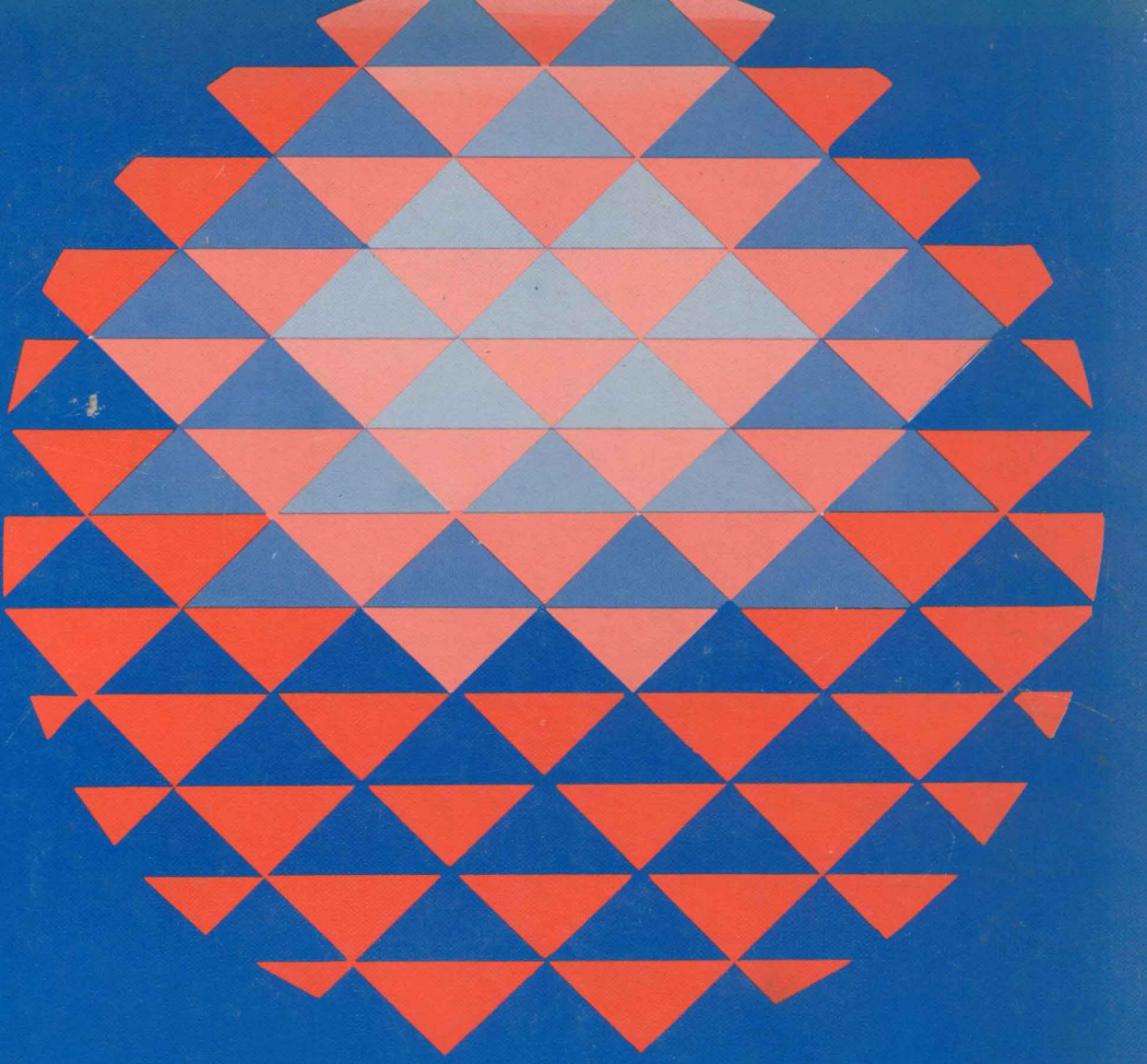

an introduction to computer programming and data structures using MACRO-11

harry r. lewis

An Introduction to Computer Programming and Data Structures Using MACRO-11

Harry R. Lewis

Aiken Computation Laboratory
Harvard University
Cambridge, Massachusetts

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

Library of Congress Cataloging in Publication Data

Lewis, Harry R
An introduction to computer programming and data structures using MACRO-11.

1. MACRO 11 (Computer program language)
2. Electronic digital computers—Programming.
3. Data structures (Computer science) I. Title.
QA76.73.M23L48 001.64'24 80-25450
ISBN 0-8359-3143-9

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

For my father

Acknowledgments

This book grew out of my experience teaching Applied Mathematics 110 at Harvard, and many people who have influenced that course have had an impact on the form and content of the book. In particular, some of the programming projects are lineal descendants of assignments designed by my forebears, some of whom I cannot identify. Certainly Tom Cheatham and Ben Wegbreit, who taught the course before I did, leave the largest imprint on the approach and choice of material I have adopted. I have received invaluable help both from my own course staff and from the staffs of earlier years. Some of those who deserve mention are Bob Case, Larry Denenberg, Bob Greenberg, Jeff Herrman, Michael Kahl, Henry Leitner, Geoff Peck, Eric Roberts,* Walter Scott, and Tucker Taft. To them I express my gratitude and my apologies for the lack of more specific attributions; and to the others, not mentioned here by name, I apologize for my ignorance of their contributions. And to the students in Applied Mathematics 110 in 1979 and 1980 I extend my thanks and admiration for their endurance of this material during its development.

* Who coined the phrase "MACRO-11 as a higher-level language."

Preface

This book is intended for an introductory computer science course at the undergraduate level. It assumes a minimal acquaintance with computer programming in some form; no specific material is prerequisite, but students will find it helpful to have a general understanding of the nature of programming. The main subjects covered are basic programming concepts, the use of standard types of data structures, and the elements of software such as assemblers and compilers. The programming language used throughout is the MACRO-11 assembly language for the Digital Equipment Corporation PDP-11 family of computers.*

There are several novelties in our approach and presentation. Treatments of introductory computer science that use assembly language as the medium of expression usually begin with an exhaustive discussion of machine structure and architecture, moving only much later to the construction of nontrivial programs. That sequence of material has two main disadvantages. First, it is difficult to coordinate a series of programming exercises with the introduction of architectural ideas, since fairly simple programming concepts, such as subroutines, may require the detailed explanation of rather complex aspects of instruction decoding and execution; consequently, the teacher may spend many weeks detailing dry material on machine behavior while the student can be assigned only paper-and-pencil exercises to increase familiarity with the machine. The second disadvantage is that the design motivation for many architectural features can be understood only by someone who already has a grasp of computer programming. There arises a chicken-and-egg problem vis-à-vis the teaching of machine design and the teaching of programming.

Our approach is to explain the machine and the ways it can be

* PDP is a trademark of the Digital Equipment Corporation.

programmed simultaneously, both a bit at a time. We do this by treating assembly language *as though it were a higher-level language,* and only coincidentally as the symbolic representation of any machine language. We introduce the assembly language by example, rather intuitively at first, and then in greater specificity. In fact, we discuss machine architecture only to the extent necessary to produce working programs. It is assumed that the student has access to the manufacturer's processor handbook in order to answer detailed questions about the operation of individual instructions, although a summary is given in Appendix 1. Similarly, we do not present the complete details of the MACRO-11 assembly language; we omit features, such as conditional assembly, unlikely to be of use to the novice programmer.

We assume that the student has access to a time-shared PDP-11 with facilities for loading and running programs and supporting buffered input and output. At Harvard we are using the UNIX* operating system, and the influence of that particular system is visible in parts of our presentation. But the bulk of the book does not depend on the details of the supporting software. We assume that a few macros are available to aid in programming terminal input-output operations, and in Appendix 4 we supply definitions of those macros for our UNIX environment; alternative definitions of the same macros could be given for other operating sytems.

