STEPHEN A. LONGO

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*

Introduction to DECSYSTEM-20™ Assembly Programming

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).

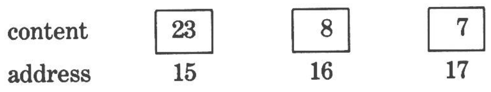


Figure 1-1 Memory Locations (Addresses) and Memory Contents

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.

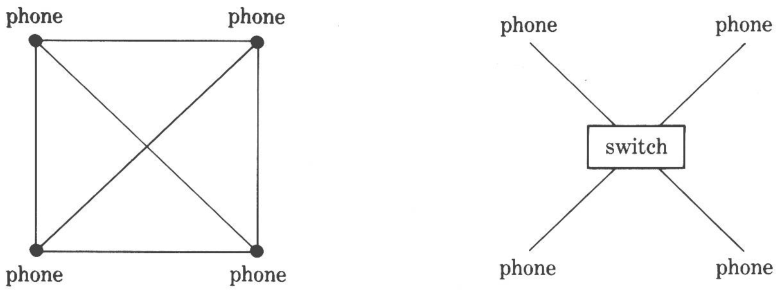
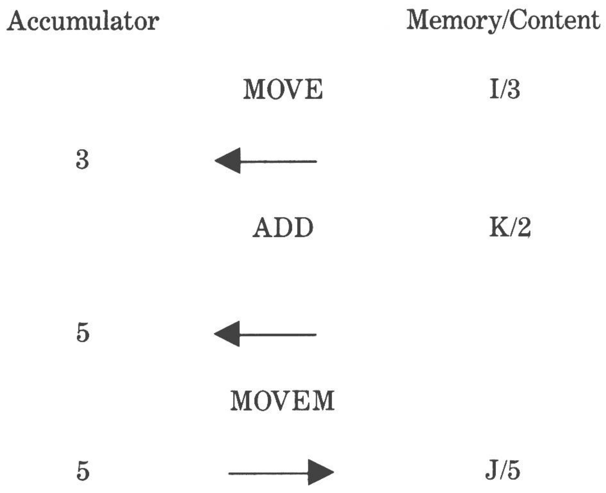


Figure 1-2 Telephone Interconnection Models

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 represe~~nted 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: