AC)_L$	TRCN	646	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \triangledown E \rightarrow (AC)_R$
TLO	661	$(AC)_L \vee E \rightarrow (AC)_L$	TRO	660	$(AC)_R \vee E \rightarrow (AC)_R$
TLOE	663	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TROE	662	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TLOA	665	$(AC)_L \vee E \rightarrow (AC)_L$ *skip*	TROA	664	$(AC)_R \vee E \rightarrow (AC)_R$ *skip*
TLON	667	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TRON	666	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$

TDN	610	*No-op*	TSN	611	*No-op*
TDNE	612	*If* (AC) ∧ (E) = 0: *skip*	TSNE	613	*If* (AC) ∧ $(E)_S$ = 0: *skip*
TDNA	614	*Skip*	TSNA	615	*Skip*
TDNN	616	*If* (AC) ∧ (E) ≠ 0: *skip*	TSNN	617	*If* (AC) ∧ $(E)_S$ ≠ 0: *skip*
TDZ	630	(AC) ∧ ~ (E) → (AC)	TSZ	631	(AC) ∧ ~ $(E)_S$ → (AC)
TDZE	632	*If* (AC) ∧ (E) = 0: *skip* (AC) ∧ ~ (E) → (AC)	TSZE	633	*If* (AC) ∧ $(E)_S$ = 0: *skip* (AC) ∧ ~ $(E)_S$ → (AC)
TDZA	634	(AC) ∧ ~ (E) → (AC) *skip*	TSZA	635	(AC) ∧ ~ $(E)_S$ → (AC) *skip*
TDZN	636	*If* (AC) ∧ (E) ≠ 0: *skip* (AC) ∧ ~ (E) → (AC)	TSZN	637	*If* (AC) ∧ $(E)_S$ ≠ 0: *skip* (AC) ∧~ $(E)_S$ → (AC)
TDC	650	(AC) ▽ (E) → (AC)	TSC	651	(AC) ▽ $(E)_S$ → (AC)
TDCE	652	*If* (AC) ∧ (E) = 0: *skip* (AC) ▽ (E) → (AC)	TSCE	653	*If* (AC) ∧ $(E)_S$ = 0: *skip* (AC) ▽ $(E)_S$ → (AC)
TDCA	654	(AC) ▽ (E) → (AC) *skip*	TSCA	655	(AC) ▽ $(E)_S$ → (AC) *skip*
TDCN	656	*If* (AC) ∧ (E) ≠ 0: *skip* (AC) ▽ (E) → (AC)	TSCN	657	*If* (AC) ∧ $(E)_S$ ≠ 0: *skip* (AC) ▽ $(E)_S$ → (AC)
TDO	670	(AC) ∨ (E) → (AC)	TSO	671	(AC) ∨ $(E)_S$ → (AC)
TDOE	672	*If* (AC) ∧ (E) = 0: *skip* (AC) ∨ (E) → (AC)	TSOE	673	*If* (AC) ∧ $(E)_S$ = 0: *skip* (AC) ∨ $(E)_S$ → (AC)
TDOA	674	(AC) ∨ (E) → (AC) *skip*	TSOA	675	(AC) ∨ $(E)_S$ → (AC) *skip*
TDON	676	*If* (AC) ∧ (E) ≠0: *skip* (AC) ∨ (E) → (AC)	TSON	677	*If* (AC) ∧ $(E)_S$ ≠ 0: *skip* (AC) ∨ $(E)_S$ → (AC)

Half-Word Data Transmission

HLL	500	$(E)_L$ → $(AC)_L$	HLLZ	510	$(E)_L$,0 → (AC)
HLLI	501	0 → $(AC)_L$	HLLZI	511	0 → (AC)
HLLM	502	$(AC)_L$ → $(E)_L$	HLLZM	512	$(AC)_L$,0 → (E)
HLLS	503	*If* AC ≠ 0: (E) → (AC)	HLLZS	513	0 → $(E)_R$ *If* AC ≠ 0: (E) → (AC)
HLLO	520	$(E)_L$,777777 → (AC)	HLLE	530	$(E)_L$,[$(E)_0$ × 777777] → (AC)
HLLOI	521	0,777777 → (AC)	HLLEI	531	0 → (AC)
HLLOM	522	$(AC)_L$,777777 → (E)	HLLEM	532	$(AC)_L$,[$(AC)_0$ × 777777] → (E)
HLLOS	523	777777 → $(E)_R$ *If* AC ≠ 0: (E) → (AC)	HLLES	533	$(E)_0$ × 777777 → $(E)_R$ *If* AC ≠ 0: (E) → (AC)
HLR	544	$(E)_L$ → $(AC)_R$	HLRZ	554	0,$(E)_L$ → (AC)
HLRI	545	0 → $(AC)_R$	HLRZI	555	0 → (AC)
HLRM	546	$(AC)_L$ → $(E)_R$	HLRZM	556	0,$(AC)_L$ → (E)
HLRS	547	$(E)_L$ → $(E)_R$ *If* AC ≠ 0: (E) → (AC)	HLRZS	557	0,$(E)_L$ → (E) *If* AC ≠ 0: (E) → (AC)
HLRO	564	777777,$(E)_L$ → (AC)	HLRE	574	[$(E)_0$ × 777777],$(E)_L$ → (AC)
HLROI	565	777777,0 → (AC)	HLREI	575	0 → (AC)