We include a large number of short exercises and a carefully chosen set of six longer programming projects to illustrate the ideas of the main text. The short exercises immediately follow the sections of the text to which they are relevant; the projects are grouped in Part 3. Project 1 can be completed after Part 1 of the text has been read; Project 2, after Part 2A; Project 3, after Part 2B; Project 4, after Part 2C; Project 5, after Part 2D; and Project 6, after Part 2E.

Since the book is intended as an introduction to computer science and not just to computer programming, our choice of programming projects is oriented towards the explanation of basic computer software, such as text editors, assemblers, and compilers, rather than application programs from (for example) finance or the data processing industry. In this way we can achieve two goals at once: instruction in the skill of computer programming and instruction in the theory of software. It has been our experience that students who are introduced to computer science in this way can readily transfer their expertise to other computers and languages and are well prepared either for employment as programmers or for more advanced courses on programming and algorithms.

* UNIX is a trademark of Bell Telephone Laboratories.

Contents

Part 1

MACRO-11 Programming

1.1. *INTRODUCTION*

Our aim is to teach you to program in a particular programming language called MACRO-11. As it happens, the MACRO-11 programming language was designed with a particular computer in mind, the Digital Equipment Corporation PDP-11. However, you can begin to learn MACRO-11 without at first knowing anything about the PDP-11; we shall introduce various aspects of the machine as they become necessary to understand more complicated constructs of the language. And there are some characteristics of the machine and the language we shall not bring up at all.

There is a program called the MACRO-11 *assembler* that translates programs written in the MACRO-11 language into the native language of the PDP-11 computer. The MACRO-11 program being translated into PDP-11 machine language is called the *source* program; the result of the translation is called the *object* program. The MACRO-11 assembler is itself a program that runs on the PDP-11, so that when the PDP-11 has been used to translate the source program, the resulting object program can be run on the same machine.

We shall assume that you have written a few programs in some higher-level language such as BASIC, PPL, FORTRAN, Pascal, or a version of Algol; you will be expected to know, for example, what a loop is. If this assumption is not true for you, you can take what follows as a general introduction to programming, and you will be able to understand most of what we have to say. But in that case you should be prepared to spend a little extra time studying our examples and getting help from more experienced programmers.

In any event, our style is informal and our presentation is meant to be tutorial rather than exhaustive. For further details, you may wish

to consult the *PDP-11 Processor Handbook* and the *MACRO-11 Assembler Programmer's Manual* (Digital Equipment Corporation, Maynard, Massachusetts).

1.2. BOXES

Let us begin at the beginning. This is a MACRO-11 program:

```
MOV     A,SUM
ADD     B,SUM
HALT
```

It consists of three *instructions:* a MOV instruction, an ADD instruction, and a HALT instruction. To understand what these instructions do, you must have the following picture in mind: A, B, and SUM are the names of boxes where items of information can be stored. Boxes such as A, B, and SUM are called *memory locations.* The items of information stored in boxes are in this case numbers—positive and negative integers and zero, to be precise. The program can then be read:

Move whatever is in box A into box SUM.
Add whatever is in box B into box SUM.
Halt the computer.

The three instructions are to be executed sequentially, so the net effect is to add the numbers in A and B together and to put the result in SUM.

A and SUM are called the *operands* of the instruction

```
MOV     A,SUM
```

A is called the *first* or *source* operand, and SUM is called the *second* or *destination* operand. Similarly, in the instruction

```
ADD     B,SUM
```

B is the source operand and SUM is the destination operand.

Note two things: first, the MOV and ADD instructions work from *left* to *right* (move A into SUM, add B into SUM); second, the HALT instruction ends the execution of the program. In practice we shall not want actually to stop the machine at the end of a program; indeed, because our PDP-11 is assumed to be time-shared, we shall not be able to do so. But for now it is simplest to pretend that we shall.

