VAX ASSEMBLY LANGUAGE

FRANCIS D. FEDERIGHI
EDWIN D. REILLY

MAXWELL
MACMILLAN
INTERNATIONAL
EDITIONS

VAX Assembly Language

Francis D. Federighi
Union College

Edwin D. Reilly
State University of New York at Albany

Macmillan Publishing Company
New York

Collier Macmillan Canada
Toronto

Maxwell Macmillan International
New York Oxford Singapore Sydney

Macmillan Publishing Company
866 Third Avenue, New York, New York 10022

Collier Macmillan Canada, Inc.
1200 Eglinton Avenue, E.
Suite 200
Don Mills, Ontario, M3C 3N1

Library of Congress Cataloging-in-Publication Data

Federighi, F.D.
VAX assembly language/Francis D. Federighi, Edwin D. Reilly.
p. cm.
1. VAX computers-Programming. 2. Assembler language
(Computer program language) I. Reilly, Edwin D. II. Title.
QA76.8.V32F43 1991 89-13917
005.2'45—dc20 CIP
ISBN 0-02-399255-7 (Hardcover Edition)
ISBN 0-02-946385-8 (International Edition)

IE Printing: 1 2 3 4 5 Year: 1 2 3 4 5

ISBN 0-02-946385-8

Preface

This text is intended for use by students who have prior programming experience in at least one high-level language such as Pascal, FORTRAN, or BASIC. It is nonetheless our observation that most of the students who enter our assembly language programming course—even those who profess to know three or four high-level languages—do not really know what a computer is. Our primary goal is to ensure that when they complete the course successfully, they will. There is a certain something about having to take full responsibility for every (literal) bit of a machine or assembly language program that no amount of practice with an algebraic language can provide.

The secondary goal of an assembly language course is to enable its students to attain reasonable fluency in a particular assembly language. Within reason, it does not matter which assembly language is taught, both because there is a common conceptual thread that runs through all of them and because, no matter which is chosen, students are virtually certain to be confronted with a different one at some later stage of their career. Because it provides such an excellent environment for initiation to assembly language concepts, we have chosen to base this text on the Digital Equipment Corporation's VAX assembly language running under the VMS operating system.

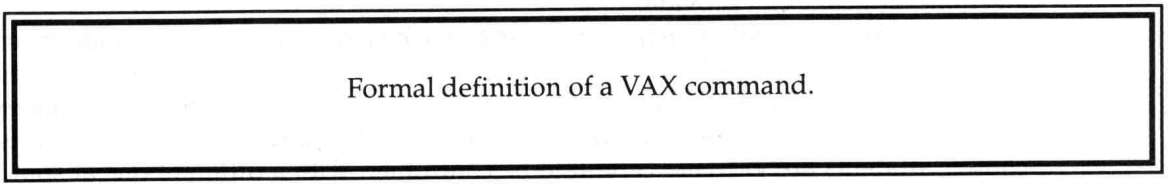
There are many texts that teach VAX assembly language but, if we have attained our goal, none like this one. Our approach is to emphasize just those classes of instruction that are likely to have counterparts on other computers that students might encounter. Very specialized instructions such as those that evaluate polynomials, manage queues, or emulate Pascal **case** statements are treated briefly in Chapter 14, which can be omitted if time is short.

A concern that we have with competitive texts is their surprising lack of programming problems. Most are full of small esoteric questions as to what bit pattern will result from such and such a combination of instructions, but one must look long and hard to find a substantive and challenging programming assignment. Since we are firm believers that students must learn by doing, this text contains far more programming problems and relatively fewer short-answer questions than most other VAX texts. This book has enough problems to allow the instructor to assign 10 to 12 per semester and yet be able to vary the assignments from semester to semester.