HLROM	566	777777,$(AC)_L \rightarrow (E)$	HLREM	576	$[(AC)_0 \times 777777],(AC)_L \rightarrow (E)$
HLROS	567	777777,$(E)_L \rightarrow (E)$ *If* AC ≠ 0: $(E) \rightarrow (AC)$	HLRES	577	$[(E)_0 \times 777777],(E)_L \rightarrow (E)$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HRR	540	$(E)_R \rightarrow (AC)_R$	HRRZ	550	$0,(E)_R \rightarrow (AC)$
HRRI	541	$E \rightarrow (AC)_R$	HRRZI	551	$0,E \rightarrow (AC)$
HRRM	542	$(AC)_R \rightarrow (E)_R$	HRRZM	552	$0,(AC)_R \rightarrow (E)$
HRRS	543	*If* AC ≠ 0: $(E) \rightarrow (AC)$	HRRZS	553	$0 \rightarrow (E)_L$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HRRO	560	777777,$(E)_R \rightarrow (AC)$	HRRE	570	$[(E)_{18} \times 777777],(E)_R \rightarrow (AC)$
HRROI	561	777777,$E \rightarrow (AC)$	HRREI	571	$[E_{18} \times 777777],E \rightarrow (AC)$
HRROM	562	777777,$(AC)_R \rightarrow (E)$	HRREM	572	$[(AC)_{18} \times 777777],(AC)_R \rightarrow (E)$
HRROS	563	777777 $\rightarrow (E)_L$ *If* AC ≠ 0: $(E) \rightarrow (AC)$	HRRES	573	$(E)_{18} \times 777777 \rightarrow (E)_L$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HRL	504	$(E)_R \rightarrow (AC)_L$	HRLZ	514	$(E)_R,0 \rightarrow (AC)$
HRLI	505	$E \rightarrow (AC)_L$	HRLZI	515	$E,0 \rightarrow (AC)$
HRLM	506	$(AC)_R \rightarrow (E)_L$	HRLZM	516	$(AC)_R,0 \rightarrow (E)$
HRLS	507	$(E)_R \rightarrow (E)_L$ *If* AC ≠ 0: $(E) \rightarrow (AC)$	HRLZS	517	$(E)_R,0 \rightarrow (E)$ *If* (AC) ≠ 0: $(E) \rightarrow (AC)$
HRLO	524	$(E)_R$,777777 $\rightarrow$ (AC)	HRLE	534	$(E)_R,[(E)_{18} \times 777777] \rightarrow (AC)$
HRLOI	525	E,777777 $\rightarrow$ (AC)	HRLEI	535	$E,[E_{18} \times 777777] \rightarrow (AC)$
HRLOM	526	$(AC)_R$,777777 $\rightarrow$ (E)	HRLEM	536	$(AC)_R,[(AC)_{18} \times 777777] \rightarrow (E)$
HRLOS	527	$(E)_R$,777777 $\rightarrow$ (E) *If* AC ≠ 0: $(E) \rightarrow (AC)$	HRLES	537	$(E)_R\ [(E)_{18} \times 777777] \rightarrow (E)$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
XHLLI	501	$E_L \rightarrow (AC)_L$			

Program Control

XCT	256	*Execute* (E)
JFFO	243	*If* (AC) = 0: $0 \rightarrow (AC + 1)$ *If* (AC) ≠ 0: $E \rightarrow (PC)$
JFCL	255	*If* AC ∧ FLAGS ≠ 0. $E \rightarrow (PC)$ ~ AC ∧ FLAGS → FLAGS
JRST	25400	$E \rightarrow (PC)$
PORTAL	25404	$0 \rightarrow$ PUBLIC $E \rightarrow (PC)$
JRSTF	25410	$(X)_L$ *or* $(Y)_L \rightarrow$ FLAGS $E \rightarrow (PC)$
HALT	25420	$E \rightarrow (PC)$ *stop*
XJRSTF	25424	$(E)_L \rightarrow$ FLAGS $(E+1) \rightarrow (PC)$
XJEN	25430	*Dismiss* PI$(E)_L \rightarrow$ FLAGS $(E+1) \rightarrow (PC)$
XPCW	25434	FLAGS, $0 \rightarrow (E)$ $PC+1 \rightarrow (E+1)$ $(E+2)_L \rightarrow$ FLAGS $(E+3) \rightarrow (PC)$
JEN	25450	*Dismiss* PI $(X)_L$ *or* $(Y)_L \rightarrow$ FLAGS $E \rightarrow (PC)$
SFM	25460	FLAGS, $0 \rightarrow (E)$

JSR	264	*If* $PC_L = 0$: FLAGS, $PC_R + 1 \rightarrow (E)$ $E+1 \rightarrow (PC)$ *If* $PC_L \neq 0$: $PC+1 \rightarrow (E)$ $E+1 \rightarrow (PC)$
JSP	265	*If* $PC_L = 0$: FLAGS, $PC_R + 1 \rightarrow (AC)$ $E \rightarrow (PC)$ *If* $PC_L \neq 0$: $PC+1 \rightarrow (AC)$ $E \rightarrow (PC)$
JSA	266	$(AC) \rightarrow (E)$ E_R, $PC_R + 1 \rightarrow (AC)$ $E+1 \rightarrow (PC)$
JRA	267	$((AC)_L) \rightarrow (AC)$ $E \rightarrow (PC)$
MAP	257	PHYSICAL MAP DATA $\rightarrow$ (AC)

Stack

PUSH	261	*If* $PC_L = 0$ *or* $(AC)_{0,6\text{-}17} \leq 0$: $(AC) + 1,1 \rightarrow (AC)$ $(E) \rightarrow ((AC)_R)$ *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} > 0$: $(AC) + 1 \rightarrow (AC)$ $(E) \rightarrow ((AC))$
POP	262	*If* $PC_L = 0$ *or* $(AC)_{0,6\text{-}17} \leq 0$: $((AC)_R) \rightarrow (E)$ $(AC) - 1,1 \rightarrow (AC)$ *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} > 0$: $((AC)) \rightarrow (E)$ $(AC) - 1 \rightarrow (AC)$
PUSHJ	260	*If* $PC_L = 0$: $(AC) + 1,1 \rightarrow (AC)$ FLAGS, $PC + 1 \rightarrow ((AC)_R)$ *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} \leq 0$: $(AC) + 1,1 \rightarrow (AC)$ $PC+1 \rightarrow ((AC)_R)$ *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} > 0$: $(AC) + 1 \rightarrow (AC)$ $PC+1 \rightarrow ((AC))$ $E \rightarrow (PC)$
POPJ	263	*If* $PC_L = 0$: $((AC)_R)_R \rightarrow (PC)$ $(AC) - 1,1 \rightarrow (AC)$ *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} \leq 0$: $((AC)_R) \rightarrow (PC)$ $(AC) - 1,1 \rightarrow (AC)$ *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} > 0$: $((AC)) \rightarrow (PC)$ $(AC) - 1 \rightarrow (AC)$
ADJSP	105	*If* $PC_L = 0$ *or* $(AC)_{0,6\text{-}17} \leq 0$: $(AC) + [\pm] E_R$, $E_R \rightarrow (AC)$ *If* $PC_L \neq 0$ *and* (AC) $_{0,6\text{-}17} > 0$: $(AC) + [\pm] E_R \rightarrow (AC)$

Byte Manipulation

IBP	133	*Linear operations on pointer in* E *or* E,E+1
	AC = 0	*If* $P - S \geq 0$: $P - S \rightarrow P$ *If* $P - S < 0$: $Y + 1 \rightarrow Y$ $36 - S \rightarrow P$
ADJBP	133	*Array operations on pointer* in E *or* E, E + 1
	AC ≠ 0	*Let* A = REMAINDER $\dfrac{36 - P}{S}$