1.3 INITIALIZING MEMORY LOCATIONS

How can we put numbers in the boxes A and B to begin with? A quantity of information that is the right size to fit in one box is called a

word. In MACRO-11 we *label* a box with a particular name, for example A, by writing

```
A:
```

and we say that the number to go into that box is (for example) 5 by writing

```
A:      .WORD     5
```

This is not a PDP-11 instruction; it is simply a statement to the MACRO-11 assembler that a box named A should be set up with the number 5 in it. Such a statement is called a *directive;* directives are recognizable by the fact that they always begin with a period. Of course, we also want to set aside a box for B, say

```
B:      .WORD     -3
```

We also need a box for SUM, but in this case there is no need to specify what should be in the box; the purpose of the program is, after all, to put something (the sum of whatever is in A and whatever is in B) into the box named SUM. So we write

```
SUM:    .WORD
```

which simply holds a box named SUM without specifying what goes in it.

Actually, .WORD by itself means the sme as .WORD 0; but if it were important that SUM start off containing 0, then we should write .WORD 0 explicitly instead.

Thus our whole program now looks like:

```
        MOV       A,SUM
        ADD       B,SUM
        HALT
A:      .WORD     5
B:      .WORD     -3
SUM:    .WORD
```

A label may, by the way, consist of any combination of one to six letters and digits of which the first must be a letter. The same rule applies to other kinds of symbols in MACRO-11. The exceptions to the rules are certain special-purpose symbols, which may contain periods (.) and dollar signs ($) as well, but these are symbols made up by systems programmers. The ones you create should not contain periods and dollar signs.

1.4. CONTENTS

After the program has been run, or *executed* as we say, the SUM box will contain the sum of 5 and −3, or 2. That is, the *contents* of SUM will be 2. In general, we say "the contents of . . ." instead of "whatever is in. . . ." We also have a shorthand notation for the same thing: we write (A) for the contents of A and (SUM) for the contents of SUM. Since the contents of a box may change as a program is run, the value of (SUM) may differ depending on the point in the execution of the program at which (SUM) is mentioned. Thus (SUM) starts off being 0; after the first instruction is executed, (SUM) is 5; and after the second instruction is executed, (SUM) is 2.

Using Algol-like notation, we describe the effect of the instruction

```
MOV     A,SUM
```

by

```
(SUM)  ←  (A)
```

and the effect of the instruction

```
ADD     B,SUM
```

by

```
(SUM)  ←  (SUM)+(B)
```

Here the ← means "becomes" or "is changed to"; so

```
(SUM)  ←  (SUM)+(B)
```

is read "the contents of SUM becomes the (previous)contents of SUM plus the contents of B."

1.5 ADDRESSES

Actually, the PDP-11 *computer* (as opposed to the MACRO-11 *assembler*) does not know anything about symbolic names such as A and SUM. Instead, the memory locations have *addresses,* which are numbers (nonnegative integers). The address of a memory location is a different thing from the number that the memory location contains, just as the street address of a house is different from the person who lives in it. The address of a memory location always stays the same, but the program may store different things in that memory location at different times. Until further notice, all addresses will be *even* numbers.

Writing

```
A:
```

sets up a correspondence between the symbol A and a particular ad-

dress, say 50. In general we neither know nor care to what address a particular symbol refers; all addresses are created equal, and in MACRO-11 we are freed from having to make arbitrary choices in this respect. What is useful to know, however, is that if we specify a sequence of things to be put into boxes, MACRO-11 will put them at sequential addresses. *Sequential addresses* here are addresses that differ by two, since for the time being all addresses are assumed to be even. Thus if we write

```
TAB:    .WORD    3
        .WORD    -1
        .WORD    4
        .WORD    -5
```

the MACRO-11 assembler will pick an address corresponding to the symbol TAB, say 30, and then put 3 in the memory location with address 30, −1 in that with address 32, 4 in that with address 34, and −5 in that with address 36. Another way of saying it, which avoids mentioning the exact address of TAB, is that (TAB) will be 3, (TAB+2) will be −1, (TAB+4) will be 4, and (TAB+6) will be −5. Now TAB itself is not 3—that is the *contents* of the memory location TAB. The number that is meant by TAB itself is the address corresponding to this symbol, 30 in our example. That is why we can write TAB+2 to mean 32. To say it again,

```
TAB      is 30
TAB+2    is 32
(TAB)    is (30) is 3
(TAB+2)  is (32) is -1
```

and, if there were any reason to talk about it, (TAB)+2 would be (30)+2, that is, 5.