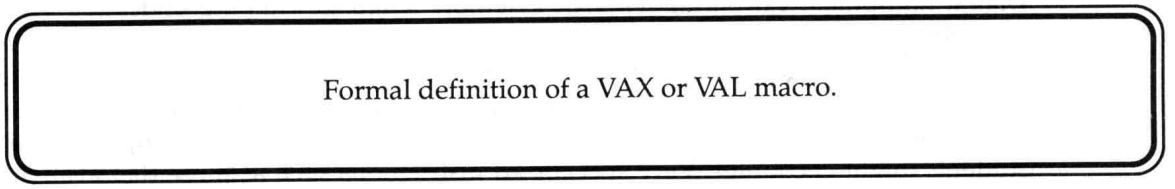
The text begins by describing the architecture of a simplified 10-instruction computer called VAXINE, a "little VAX" that is an abstraction of its more complex

progenitor. Then, in keeping with our stated primary objective that students finally learn what a computer is, they are asked to write a bit-faithful simulator for VAXINE in a high-level language of their choice. We strongly advise that this assignment not be skipped. VAXINE was not invented in some misguided attempt to protect the student from a too-early exposure to the full complexity of the VAX. The objective is not to make students write programs that are processed by a simulator supplied by the instructor, but rather to make students write their own simulator to run programs written by the instructor. The students who do this successfully—and most will—cannot fail but gain an appreciation for the basic interpretive cycle and operand addressing modes that form the fundamental basis for digital computation. This first assignment, a simulator, and the last assignment, chosen from those in Chapter 13, are the heart and soul of the text.

The final goal of this text is that it be a reasonably complete reference to the entire VAX instruction repertoire and to the VAX MACRO assembler. For this reason, we have included a large number of appendices and have also given complete formal definitions of all commands and macros as they were introduced. To highlight and differentiate between these formal definitions, we have placed those for commands in square-cornered boxes



and those for macros in round-cornered boxes



Some students who are reasonably adept at using BASIC or Pascal never do learn to cope with a tool as precise, demanding, and unforgiving as an assembler, but those who do usually report that working in such a medium is a stimulating intellectual adventure and great fun. We think so too, and so we have done our best to write in a style that lets that belief show through. Let the adventure begin.

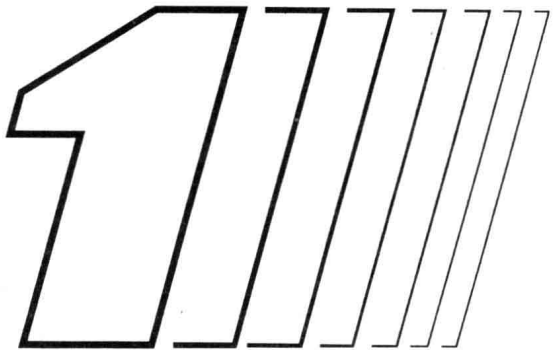
F.D.F.
E.D.R.

Acknowledgments

We wish to thank the many colleagues and students who read drafts of the manuscript of this book and helped with its creation. We especially wish to thank Joe Haas, David O'Connor, and Barry Werger who created much of the software needed to make the VAL system work. We are grateful to the many students at State University of New York at Albany who helped refine the final draft of this text, especially Bernard Badgley, Nancy Bartsch, Lance Biesele, David Damo, Andy Gallo, Ginni Goggins, Joe Houghtaling, Brian Kell, Todd Kornfeld, Bill Rennie, Judith Swota, and Brian Victor. We also want to mention the following reviewers who were recruited by our Macmillan editors; each in his or her way has helped make this a better book:

Claude Anderson, Rose Hulman Institute of Technology
Kenneth Cooper, Jr., Eastern Kentucky University
John McCormick, SUNY Plattsburgh
Bina Ramamurthy, SUNY Buffalo
Paul Ross, Millersville State University
Kenneth Sochats, University of Pittsburgh

Contents



Digital Computers

Von Neumann was a great mathematician and had the reputation at that time of being the cleverest man in the world. He was supposed to be the intellectual force driving the whole development of computers. He was a great thinker and a great entrepreneur. And yet he totally misjudged the role that computers were to play in human affairs.

— FREEMAN DYSON
Infinite in All Directions

Just as the proverbial blind men gave very different descriptions of an elephant, depending on which of its parts each was able to examine, there are many different answers to the question "What is a digital computer?" To the electrical engineer, it is the collection of circuitry that does the computer's fundamental arithmetic and logical operations. To the motor vehicle clerk, the computer appears to be a special-purpose device that keeps track of car and driver licenses. To the Pascal or FORTRAN programmer, it is an algebraic engine that seems to know how to evaluate formulas. This chapter will introduce yet another view, namely, that a digital computer is a faithful servant that will, indefinitely and without tiring and without error, execute a sequence of commands, each of which is chosen from a well-defined repertoire.

This chapter will examine the fundamental principles of how stored-program digital computers work in general and how a simple one is programmed in particular and will suggest that the best way to master such a computer's functions is to simulate that computer by using a high-level language program of the student's own creation.

1.1 Fundamental Principles of a Digital Computer

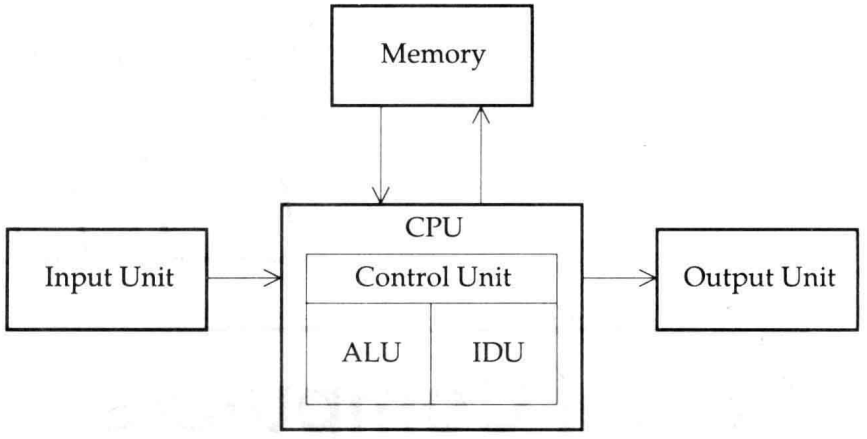


Figure 1.1 The Principal Components of a Stored-Program Digital Computer

Schematically, at least, a typical digital computer consists of the components shown in Figure 1.1. Data read from an *input unit*—perhaps the keyboard of a terminal or personal computer—is stored temporarily in the computer's *memory*. This input data is processed by the electronic circuits of the *arithmetic and logic unit* (ALU) in the *central processing unit* (CPU), according to a sequence of instructions called a *program*. The results from the programmed computation are then sent to an *output unit*—perhaps a printer or the display screen of a terminal or personal computer.

In the earliest days of digital computation—up through 1940 or so—program instructions were fed to the computer, obeyed, and discarded. The only way that an identical sequence of instructions could be repeated was to reenter the sequence through the input device. Beginning early in the 1940s, it occurred to several people independently that the instructions could be read just once and stored in memory just as if they were data and that important benefits could be derived from doing so. Computers that can do this are called *stored-program computers* or, after one of these early leaders, *von Neumann computers*. Thus the CPU of a stored-program computer must contain—in addition to the ALU—a *control unit* that determines the sequence in which the instructions and data are selected from memory and also an *instruction decoding unit* (IDU) which decodes the digits of an instruction to determine what the ALU is to do.

Virtually all modern digital computers store their data (numbers, instructions, etc.) and do their arithmetic and other operations in the binary number system, but this observation is not crucial to understanding the fundamental principles of stored-program digital computation. We can explain these principles just as well in the more familiar decimal number system; binary notation can wait for Chapter 2.

Computer Memory

Regardless of the arithmetic system used, both instructions and data are stored in a memory that has certain characteristics. The memory has both a width, which is related to the precision of the data it can store, and a depth, which is proportional to the number of data values it can store. Each data value occupies a unit of memory called a *word*, and each word is given a unique *location* or *address*, a number that ranges from zero for the first address up to some maximum value, the address of the last word. Figure 1.2 shows two different ways in which the memory of a hypothetical 10-decimal-digit computer might be arranged. When we learn the *architecture* of a new computer, we must learn what scheme is used with that particular design. The computer's design engineer chooses the addressing scheme, and we can only react to it.

Address	Content
00	+3418435437
01	-6540398821
02	-8720211058
03	+0000000003
04	+2045768109
	.
	.
49	+6482947082
	.
	.
99	+5657000034

a. One hundred sequentially numbered cells of a word-addressable machine. Each cell contains one word (depth—100 words).

Address	Content		Address
01	+34184	+35437	00
03	-65403	-98821	02
05	-87202	-11058	04
	.		
	.		
	.		
	.		
99	+64829	+47082	98

b. One hundred sequentially numbered cells of a halfword-addressable machine. Each cell contains a halfword (depth—50 words or 100 halfwords).

Figure 1.2 Two Alternative Designs for a Small Memory

As we can see from Figure 1.2, it is sometimes necessary to distinguish a *word* from a *cell*. Cells are the units of memory that are given sequential addresses, whereas words are the combinations of one or more consecutive cells that hold the operands that can be processed with the most common arithmetic commands: add, subtract, and (perhaps) multiply and divide. In Figure 1.2a, words are given sequential addresses, so with that memory arrangement, a cell is a word (or more precisely, a cell is said to hold one word of data). In Figure 1.2b, halfwords are given sequential addresses, implying that there are commands to manipulate data expressed in halfwords as well as word instructions that manipulate the two cells needed to hold one 10-digit word of data. Using a lower-numbered (even) address for the least significant half of a number (right halfword) and the next higher-numbered (odd) address

for the most significant half (left halfword) is a convention that we will see again on the Digital Equipment Corporation (DEC™) VAX™.

Note that an address is not related to the content of that address. In Figure 1.2a, cell 03 happens to contain a 3, but that is merely a coincidence. Other cells contain numbers clearly unrelated to their address. Cell zero, for example, contains `+3418435437`. Now, just what is the significance of that sequence of digits? Might it be the large number 3,418,435,437? Or might it be a coded instruction that tells the computer to do something? The answer is that we can't tell yet. If that part of the computer called the control unit in Figure 1.1 sends that digit sequence to the arithmetic and logic unit, then the sequence will be processed as if it were the large number quoted. But if it sends the same sequence to the instruction decoding unit, the ten digits will be torn apart and interpreted as instructions according to some scheme that we must learn for each new digital computer we seek to program.

Not only can't we tell the significance of the `+3418435437` without being given more information, but also the same sequence can be processed as data during one phase of a program and later be interpreted as an instruction. What might that instruction be? Let's describe one possibility. Suppose that the first two digits specify which of 100 different *commands* is to be obeyed and that command 34 is the ADD command. What is to be added? That information must be conveyed by the remaining digits of the instruction. Are we to add 18 + 43 + 54 + 37? 1843 + 5437? Probably neither, because stored-program computers are much more flexible. The *operands* (data to be operated upon) of the ADD are not usually embedded in the instruction itself but, rather, are stored at the memory addresses specified in the ADD instruction. This computer, for example, might process the instruction as meaning "Add the contents of cell 18 to the contents of cell 43; place the sum in cell 54; and then tell the control unit to fetch the next instruction from cell 37." A computer that worked that way would be called a *four-address computer.*

Are all four operand addresses needed? Not the last one, certainly. The design described would allow a program to hop all over memory, making it very hard for the human (but not the computer) to understand it. Any programmer using such a machine would probably put sequential instructions in sequential memory cells and deviate from this only when necessary to permit the program to take alternative paths depending on the input data or the results of previous calculations. More often the programmer would dispense with the fourth operand (reducing the word size to eight digits: `+34184354`), with the understanding that when the instruction being processed is finished, this *three-address computer* will go automatically to the next address in sequence. This sequence is normally cyclic, so when the machine has just finished executing the instruction at 99, it will go to 00 to find the next one. But suppose we occasionally want to break such a uniform sequence; what can we do? The need to do this occurs quite often, so the instruction repertoire of any computer will include commands that tell the computer to take its next instruction from whatever new address we specify, either unconditionally (always jump) or conditionally, depending on whether some number is zero, nonzero, or larger or smaller than some other number. Such jumps are also called *branches,* conveying the idea of a fork in a path at which point a decision must be made as to which branch to follow.

Now let's see whether we really need three operand addresses. In the scheme just described, the third address tells where to put the sum of the first two operands. But using a new memory cell is a luxury. The design could just as well force us to put the sum on top of one of the operands being summed. "On top of" doesn't give quite the right connotation. When a number is written to a specified memory cell, its former contents are lost, and we do not have to clear (set to zero) the destination address before overwriting it. The new instruction, +341843, would fit in a six-digit word and mean "Add the contents of cell 18 to the contents of cell 43 with the result going back into cell 43." Of course, on such a *two-address computer*, one of its operands is destroyed, and if we do not want to lose that operand, we make a copy of it before the add. The ability to copy the contents of one cell to another is always one of the primitively available commands in the machine's repertoire.

There are also one-address and even zero-address computers, but we have gone far enough here to establish the background for the general principle that the computer architect has considerable freedom in designing a computer and that we must learn the details of that design before we can write effective programs.

Precision Versus Accuracy

In a two-address computer that stores one word per cell, the memory would look something like that shown in Figure 1.3.

	Address	Content
	00	+341843
	01	+654039
	02	-872021
PC →	03	+204783
	04	+308971
	.	.
	.	.
	.	.
	99	+565700

Figure 1.3 A 100-Cell Six-Digit-Word Memory

If we accept the efficacy of the two-address scheme, we can program this computer almost as easily as if it were a three-address computer; yet something has been lost. When the contents of a given cell is taken to be a data value, that six-digit value is intrinsically less precise than is its earlier eight-digit counterpart. Does this mean that a computer of this design is inherently less accurate? That's another matter. *Accuracy* is freedom from error. As long as we don't try to add numbers whose sum exceeds 999,999, a computer of this design can be perfectly accurate. But the available *precision*—six digits rather than the former eight or ten—certainly limits at-

tainable accuracy because we can no longer add two six-digit numbers with the assurance that their sum will fit into one of our cells. There are ways to program around such limitations to precision, but their elaboration will be deferred until later in this book.

Running a Program

We have stated that without additional information, it is impossible to tell whether the contents of a given computer word is to be processed as a data value or interpreted as an instruction. But what is that additional information? The question is equivalent to our asking how the control unit "decides" to send a word to the arithmetic and logic unit or to the instruction decoding unit. The method used is quite simple.

Note that in Figure 1.3, the symbol PC is pointing to a particular one of the computer's 100 words, cell 3. PC stands for *program counter* and is itself a little piece of memory storage that is just big enough to hold an address—two digits in this case. The PC is a special kind of memory cell called a *register* and is part of the control unit, not part of addressable main memory. Having the PC point to cell 3 is just a way of showing visually that the program counter currently contains the address 03. For a related reason, memory addresses are sometimes called *pointers*, because they point to some particular piece of information of current interest. In this case, the PC is saying to cell 3, "You are next," meaning that its contents—+204783—is to be sent to the instruction decoding unit. Just after doing so, but before actual decoding, the PC is advanced to point to the next cell, 4 in this case. If the instruction at cell 3 does not cause a branch (a change in value of the PC), the next instruction to be executed will be taken from cell 4. This process can be summarized as follows:

```
┌──> Fetch the instruction whose address is in the PC
│    Increment the PC so it points at the next instruction
│    Decode the instruction fetched in step 1
│    Execute that instruction
└──  Loop back to step 1
```

Once started, the computer continues to execute this loop indefinitely until someone pulls the plug (or until one of the decoded instructions says halt). The instruction decoded may be a branch instruction that changes the PC and thus overrules the incrementation by 1 at step 2, but the computer doesn't care; it just goes back to step 1 and takes its next instruction from wherever the current value of the PC indicates.

To understand machine language programming in further detail, we need to refer to a particular computer design and its associated instruction repertoire. That is the purpose of the next section.