If $S > 36 - A$: $1 \rightarrow$ NO DIVIDE
If $S = 0$: $(E) \rightarrow (AC)$ *or* $(E,E+1) \rightarrow (AC,AC+1)$
If $0 < S < 36 - A$: *make copy* C *of* (E) *or* (E,E+1)

Compute $(AC) + \left[\dfrac{36 - P}{S}\right] = Q \times$ BYTES/WORD $+ R$

$1 \leq R \leq$ BYTES/WORD $= \left[\dfrac{36 - P}{S}\right] + \left[\dfrac{P}{S}\right]$

$Y\{C\} + Q \rightarrow Y\{C\}$

$36 - R \times S - A \rightarrow P\{C\}$

$C \rightarrow (AC)$ *or* $(AC,AC+1)$

LDB	135	BYTE IN ((E)) → (AC)
DPB	137	BYTE IN (AC) → BYTE IN ((E))
ILDB	134	IBP *and* LDB
IDPB	136	IBP *and* DPB

In-Out

CONO	70020	E → COMMAND	CONSZ	70030	*If* $\text{STATUS}_R \wedge E = 0$: *skip*
CONI	70024	STATUS → (E)	CONSO	70034	*If* $\text{STATUS}_R \wedge E \neq 0$: *skip*
DATAO	70014	(E) → DATA	DATAI	70004	DATA → (E)

BLKO	70010	(E) + 1,1 → (E)	$((E)_R)$ → DATA	*If* $(E)_L \neq 0$: *skip*
BLKI	70000	(E) + 1,1 → (E)	DATA → $((E)_R)$	*If* $(E)_L \neq 0$: *skip*

POWERS OF TWO

2^N	N	2^{-N}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 466 951 953 614 183 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125
18 446 744 073 709 551 616	64	0.000 000 000 000 000 000 054 210 108 624 275 221 700 372 640 043 497 085 571 289 062 5
36 893 488 147 419 103 232	65	0.000 000 000 000 000 000 027 105 054 312 137 610 850 186 320 021 748 542 785 644 531 25
73 786 976 294 838 206 464	66	0.000 000 000 000 000 000 013 552 527 156 068 805 425 093 160 010 874 271 392 822 265 625
147 573 952 589 676 412 928	67	0.000 000 000 000 000 000 006 776 263 578 034 402 712 546 580 005 437 135 696 411 132 312 5
295 147 905 179 352 825 856	68	0.000 000 000 000 000 000 003 388 131 789 017 201 356 273 290 002 718 567 848 205 566 406 25
590 295 810 358 705 651 712	69	0.000 000 000 000 000 000 001 694 065 894 508 600 678 136 645 001 359 283 924 102 783 203 125
1 180 591 620 717 411 303 424	70	0.000 000 000 000 000 000 000 847 032 947 254 300 339 068 322 500 679 641 962 051 391 601 562 5
2 361 183 241 434 822 606 848	71	0.000 000 000 000 000 000 000 423 516 473 627 150 169 534 161 250 339 820 981 025 695 800 781 25
4 722 366 482 869 645 213 696	72	0.000 000 000 000 000 000 000 211 758 236 813 575 084 767 080 625 169 910 490 512 847 900 390 625

APPENDIX

C

Debugging a Program

DEBUG is a program that accepts your program as input and allows you to examine, change, add, delete, and execute lines. It is a handy tool to use when a program is not working properly. DEBUG allows you to *single step* through a program. In other words, you can examine each line, as well as other locations, before executing it. You can also reexamine any location after executing a line to see what effect the new line has had. To acquaint you with DEBUG (or DDT), we will use the following program in the examples in this chapter.

```
        TITLE TESTDEBUG
        SEARCH MONSYM
NUMB:   1234
        4321
STRING: ASCIZ/ABC/
LETTER: "A"
        32,,14
START:  MOVE 1, [POINT 7, STRING]
        PSOUT%
        ADD 1, NUMB
        MOVEM 1,2
DONE:   HALTF%
        END START
```

C.1 SETTING UP DDT

You can initiate the debugging facilities at TOPS-20 with the command

```
@DEBUG Programname.MAC
```

The system will respond with

```
[LINK:    loading]
[LNKDEB DDT execution]
DDT
```

There is need no special prompt for DDT other than the cursor position indicator. The computer loads the DDT program into memory and places the program to be debugged in the memory locations following the DDT program. The starting location for your program is .JBDA (the leading dot is part of the address). DDT uses this symbol to identify some locations within your program. To examine a location, you must type the location value, either its numerical address or symbolic name, followed by a slash (/). The system will then tab once, display the contents of the location, tab once again, and then wait for your next instruction. For example,

```
NUMB/    1234      ,      ; / opens a location
```

NUMB is underlined here to indicate the part that you type; the computer types the rest. The arrow indicates the cursor's position. (Underlining and arrows will continue to have these meanings in the examples that follow.) You can now close the location with one of the following closing commands: CR, carriage return; LF, line feed; circumflex (^), TAB, or back slash (\). Typing only one of these commands will not change the contents of the location. In closing a location, CR will echo to your terminal, and DDT will then wait for your next instruction. Remember you will receive no prompt. LF will close the present location, open the next location, and display its contents. A circumflex will close the present location, open the previous location, and show its contents. For example,

```
NUMB/ 1234(LF)       ; LF  closes, then opens next location
.JBDA + 1/  4321  ^  ;   closes, then opens previous location
.JBDA/      1234  ↑
```

(A circled element indicates a key on the keyboard.) A TAB or back slash will open the location addressed by bits 18 to 35 of the present opened

location. This command allows you to see the effects of direct addressing but not indexed nor indirect addressing. The difference between the TAB and the back slash is that TAB moves the DDT program counter to the opened location, while the back slash leaves the DDT PC at the location of the first back slash. The dot symbol indicates the present value of the DDT PC.

The examples below demonstrate the use of TAB, back slash, and dot.

```
START+2/  ADD 1,  .JBDA (TAB)      ;TAB opens operand location
.JBDA/    1234    ./    1234       ; ./ opens present location
```

or

```
START+2/  ADD 1, .JBDA   Y\

.JBDA/    1234   Y\   ADD 1, .JBDA
```

The operand of the first instruction in DEBUG (the contents of START) is a literal. If you open START, the computer will respond as follows.

```
START/ MOVE 1, DONE^£+1
```

The last line in the program is DONE. Because the literal pool begins after the last line, DONE£+1 is the operand in the location START. DDT treats program labels as created symbols; thus, program labels have trailing £'s.

To change a value of a location, open the location and then type the new contents. You can then issue any of the closing commands. For example,

```
NUMB/    1234    456 (CR)
NUMB/    456
```

DDT does not change values permanently. They change only while the program is in DDT. If the change is to be permanent, you must use the editor.

If DDT does not understand your request—for example, you try to open a location not available to you or type an incorrect command symbol—it will respond with the letter U (unrecognized) and wait for your next request.

C.2 BREAKPOINTS AND EXECUTION COMMANDS

The real power of DDT is its ability to stop the execution of a program at almost any line so that you can examine and/or change the contents of locations, add or delete statements, and then continue with the program. A *breakpoint* (B) will stop the program for you. You can set it at a location

by typing the name of the location, pressing the escape key (it echoes as a dollar sign), and then typing B (no spaces). The system will respond with a TAB.

location(esc)B ;B sets a breakpoint

Typically you will set a breakpoint at the beginning of a program. The system will handle up to eight active breakpoints at a time.

START(esc)B DONE (esc) B ↑

DDT refers to the breakpoints as 1B, 2B,...8B and assigns these numbers in the order in which they were set. To see the *N*th breakpoint, you must press the escape key *N* B/ (no spaces).

(esc)1B/ START ;show first breakpoint

To remove the *N*th breakpoint, type in "0 escape key *N* B" (no spaces). The example below will remove the breakpoint at DONE.

0(esc)2B ;remove second breakpoint

You cannot set breakpoints in the middle of an effective address calculation, nor can you set a breakpoint at HERE.

```
        JRST @HERE
        ...
HERE:   THERE
THERE:  ...
```

Once the breakpoints are set, you can run the program by typing in the location of the line to be executed, then the escape key, and finally the letter G (no spaces).

location (esc) G ;resume execution at location

The system will execute the line at location and the following lines until it finds a breakpoint. It will not execute the line at the breakpoint, and if you do not specify a location, such as esc G, it will begin at the first line of the program. Once at a breakpoint, you can resume execution with either an escape X or an escape P. Escape X will simply execute the next line via single stepping; if you want to execute more than one line, use

N (esc) X ;execute N lines

(no spaces) where *N* is the number of octal lines. If the system encounters

a breakpoint within the *N* lines, it will not stop. It will only stop at the end of the *N* lines. To proceed to the next breakpoint, type escape P.

```
(esc) P          ;proceed until next breakpoint
```

C.3 CHANGING OUTPUT FORMAT

The default output mode is the symbolic mode, which translates each machine word as "opcode ac, operand." To see the machine code value of an opened location, type an equal sign.

```
START+2/    ADD 1, .JBDA   =270040,,140
```

To go from machine code to symbols, use the underscore key (_); on some terminals, this key may be a back arrow.

```
START+2/    ADD 1, .JBDA   =270040,,140_   ADD 1, .JBDA
```

Of course, once the value is opened, you can change it by typing either symbolic or machine code.

```
START+2/    ADD 1, .JBDA    ADDI 1, NUMB (CR)
./  ADDI 1, NUMB    270040,,140   -ADD 1, .JBDA
```

To display the right half of a location in numerical form, type escape A (address).

```
START+2/  ADD 1, .JBDA    (esc) A./    ADD 1, 140 (LF)
150/  MOVEM 1,2
```

Note that escape A prints locations in their numerical form (150/). To change back to the usual form, type escape R. Escape A and escape R are subforms of escape S. The normal mode is "escape S escape R." The letters C, F, T, H, or R preceded by the escape key will enable you to use other output modes. (If you are only changing a mode, you will want to type two escape keys followed by a letter, as in (esc) (esc) C). The C mode (code) will select machine code as the output form.

```
START+2/  ADD 1, .JBDA    (esc) C./  270040,,140   (LF)
START+3/  202040,,2_   MOVEM 1,2   (esc) S  (esc)  R
```

If you open location STRING in the normal mode, the output will be

```
STRING/    ANDM 1, 300000(10)
```

by typing the name of the location, pressing the escape key (it echoes as a dollar sign), and then typing B (no spaces). The system will respond with a TAB.

location(esc)B ;B sets a breakpoint

Typically you will set a breakpoint at the beginning of a program. The system will handle up to eight active breakpoints at a time.

START(esc)B DONE (esc) B ↑

DDT refers to the breakpoints as 1B, 2B,...8B and assigns these numbers in the order in which they were set. To see the *N*th breakpoint, you must press the escape key *N* B/ (no spaces).

(esc)1B/ START ;show first breakpoint

To remove the *N*th breakpoint, type in "0 escape key *N* B" (no spaces). The example below will remove the breakpoint at DONE.

0(esc)2B ;remove second breakpoint

You cannot set breakpoints in the middle of an effective address calculation, nor can you set a breakpoint at HERE.

```
        JRST @HERE
        ...
HERE:   THERE
THERE:  ...
```

Once the breakpoints are set, you can run the program by typing in the location of the line to be executed, then the escape key, and finally the letter G (no spaces).

location (esc) G ;resume execution at location

The system will execute the line at location and the following lines until it finds a breakpoint. It will not execute the line at the breakpoint, and if you do not specify a location, such as esc G, it will begin at the first line of the program. Once at a breakpoint, you can resume execution with either an escape X or an escape P. Escape X will simply execute the next line via single stepping; if you want to execute more than one line, use

N (esc) X ;execute N lines

(no spaces) where *N* is the number of octal lines. If the system encounters

a breakpoint within the *N* lines, it will not stop. It will only stop at the end of the *N* lines. To proceed to the next breakpoint, type escape P.

```
(esc) P          ;proceed until next breakpoint
```

C.3 CHANGING OUTPUT FORMAT

The default output mode is the symbolic mode, which translates each machine word as "opcode ac, operand." To see the machine code value of an opened location, type an equal sign.

```
START+2/    ADD 1, .JBDA   =270040,,140
```

To go from machine code to symbols, use the underscore key (_); on some terminals, this key may be a back arrow.

```
START+2/    ADD 1, .JBDA   =270040,,140_   ADD 1, .JBDA
```

Of course, once the value is opened, you can change it by typing either symbolic or machine code.

```
START+2/    ADD 1, .JBDA    ADDI 1, NUMB  (CR)
./  ADDI 1, NUMB    270040,,140   -ADD 1, .JBDA
```

To display the right half of a location in numerical form, type escape A (address).

```
START+2/  ADD 1, .JBDA    (esc) A./    ADD 1, 140 (LF)
150/  MOVEM 1,2
```

Note that escape A prints locations in their numerical form (150/). To change back to the usual form, type escape R. Escape A and escape R are subforms of escape S. The normal mode is "escape S escape R." The letters C, F, T, H, or R preceded by the escape key will enable you to use other output modes. (If you are only changing a mode, you will want to type two escape keys followed by a letter, as in (esc) (esc) C). The C mode (code) will select machine code as the output form.

```
START+2/  ADD 1, .JBDA    (esc) C./  270040,,140   (LF)
START+3/  202040,,2_   MOVEM 1,2   (esc) S (esc) R
```

If you open location STRING in the normal mode, the output will be

```
STRING/    ANDM 1, 300000(10)
```

because the location STRING contains ASCII and not opcode ac, operand. Use the T (text) mode to read ASCII. Reopening STRING in the T mode will yield the following.

(esc) T./ ABC START+2/ .^B^@^@0

The T mode does not apply for the normal form. To see the numeric code of a location in another base, type escape *N* O, where *N* is a decimal number from 2 to 36. This command is handy for unpacking locations.

(esc) T STRING/ ABC (esc) 80./ 101,102,103,0,0,0

If a location contains a floating point value, you should type the location contents in the F mode. To display the contents of a location in half-word format, use the H mode.

C.4 INPUT MODES

To input new values, simply open a location and type in the new contents. The only acceptable input radix is octal (except for the single digits 8 and 9). Fix point and exponent notation is acceptable. To replace a packed ASCII string, use a double quote and delimit the replacing string.

STRING/ ABC "/CDE/ (CR)
./ CDE

Double commas will supply half words.

LETTER+1/32,,145,, 1 (CR)
./ 5,,1

You can add symbols to the symbol table by using

value<symbol: ;adding a symbol

as in this example:

```
3<AC:
```

which places ac in the symbol table and assigns it the value 3. To place a label at a location, open the location and then type

label:

For example,

START+2/ ADD 1, NUMB SHOW: (CR)
SHOW/ ADD 1, NUMB

You can add instructions to the end of a program by opening the last line (PAT../) and then typing in the desired line. To remove symbols from the symbol table, use

symbol (esc) K ;removing a symbol

APPENDIX

D

Fortran and Macro

This appendix discusses how you can move back and forth between FORTRAN and MACRO subroutines. Often this method is easier than rewriting a long special routine that you will rarely need. For example, if you need to take the square root of a number while in a MACRO program, you can leave MACRO, use FORTRAN's square root operation, and return to MACRO with the answer. Alternatively, if you need a fancy I/O routine while in a FORTRAN program, you can temporarily use MACRO. For instance, if you do not want the FORTRAN READ to stop a program until it is satisfied, you can leave FORTRAN, use MACRO's RDTTY% with RD%RIE to check the input buffer, and then return to FORTRAN.

D.1 MACRO CALLING FORTRAN

To communicate with FORTRAN (or any other language), you must understand how FORTRAN does calls and returns and passes parameters to subroutines. PUSHJ/POPJ, which use accumulator 17 (octal), will call/return a FORTRAN subroutine. If the call originates in a FORTRAN program, FORTRAN will set up the necessary stack. If the call originates in MACRO, you

must set up the stack. Accumulator 16 (octal) handles parameters. It contains the address of a table containing the list of addresses storing the values of the parameters. A great deal of indirect addressing takes place. Consider the FORTRAN fragment.

```
A=1
B=2
C=3
...
CALL SUB (A,B,C)
```

FORTRAN handles the following part of the program.

```
      -3,,0                ;number of parameters
HERE: A                    ;table of parameter
      B                    ;locations
      C
        ...
A: 1                       ;value of parameters
B: 2
C: 3
          ...
          MOVEI 16, HERE   ;table address
          PUSHJ 17, SUB    ;jump to subroutine
          ...
```

Once in the subroutine, the system references the parameter values by using operands

```
@(16)     ;the value of A
@1(16)    ;B
@2(16)    ;C
```

Below is a FORTRAN program that uses a subroutine to add two numbers.

```
        PROGRAM FOR.USED.BY.MAC
        IMPLICIT INTEGER (A-Z)
        WRITE(5,10)
10      FORMAT(1X,'IN FORTRAN')
        ! anything
        STOP
        END
        SUBROUTINE ADDI(A,B,C)
        IMPLICIT INTEGER(A-Z)
        C=A+B
        RETURN
        END
```

If you want a MACRO program to use the subroutine ADDI, you must set up the necessary macro address table and stack.

```
                                        ;this macro calls ADDI
        TITLE MAC.CALLING.FOR           ;in FOR.USED.BY.MAC
        SEARCH MONSYM,MYSTUF
        EXTERNAL ADDI
        -3,,0
ARG:    A                               ;address table
        B
        C
        DEPTH=15
STACK:  BLOCK DEPTH
START:  RESET
        MOVE 17,[IOWD DEPTH,STACK]
        MOVEI 16,ARG
        PUSHJ 17,ADDI                   ;call ADDI
        PRINT IN MAC FROM FORTRAN
        CRLF
        OUTPUT RESULT,<A,B,C>
        HALTF%
A:      3
B:      4
C:      Z
        END START
```

The linker must be past all files and since both files are programs .REQUIRE can not be used, rather,

```
EX FORTRANPROGRAM, MACROPROGRAM
```

The order in which the files appear after EX is the same as that established in Chapter 8.

D.2 FORTRAN CALLING MACRO

The following example is a FORTRAN program calling a macro subroutine, IADD. (We could have used a third parameter, C, as well.)

```
        PROGRAM FOR.CALL.MAC
        IMPLICIT INTEGER (A-Z)
        WRITE (5,10)
10      FORMAT(1X,'STARTING IN FORTRAN')
        A=2                         ;set up parameter
        B=3                         ;values
        CALL IADD(A,B)              ;call subroutine
        WRITE(5,*)A,B
        STOP
        END
```

Since FORTRAN is doing the calling, FORTRAN must set up the stack; it also sets up the parameters via accumulator 16. MACRO must then use accumulator 16 to set the parameters.

```
        TITLE MAC.USED.BY.FOR
        SEARCH MONSYM
        ENTRY IADD
IADD:   MOVE 1,@(16)            ;get first parameter
        ADD 1,@1(16)            ;add to second parameter
        MOVEM 1,@(16)           ;put result into
                                ;first parameter
        POPJ 17,                ;return
        HALTF%
        END
```

Since the macro file is a library file rather than a program, the order in which the files pass to the linker is not important. Either of the following approaches will work.

```
EX FORTRANPROGRAM,MACROFILE
```

or

```
EX MACROFILE, FORTRANPROGRAM
```

APPENDIX

E

Pascal and Macro

In PASCAL, PUSHJ/POPJ does a procedure call/return using accumulator 17 (octal). If the call originates in a PASCAL program, PASCAL will set up the necessary stack. If the call originates in MACRO, you must set up the stack. Accumulators 2 through 5 handle parameters (at least in the Hedrick version of PASCAL). What is placed in these accumulators is determined as follows:

by VALUE, one word—the value in one accumulator
by VALUE, two words—the value in two successive accumulators
by VALUE, more than two words—address of object in one accumulator
by REFERENCE (VAR)—address of object in one accumulator.

An externally called macro routine may use all the accumulators, but the use of accumulators 15, 16 and 17 (octal) has restrictions. You must preserve the contents of these accumulators, for example. The restriction on 15 is a holdover for TOPS-10. Accumulator 16 is a pointer (frame pointer) to the local variables of the calling routine. The exact offset value from accumulator 16 varies; and if you need it, you can examine the object code

(use the PASCAL call at TOPS-20, similar to MACRO, with the switch/object). As you know, accumulator 17 is the stack, which you can use as long as you accompany each PUSHJ with a POPJ.

E.1 MACRO CALLING PASCAL

Below is a PASCAL procedure used by a macro program. This procedure is in a file called PASCALLIB.PAS. The procedure will add the two value parameters FIRSTNUM and SECONDNUM and place their sum in the variable parameter ANSWER. You must use the compiler switch (*$m-*) to signify that no main program exists.

```
(*$m-$*)                (* no main *)
procedure pasadd(var answer: integer;
                        firstnum,secondnum: integer);
begin
answer:=firstnum+secondnum;
end.
```

If you wish a macro program to use the procedure "pasadd," you must set up the stack and place the necessary parameter information in the accumulators. "Pasadd" has three parameters, a variable parameter and two value parameters. Since the first parameter (ANSWER) is a variable parameter, accumulator 2 should contain its address. In this example, however, because the first parameter is not passed to the procedure (it only comes from the procedure), you need not set up accumulator 2. Because the second (FIRSTNUM) and third (SECONDNUM) parameters are value parameters, accumulators 3 and 4 must contain the numbers to be added (not their addresses). A macro program that uses PASADD appears below. (For illustration only, we move FIRSTNUM (A = 20) immediately into ac 3, while directly addressing SECONDNUM (B = 30) into ac 4.)

```
        TITLE MAC.CALLING.PASCAL
        SEARCH MONSYM
        EXTERN PASADD
        DEPTH=10
STACK:  BLOCK DEPTH
B:      ^D30
START:  RESET%
        MOVE 17, [IOWD DEPTH, STACK]
        MOVEI 3, ^D20
        MOVE 4, B
        PUSHJ 17, PASADD
                                              ; ANS IN AC 2
        MOVEI 1, .PRIOU
        MOVEI 3, 12
        NOUT%
         Z
        HALTF%
        END START
```

(After the call, the answer, not its address, is in ac 2.) To execute this program, the linker must be past both files.

```
EX pascallib, macprogram
```

Because the PASCAL procedure is not in a program in this example (that is, we had no main program), we could have used a .REQUIRE PASCALLIB. Once the system has compiled the PASCALLIB and included the .REQUIRE statement, the following line will execute the program.

```
EX MACPROGRAM
```

E.2 PASCAL CALLING MACRO

The following PASCAL program calls a macro subroutine.

```
program pascal.call.mac
var
  a,b,c:integer;
procedure macadd(var answer: integer;
                 firstnum,
                 secondnum: integer);
          extern;

begin
  a:=20;
  b:=30;
  macadd(c,a,b);
end.
```

Since PASCAL is doing the calling, it sets up the stack. PASCAL will also set up the accumulators for passing the parameters. Accumulators 3 and 4 will have the value, and accumulator 2 will have the address for the answer.

```
        TITLE MAC.USED.BY.PASCAL
        ENTRY MACADD
MACADD: ADD 3, 4
        MOVEM 3, @2
        POPJ 17,
        END
```

You can execute the PASCAL program using either of the following lines.

```
EX PASCALPROGRAM, MACROLIB
```

or

```
EX MACROLIB, PASCALPROGRAM
```

Once you know the PASCAL-MACRO connection, you can find the memory address storing a PASCAL variable or, conversely, place a PASCAL variable in a specified memory location.

APPENDIX

F

Real Numbers

F.1 REPRESENTATIONS OF REAL NUMBERS

You are undoubtedly familiar with real numbers. For our purposes here, let's accept the following definition: A real number is a number that contains a decimal point. You can represent a decimal number in two ways—using a fixed point or a floating point. Consider a computer word that can represent a six-place, base-10 number (plus a sign bit). You can permanently fix the decimal point between the third and fourth decimal digits, for instance. Thus, you make 0.001 the smallest interval between two numbers, and you reduce the size of the largest number the computer word can represent. In other words, by fixing the decimal point, you reduce the range of numbers represented by the computer word (from −999999/999999 to −999.999/999.999) but increase the resolution error (the smallest difference between two numbers) from 1 to 0.001. The resolution is still an absolute (fixed) error.

Knowing that the decimal point is always at a specific spot helps in the bookkeeping of decimal arithmetic, but the reduced range can cause problems. For instance, if you wish to multiply, the six-decimal digit word we

are using as an example will run into overflow problems when multiplying numbers larger than 33.622.

Floating-point representation is very similar to scientific notation in that the decimal point shifts (floats) to the left or right so that numbers have a consistent form (decimal numbers always look like they are in the range 0 to 1). We need to use some of the digits in the computer word to keep track of the decimal position. If we set aside two decimal digits, the range is over 100 powers of 10. Again, however, we increase the range represented by a word at the expense of resolution. If we use two digits in a six-decimal digit word for the decimal position, we have only four places left to represent the digits in the number (these digits make up the *mantissa*). In the process, we must drop lesser significant digits, leading to a relative error. Thus, the difference between sequential numbers depends on the floating position of the decimal point. (If the power of 10 were 6 or 7, and the mantissa had four digits, the difference between neighboring numbers would then be 100, or 1000). With the extended range allowed by a floating decimal point, overflow does not arise as frequently, but setting up the representation (doing simple addition) becomes a challenge. (*Note:* Those readers interested primarily in how the DECSYSTEM-20 handles real numbers can skip the next two sections in this appendix.)

F.2 FIXED POINT

Though the value may change, the digits in the answers of mathematical operations do not change if a decimal point is present.

123.45	12345	12.345
+678.90	+67890	+67.890
802.35	80235	80.235

Therefore, all the arithmetic operations we have discussed so far (though they have used only integers) can use fixed-point operations. Because DECSYSTEM-20 MACRO does not have directives and monitor calls for fixed-point representation, you must set up the representations, choose the location of the decimal point, and set up I/O calls. The DECSYSTEM-20 word is large enough to store both the whole part and the fractional part of a real number in a single word. For instance, the whole part of a real number will fit in the left half of a word, with the fractional part in the right half of the same word. For our purposes here, we will use one word for the whole part and another word for the fractional part. (Double-word rep-

resentation allows us to handle a very large number—that is, a number that will not fit into one word.) We can then present a constant, such as 123.45, in this way.

```
wh1:    123
fr1:    45
```

But this method cannot handle 123.045 because it cannot handle the leading 0. It requires us to express all fractions in the same number of digits, shifting the problem from leading zeros to trailing zeros. (This problem is not unique to fixed-point representation, nor to double-word representation; it is a consequence of working with real numbers.) If we want the precision of three decimal places, we can code 123.45 as

```
wh1:    123
fr1:    450
```

And we can code 123.045 as

```
wh2:    123
fr2:    045
```

(The leading zero is not necessary.) To add two fixed-point numbers, using two-word representation, you must check for a carry from the fractional part to the whole part. (This step is not necessary with one-word representation.) The number of digits in the fractional part will determine the presence of a carry. For example (assuming three fractional decimal digits),

```
        ONE=1000        ;three-digit decimal
        ...
        MOVE 1, WH1
        ADD 1, WH2
        MOVE 2, FR1
        ADD 2, FR2
        CAIL 2, ONE     ;carry?
         JRST OK        ;no carry
        SUBI 2, ONE     ;yes, adjust frac
        ADDI 1, 1       ;add the carry
OK:     ...
```

To multiply two fixed-point numbers, you must use the distribution rules of multiplication.

$$(wh1 + fr1)*(wh2 + fr2) = wh3 + fr3$$

where

$$wh3 = wh1*wh2 + \text{quotient of } (wh1*fr2 + wh2*fr1)/one$$
$$fr3 = \text{remainder of } (wh1*fr2 + wh2*fr1)/one + \text{quotient of } (fr1*fr2)/one$$

The next problem is to accept a fixed-point number from the terminal. Typing a nondigit will terminate a NIN%. Therefore, you can use a NIN%, terminated by a decimal point, to get the whole part of a fixed-point number. A NIN% will not obtain the fractional part because it ignores leading zeros (the 0.45, 0.045 problem). You must use PBIN% plus a counter to count the number of digits to obtain the fractional part. The system will subtract the number of digits typed from the number of accuracy digits, which will yield the number of zeros to be added to the end of the fractional part of the real number. For example, with three accuracy digits (as in the above example) and a fractional part of 45 with two digits counted, the system will store 450. If the fractional part were 45 with three digits counted (045), it would store 45.

```
                                ;accepting a fixed-point number
        ACCNUM=3
                                ;whole part
        MOVEI 1, .PRIIN
        MOVEI 3, 12
        NIN%
         Z
        MOVEM 2, WH             ;save whole part
        SETZB 2, 3
                                ;fractional part
IN:     PBIN%
        CAIN 1, 15              ;carriage return
         JRST LF                ; yes
        SUBI 1, 60              ;sub ascii offset
        IMULI 2, 12             ;shift left
        ADD 2, 1                ;add new digit
        AOS 3                   ;count digit
        JRST IN
LF:     PBIN%                   ;get line feed
                                ;shift fractional part left
        SUBI 3, ACCNUM          ;places to shift left
SHF:    AOSLE  3
         JRST DONE
        IMULI 2, 12             ;add trailing zeros
        JRST SHF
DONE:   MOVEM 2, FR             ;save fractional part
```

Note that NIN% requires a digit before the decimal point (0.1 rather than .1). Note also that our example does not properly handle negative numbers. NIN% handles the whole part; extra steps are necessary for the fractional part. (If "wh" is negative, you must use a MOVN 2, 2 after constructing the fractional part. Finally, we have no provision to trap any fractions with more digits than "accnum."

F.3 FLOATING POINT

The DECSYSTEM-20 has opcodes, assembly directives, and monitor calls that handle floating-point numbers. Remember that you can convert (normalize) all decimal numbers to the range 0 to 1 by multiplying them by an appropriate power of 10.

12.3	0.123 × 10**2	123, 2
0.045	0.45 × 10**−1	45, −1

A mantissa (the 123 or the 45) and an exponent (the 2 or the -1) now represent each number. The system can store the mantissa and exponent in specific locations within a single word. Since the mantissa requires greater resolution, its field width (number of digits) is generally larger than the field width of the exponent. Traditionally, the exponent field is to the left of the mantissa field—that is, the computer word stores (2, 123) rather than (123, 2).

You must handle two signs—the overall sign of the number and the sign of the exponent. You can treat the overall sign as usual, giving it the status of bit 0. The left-most bit in the exponent field accommodates the sign of the exponent but it is set for positive numbers and cleared for negative numbers. For the DECSYSTEM-20, bit 0 is the sign bit and the exponent is in bits 1 to 8, with bit 1 the sign bit for the exponent. The mantissa is in the remaining 27 bits. Again, for positive exponents, bit 1 is set. The binary pattern 1000000 (octally, 200) represents a 0 exponent. The largest positive exponent is represented by 377. Removing the 200 offset yields an octal 177, which is 127 in decimal. The most negative exponent is represented by 000, which is 200 octal below the offset, or −128 decimal. The DECSYSTEM-20 allows the exponent a decimal range of −128 to +127, with 27 bits for the mantissa. The resolution is 1 bit in 27, or 1 out of 2**27, or 1/2**27, which is 0.000000007.

Below is an example of a program that accepts decimal digits from the terminal and stores them in DECSYSTEM-20 format. The program arbitrarily designates the smallest resolution, PRECIS, 2**−4 = 0.0625, and

sets PRENUM equal to 4. NIN%, terminated by the decimal point, obtains the whole part, and the system stores it in ANS. The number of bits in the whole part will be in location WHLPOW. PBIN% obtains the fractional part, which the system will store in FRAC; it will store the number of bits in the fractional part in FRCPOW. As with fixed-point representation, you must add trailing zeros to the fractional number and then divide it by PRECIS to convert it to a binary number.

```
            TITLE REALS
            SEARCH MONSYM
            PRECIS=^D0625
            PRENUM=4
            BASEIN=12
FRAC:       Z
WHLPOW:     Z
FRCPOW:     Z
            ANS=5
START:      RESET%
            SETZM WHLPOW
            SETZM FRCPOW
                                    ;get whole part
            MOVEI 1, .PRIIN
            MOVEI 3, 12
            NIN%
             Z
            MOVEM 2, ANS
            JUMPE 2, GETFRC
                                    ;get whlpow
POW2:       LSH 2, -1
            AOS WHLPOW
            JUMPN 2, POW2

                                    ;get fraction
            SETZ 2,
GETFRC:     PBIN%
            CAIN 1, 15              ;cr
            JRST LF
            SUBI 1, 60;offset
            IMULI 2, BASEIN
            ADD 2, 1
            AOS FRCPOW
            JRST GETFRC             ;increment FRCPOW
LF:         PBIN%                   ;get lf

                                    ;convert frac to binary
                                    ;calculate number of
                                    ;trailing zeros needed
```

```
                JUMPE 2, WHLSHF
                MOVE 1, FRCPOW
                SUBI 1, PRENUM
                                        ;add trailing zeros
LOOP:           IMULI 2, BASEIN
                AOSE 1
                 JRST LOOP
                                        ;determine number of times
                                        ;'precis' is in number
DIV:            IDIVI 2, PRECIS
                CALL 3, PRECIS/2        ;rounding
                 ADDI 2, 1
                MOVEM 2, FRAC
                                        ;ac 2 now has binary
                                        ;fraction
```

Now the whole part is in "ans," and the fractional part is in "frac." We must combine them into one word. The fractional part occupies PRENUM bits. To form one word, we shift ANS to the right by PRENUM and IOR this with FRAC. Now the resultant number must shift left so that the left-most set bit is in bit position 9. If we have a nonzero whole part, the number must shift 27 places minus PRENUM minus the number of bits in WHLPOW. We then find the exponent part using WHLPOW. If the whole part is 0, we must place the leading set bit of the fractional part in bit 9 and calculate the number of leading zeros. To find the leading set bit, shift the fractional number to the left "^D36−PRENUM" positions. If the fractional part contains a leading 1, this bit will be in bit position 0, signifying a negative number. If the leading fractional bit is 0, then bit 0 will not be set, signifying a positive number. If the leading fractional bit is 0, the number must shift left and each shift must be counted until the sign bit is set. The number of shifts is the negative exponent power for the floating-point base 2 representation.

```
        JUMPE ANS, LFSHF        ;if whole=0 then shift left

WHLSHF: LSH ANS, PRENUM         ;else normalized whole part
        IOR ANS, 2
        MOVEI 3, ^D27-PRENUM
        SUB 3, WHLPOW
        LSH ANS, @3
```

```
                                        ;fix exponent part
        MOVE 2, WHLPOW                  ;if whole=0 then normalize
        JRST EXP                        ;fractional part
LFSHF:  SETZ2,
        MOVE 1, FRAC
        LSH 1, ^D36-PRENUM
HERE:   JUMPL 1, DONE                   ;counting leading zeros
        SOS 2
        LSH 1, 1
        JRST HERE
DONE:   MOVEI 3, ^D27-PRENUM
        SUB 3, 2
        MOVE ANS, FRAC
        LSH ANS, @3
```

Now we must adjust the exponent part of the floating-point number. Accumulator 2 contains the number of bits either in the whole part or in the fractional part. The system adds this value to the offset, 200, and places the result in bits 1 to 8.

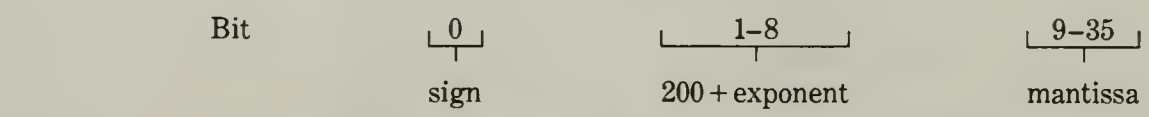
```
EXP:    MOVEI 1, 200        ;offset value
        ADD 1, 2
        LSH 1, ^D27
        IOR ANS, 1
```

Now ANS contains the floating-point representation of the number.

F.4 DECSYSTEM-20 REAL NUMBERS

The DECSYSTEM-20 stores a decimal floating point number in a word using normalized binary representation. The system places the binary exponent in bits 1 to 8. This number is offset by octal 200, meaning that positive exponents are in the range 200 to 377 (decimal 0 to 127). The negative exponents are represented by 177 to 0 (decimal −1 to −128). Bit 0 is still the overall sign bit. Bits 9 to 35 contain the normalized binary fraction (mantissa).

Floating-Point Word



A decimal point acts as an assembly directive, requesting that the assembler convert a decimal number to the normalized binary format. Two

monitor calls govern I/O of floating-point numbers, FLIN% and FLOUT%. To use a FLIN%, place in accumulator 1 the source device. A FLIN% returns +1, +2, with the normalized binary number in ac 2. A FLOUT% uses three accumulators: Accumulator 1 contains the destination, accumulator 2 contains a normalized floating-point number, and accumulator 3 contains a format word. You can use the simplest format by setting ac 3 to 0. The floating-point opcodes FADR, FSBR, FMPR, and FDVR add, subtract, multiply, and divide, and round the result. All these opcodes are one-word opcodes.

```
        TITLE REALS
        SEARCH MONSYM
NUM:    12.34                   ;note decimal point
START:  MOVEI 1, .PRIIN
        FLIN%
         Z
        FADR 2, NUM
        MOVEI 1, .PRIOU
        SETZ 3,                 ;format
        FLOUT%
         Z
        HALTF%
        END START
```

INDEX

Assembly Directives

Monitor Calls

STEPHEN A. LONGO

INTRODUCTION TO DECSYSTEM 20™ ASSEMBLY PROGRAMMING

37 B 4700
ISBN 0-534-02942-6