A sequence of memory locations like TAB, TAB+2, TAB+4, and TAB+6 that contain similar information (in this case, four numbers) is called a *table.* The individual items of information (the four numbers) are called *entries* in the table. Tables are our first (and for a while, our only) example of information-organizing devices called, as a class, *data structures.*

By the way, the same effect could be accomplished by writing

```
TAB:    .WORD    3,  -1,  4,  -5
```

instead of

```
TAB:    .WORD    3
        .WORD    -1
        .WORD    4
        .WORD    -5
```

Exercises

1.5.1 What does this instruction do?

```
MOV     TAB+6,SUM
```

Explain it in English and write it using the Algol-like notation.

1.5.2 Suppose that the following is part of MACRO-11 program.

```
        .WORD 4
        .WORD 1
XYZ:    .WORD 6
        .WORD 2
        .WORD 3
```

What location is changed by the instruction

```
ADD     XYZ+2,XYZ-2
```

and what are the new contents?

1.6. THE SYMMETRY AND INITIAL VALUE PRINCIPLES

The following program computes the sum of 3, −1, 4, and −5, albeit in a fairly illogical way:

```
        MOV     TAB+4,SUM
        ADD     TAB+2,SUM
        ADD     TAB+6,SUM
        ADD     TAB,SUM
        HALT

TAB:    .WORD   3
        .WORD   -1
        .WORD   4
        .WORD   -5
SUM:    .WORD
```

There is no good reason to have the four numbers added in this order—it might not confuse the computer, but it would surely confuse any person attempting to understand the program. So this version is better:

```
        MOV     TAB,SUM
        ADD     TAB+2,SUM
        ADD     TAB+4,SUM
        ADD     TAB+6,SUM
        HALT
```

```
TAB:    .WORD    3
        .WORD    -1
        .WORD    4
        .WORD    -5
SUM:    .WORD
```

Still, it is a little ugly to have the first item treated differently from the rest—being handled by a MOV instead of an ADD like the others. In general, programs should do the same thing in the same way and should avoid irregularities that serve no useful purpose. In other words:

> *Symmetry Principle.* Programs should be symmetric and uniform when performing similar functions, unless there is good reason to the contrary.

One may be tempted to enforce the Symmetry Principle simply by changing the MOV to an ADD; we did mention, after all, that .WORD by itself means the same as .WORD 0, so the modified program will add all four numbers to zero and get the right answer. Or if we did not care to remember that .WORD is the same as .WORD 0, we could in addition change the last line to

```
SUM:    .WORD    0
```

Unfortunately, each of these versions, while technically correct, violates another principle:

> *Initial Value Principle.* A memory location of which the contents is changed by a program should not be assumed to have any particular value at the beginning.

That is, since SUM is changed by this program, we should not set it up so that the program runs correctly only if SUM contains the value 0 initially.

One reason for the Initial Value Principle is that programs are often run more than once, and the program may run correctly only the *first* time if the principle is violated. In our example, SUM contains 1 after the program is run the first time; if the suggested change (changing MOV to ADD in the first instruction) were made, then the second time the program were run, it would incorrectly leave SUM containing 2.

How can the Symmetry and Initial Value Principles be maintained in this case? By arranging the program so that it begins by putting zero in SUM, before adding the four numbers to SUM. Of course, this involves setting aside a box that we can be sure will always contain zero, but we know how to do that. Thus the following program does the trick: