Karen A. Lemone
Martin E. Kaliski

Assembly Language Programming for the VAX-11

Foreword by Gerald M. Weinberg

Little, Brown Computer Systems Series

Assembly Language Programming for the VAX-11

Little, Brown Computer Systems Series

Gerald M. Weinberg, *Editor*

Barnett, Michael P., and Graham K. Barnett
Personal Graphics for Profit and Pleasure on the Apple II Plus Computer

Basso, David T., and Ronald D. Schwartz
Programming with FORTRAN/WATFOR/WATFIV

Chattergy, Rahul, and Udo W. Pooch
Top-down, Modular Programming in FORTRAN with WATFIV

Coats, R. B., and A. Parkin
Computer Models in the Social Sciences

Conway, Richard, and David Gries
An Introduction to Programming: A Structured Approach Using PL/I and PL/C, Third Edition

Conway, Richard, and David Gries
Primer on Structured Programming: Using PL/I, PL/C, and PL/CT

Conway, Richard, David Gries, and E. Carl Zimmerman
A Primer on Pascal, Second Edition

Cripps, Martin
An Introduction to Computer Hardware

Easley, Grady M.
Primer for Small Systems Management

Finkenaur, Robert G.
COBOL for Students: A Programming Primer

Freedman, Daniel P., and Gerald M. Weinberg
Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products, Third Edition

Graybeal, Wayne, and Udo W. Pooch
Simulation: Principles and Methods

Greenfield, S. E.
The Architecture of Microcomputers

Greenwood, Frank
Profitable Small Business Computing

Healy, Martin, and David Hebditch
The Microcomputer in On-Line Systems: Small Computers in Terminal-Based Systems and Distributed Processing Networks

Lemone, Karen A., and Martin E. Kaliski
Assembly Language Programming for the VAX-11

Lias, Edward J.
Future Mind: The Microcomputer—New Medium, New Mental Environment

Lines, M. Vardell, and Boeing Computer Services Company
Minicomputer Systems

Mashaw, B. J.
Programming Byte by Byte: Structured FORTRAN 77

Mills, Harlan D.
Software Productivity

Foreword

Sometimes I worry that my position as Series Editor gives me too much power to indulge my prejudices. For instance, it's very easy for me to reject a book on teaching your turkey to program in FORTRAN, because I'm slightly prejudiced against turkeys, substantially prejudiced against FORTRAN, and completely prejudiced against teaching turkeys to program.

On the other hand, it was very easy for me to accept *Assembly Language Programming for the VAX-11*—and not just because it isn't about FORTRAN or turkeys. Lemone and Kaliski have written a superbly crafted course in assembly language for readers with some prior experience in programming higher level languages. Their effort thus appeals simultaneously to three of my long-standing predilections—for good writing, for assembly language, and for teaching assembly language to anyone seriously interested in the practice of programming. Perhaps I'd better explain my bias in favor of this book, so you can judge for yourself.

It may be difficult to explain my prejudice in favor of good writing because, in the more technical subjects, good writing is so rare that some readers may never have seen it. In reading about a subject like assembly language, many readers get turned off because the writing is poor, not because the subject is difficult. They are novices, and they have no way to separate the dancer from the dance. They simply put down the book and give up on the subject.

But teachers who adopt Lemone and Kaliski's *Assembly Language Programming for the VAX-11* don't have to fear that the writing is going to turn the students against the course. The book is written clearly, with precision, and at just the right level. It is comprehensive without being superficial, detailed without being trivial, and altogether pleasant to read.

How can a book on assembly language be "pleasant to read"? That's my

prejudice again! Assembly language was my first language, and my second and third. The first book I ever wrote was on assembly language, as was the first computer course I ever taught. These experiences do give me a bias in favor of assembly language that some may not share, but on the other hand they also give me a prejudice *against* any author who mistreats assembly language. I find Lemone and Kaliski to be sensitive to both the opportunities and limitations of assembly language. If this is prejudice, then I plead guilty.

Assembly language is not just fun to read about—it is an essential part of the education of any true computer professional. The first assembly language course is a pivotal course for the budding computer scientist or electrical engineer. In fact, assembly language is pivotal precisely because it is the meeting ground between the two disciplines. Not every teacher may agree with this prejudice of mine, but after more than a quarter century of training both computer scientists and engineers, I'm not going to be talked out of it easily. I've seen too many students pulling it all together for the first time in the assembly language course.

And pulling it all together is what Lemone and Kaliski do. I particularly like their treatment of assembler design issues as a way of making the course into something more than an unordered collection of random facts. The student who works through *Assembly Language Programming for the VAX-11* will have a real feeling of accomplishing something worthwhile, and will be well prepared to move in any one of several different directions for more depth—hardware design, design of languages, design and implementation of translators, or specific assembly languages on other machines.

So if you are teaching or learning assembly language on the VAX-11, I recommend that you use Lemone and Kaliski's book. But since I am prejudiced, why don't you see for yourself?

Gerald M. Weinberg

Preface

This is a two-part text about assembly language programming in the VAX/MACRO language. Unlike many texts on assembly language that are concerned solely with the assembly language per se, this text also addresses the design of assemblers, macroprocessors, and linkers. It is divided into two stylistically different parts.

In Part I the fundamentals of assembly language programming in the VAX/MACRO language are discussed. It is aimed at the beginning assembly language programmer, conforming with current ACM recommendations concerning introductory assembly language programming courses.

Chapter 1 introduces the basic vocabulary and concepts of assembly language programming. It is a learn by doing chapter that encourages the reader to think in assembly language terms and serves to motivate the ensuing discussion in Chapters 2 and 3. In Chapter 2 the VAX organization and architecture are discussed. Chapter 3 covers the binary, decimal, and hexadecimal number systems and describes the ways that data can be stored in memory. Data storage directives are introduced in this chapter.

Chapter 4 describes the various addressing modes in VAX/MACRO and their uses. The instruction set is also introduced. After having completed Chapter 4, the student should be able to write simple programs in the VAX/MACRO language.

Chapter 5 describes some fundamental assembly language programming constructs, relating them to analogous higher-level language constructs (in FORTRAN, BASIC, Pascal, and pseudo-code). Topics such as assignment statements, conditional statements, loops, and array operations are addressed. Chapter 5 is the heart of Part I.

Chapters 6 and 7 discuss, respectively, macros and subroutines/procedures. The reasons for using these techniques are examined, and through the examples of these chapters the material of Chapters 4 and 5 is solidified. Input/output program-

ming is studied in Chapter 8. Because this chapter is highly dependent upon the VAX/VMS operating system, the reader may prefer alternative material on input/output widely available. Chapter 9 provides an introduction to more advanced techniques in assembly language programming, such as conditional assembly and character string manipulation.

The flavor of the discussion changes in Part II of the text: Part II's system viewpoint complements the user's point of view.

Chapter 10 is concerned with the basic issues of assembler design, taking a modular approach to the software design of assemblers. Chapter 11 extends the methodology of Chapter 10 to macroprocessor design issues, and Chapter 12 discusses linker design.

The treatment of these topics in Part II contrasts with the basic approach of Part I. The discussion is more general and the exercises more advanced. It is hoped that this will serve to round out the reader's knowledge of assembly language and assembly language programming techniques.

There are five appendices to this text. Appendix A presents introductory material allowing the reader to use the VAX/VMS operating system. Appendix B highlights the design issues not covered in Chapter 10, as does Appendix D for Chapter 11. Appendices C and E attempt to define a restricted version of VAX/MACRO, called SUBMAC, suitable for use in system software design projects.

Acknowledgments

Although we have made every effort to eliminate errors, it is possible that some still exist. If so, we would appreciate knowing about them. Feel free to write to either of us or to the editors at Little, Brown. If possible, we will answer.

We would like to mention that the errors you *don't* see were detected in previous drafts by the following people (to mention just a few): Brian Alves, Susan Blyde, Willy Burgess, John Crosby, Sharon Giggey, Chris Hacket, Whitney Harris, Ellen Hollis, Lenny Leffand, Riad Loutfi, Steve Morth, Heyedeh Motallabi, Duane Pawson, Hank Thoelke, Richard Tyson, Professor Tom Westervelt, Mark Woodbury, and the entire Winter '81 6.130 class at Northeastern University. We apologize to anyone we have left out! The errors that remain are all ours.

In addition, we would like to thank Professor Richard Carter for having the courage to use this text in some of its previous versions and the people at Massachusetts Computer Associates for sharing their expertise in computering.

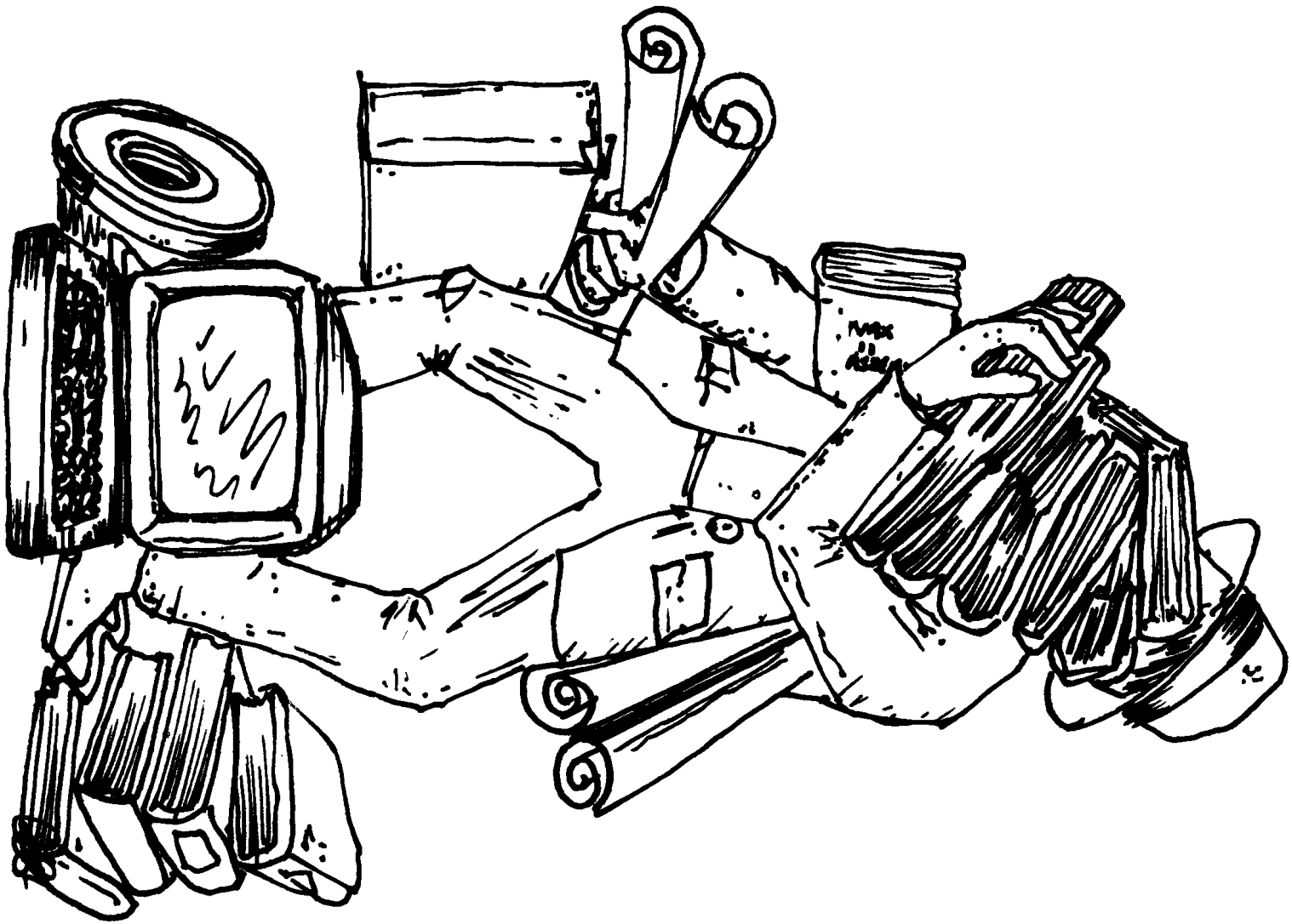
This text was typed and edited on line by Laurie Reynolds, Jonathan Chappell, Kathi Marks, Hank Thoelke, Audrey Aduama, and the authors. The excellent drawings are by George Capalbo. Special thanks go to Hank Thoelke for his unselfish help throughout this project.

Karen A. Lemone and Martin E. Kaliski

Contents

Part I

VAX/MACRO Assembly Language



Chapter 1

Getting Started

This chapter introduces the VAX-11 computer and some vocabulary and concepts of assembly language programming.

Readers familiar with a compiler language such as BASIC, COBOL, FORTRAN, or Pascal are familiar with the way a set of instructions solves a problem. These compiler languages deal with inputs and outputs and the algebraic manipulations necessary to convert one (inputs) into the other (outputs). Compiler languages operate on inputs and outputs; there is little need to know what the computer is actually doing with the data. Languages such as assembly language and machine language, however, operate more directly on the machine or the machine parts. For example, arithmetic generally takes place in high-speed storage locations called *registers*. In an assembly language or machine language program, we refer to these registers directly. Thus, we need to know something about registers—how many there are, which ones to use, how to refer to them, what size they are, and so on. Most compiler languages make no mention of registers at all. The compiler decides what registers, if any, to use. But any discussion of assembly language or machine language must include an explanation of registers and various other parts of the computer (known collectively as the machine *architecture*). The VAX machine architecture will be described in Chapter 2. This chapter defines the terms *machine language* and *assembly language* and compares them with *compiler languages*.

1.1 Machine Language and Assembly Language

Machine language

Machine language is the computer's "native" language. There are only two symbols in machine language. These are 0 and 1, which are called binary digits or

bits for short (from *b*inary dig*it*). Each statement in machine language consists of a sequence of bits called a *bit pattern:*

```
010101100000001011010000
```

An executable computer program is nothing more than a stored collection of bit patterns. These bit patterns are stored in the part of the computer known as *memory*. Each of these bit patterns may represent an instruction, a piece of data, or even the location of an instruction or piece of data.

In the preceding example, the rightmost eight bits represent the instruction called "Move." The next eight bits represent the datum "2," and the leftmost eight bits stand for "Register 6" (denoted by R6):

```
010101100000001011010000
```

R6 2 Move

Thus the bit pattern for the statement "Move the constant 2 into Register 6" contains an instruction (Move), a piece of data (2), and a reference to a machine part (R6). Notice that the instruction must be read from right to left!

A separate component of the computer called the *central processing unit* (denoted CPU) interprets these bit patterns that have been stored in the computer's memory. The CPU is designed to recognize what is an instruction and what is a datum. It is the CPU that executes a machine language program that has been stored in memory. That is, the CPU understands this machine language. Humans, however, find such sequences of 0's and 1's somewhat incomprehensible and quite difficult to remember. Dealing with bit patterns requires the programmer to remember the numeric code for each instruction and the location in memory of each data item—all in binary. For these reasons we do not write programs in machine language if we can avoid it. Instead we write programs in a more understandable form: *assembly language*.

Assembly language

In assembly language, as in compiler languages, a data item may be addressed by a symbolic name such as *a, min, result,* and *factorial*. Also a descriptive, mnemonic instruction is used instead of a numeric code.

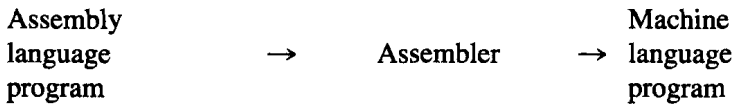
EXAMPLE

```
movl #2,r6
```

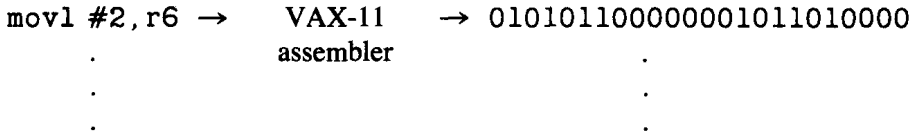
is the VAX-11 assembly language code for the machine code of the previous example. It is much easier to believe and remember that this means "Move a 2 into Register 6."

Unfortunately, computers cannot understand assembly language; they understand only the bit patterns of machine language. Thus, before an assembly

language program can be executed, it first must be translated into machine language. A program called an *assembler* performs this translation. (Part II of the text discusses the design of assemblers in greater detail.)



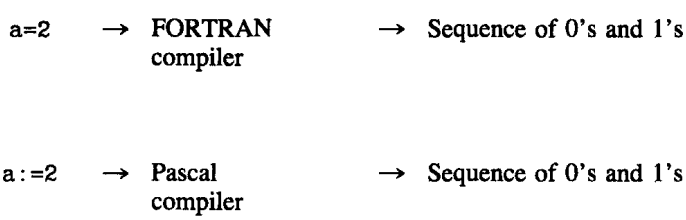
EXAMPLE



There is exactly one machine language instruction for every assembly language instruction.

Compiler language

In languages such as FORTRAN, BASIC, COBOL, and Pascal a program called a compiler frequently translates the instruction into machine language. Consider the FORTRAN statement *a=2* and the Pascal statement *a:=2*.



1.2 Thinking in Assembly Language

*Calculating 2*3+4*

Programming in assembly language is similar to operating a calculator. Each data value must be entered and, in general, each operation such as addition or multiplication must be performed separately. Compiler languages, on the other hand, can frequently perform more than one operation in a single statement. Consider the following assignment statement:

```
result=2*3+4
```

which multiplies 2 times 3, adds 4, and assigns the result to the variable named *result*. (In some languages such as Pascal this would be written "result:=2*3+4".) To accomplish this statement in assembly language, we must first enter the value 2. Register 6 will be chosen (at random) to hold this value. To enter the number 2 into Register 6:

```
movl #2,r6
```

The instruction *movl* tells the CPU to copy the number 2 into Register 6. Next, multiply the contents of Register 6 by 3:

```
mull2 #3,r6
```

and lastly,

```
addl3 #4,r6,result
```

adds a 4 to the contents of Register 6 and moves a copy of the sum to the *location* whose *symbolic name* is *result*. The entire sequence is

```
movl  #2,r6
mull2 #3,r6
addl3 #4,r6,result
```

and is one way of calculating 2*3+4 and storing the answer at the location whose name is *result*.

Instruction parts

Notice that there are two parts to each of the instructions above—an operation and some operands (we will see two optional parts later):

```
movl     #2,r6
 ↑        ↑
Operation Operands
```

Some operations end in 2 (e.g., *mull2*). This indicates that there are two operands (e.g., #2 and R6). Similarly, an instruction ending in 3 has three operands. For example, *addl3* above has three parts to its operand—#4, R6, and *result*. Note that some instructions (e.g., *movl*) have neither a 2 nor a 3, which indicates a fixed number of operands. *movl* always has two operands.

*Calculating a*b+c*

Next, consider the more general case of calculating a*b+c and storing the answer in *result:*

```
result=a*b+c
```

In VAX assembly language this becomes