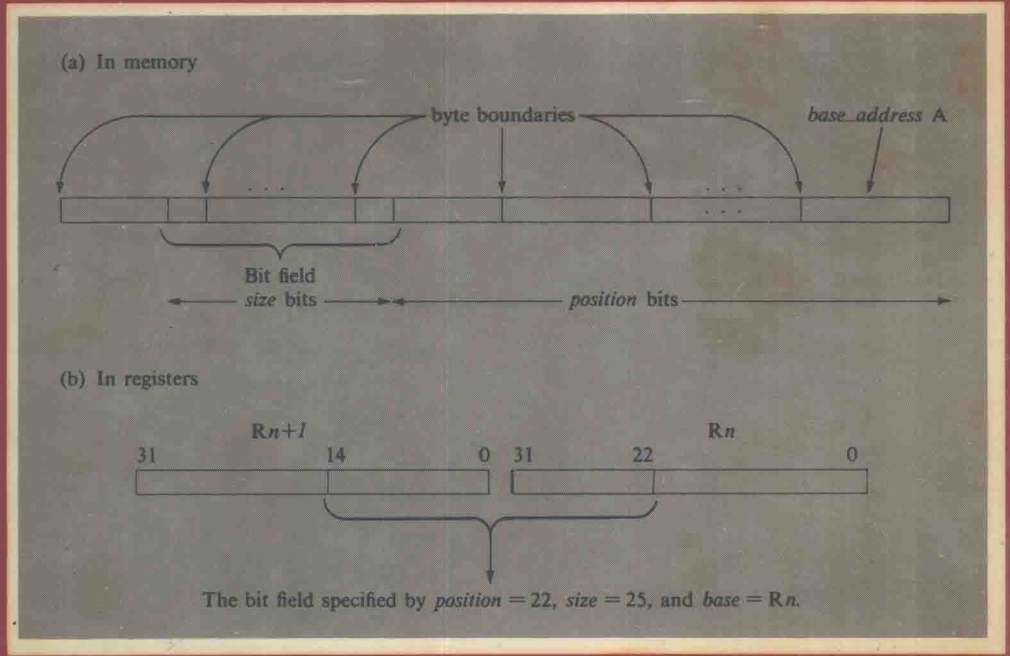
VAX-11 Assembly Language Programming



Sara Baase

VAX-11 Assembly Language Programming

Sara Baase

San Diego State University

PRENTICE–HALL, INC., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Baase, Sara.
VAX-11 assembly language programming.

Includes index.
1. VAX-11 (Computer)—Programming. 2. Assembler language (Computer program language) I. Title.
II. Title: V A X-eleven assembly language programming.
QA76.8.V37B3 1983 001.64'2 82–18101
ISBN 0–13–940957–2

Editorial/production supervision and interior design: Linda Mihatov
Cover design: Edsal Enterprises
Manufacturing buyer: Gordon Osbourne

To all the students I've enjoyed having in my classes

Prentice–Hall Software Series
Brian Kernighan, Advisor

Printed in the United States of America

10 9 8 7 6

ISBN 0-13-940957-2

Prentice–Hall International, Inc., *London*
Prentice–Hall of Australia Pty. Limited, *Sydney*
Editora Prentice–Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice–Hall Canada Inc., *Toronto*
Prentice–Hall of India Private Limited, *New Delhi*
Prentice–Hall of Japan, Inc., *Tokyo*
Prentice–Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

Preface

This book is intended as a text for an assembly language course such as CS 3: Introduction to Computer Science in the ACM's Curriculum '78. It is also for anyone who wants to learn about the instruction set and assembly language for the VAX-11. The book is written primarily for readers who do not already know any assembly language; the VAX-11 is used partly as a vehicle for teaching about features and principles common to many large modern computers and assembly languages. The book should be suitable for readers who are already familiar with assembly language if some sections are skimmed. The summary sections at the ends of the chapters include reference tables that should be helpful to both the novice and those experienced in assembly language who want to learn about the VAX. It is assumed that the reader is familiar with a high-level language.

The book has been organized to make the chapters easy to cover in sequence in a class where the students begin programming early and write programs regularly on the material as it is encountered. With this aim in mind, I introduce some topics and instructions informally as needed before they are covered in complete detail, and I intersperse chapters on topics that would be the subject of programming assignments (e.g., branching, procedures) with chapters on topics that would not be (e.g., machine code, assembler expressions). Topics are not divided up in precise, logically distinct chunks as they are in manuals. For example, although loop control instructions "belong" in Chapter 7, "Branching and Looping," one loop instruction is introduced early in Chapter 6 so that students can write a nontrivial program.

In our one-semester course we have covered almost all of the text. A few addressing modes (Chapter 8), packed decimal instructions (Chapter 13), Chapter 14, and some of the examples were skipped. (Some topics were skimmed.)

Chapters 1 and 2 are very short introductory chapters and should be covered quickly. Chapter 3 on hexadecimal numbers and two's complement representation may be skimmed if that material has been covered in an earlier course.

Chapter 4 begins the presentation of machine instructions and assembly language. The VAX has a very large set of operand addressing modes. Some of the simpler

ones are described in Chapter 4, while some more complex and less commonly available ones are left for Chapter 8.

To allow students to do simple I/O at a terminal, I have defined some I/O macros that can be used quite easily. The VAX Run-Time Library contains procedures that do terminal I/O, but the macros are simpler for students to use. The macros are READLINE, READRCRD, PRINTCHRS, and DUMPLONG. Their functional characteristics and argument formats are described in Chapter 5, and the macro definitions are given in Appendix D.

Students should be able to run simple programs after Chapter 5 and programs with loops after Chapter 6. While Chapters 1 through 5 are being covered in class, the students may be doing an assignment to gain familiarity with their timesharing system and editor. If the first few program assignments use primarily character data and if hex output from DUMPLONG is acceptable for a small amount of numeric output, the somewhat complex conversion between two's complement and character code for input and output, covered in Section 6.5, may be delayed in order to cover sooner some material on conditional branching from Chapter 7.

In some ways Chapter 8, "Machine Code Formats, Translation, and Execution," is the most important chapter, even though it has little to do with programming. Here, probably for the first time, students will begin to see how a computer actually executes instructions and how assembly language statements are translated.

Chapter 8 follows the chapter on branching and looping so that students can begin writing nontrivial programs early in a course. However, since they will see machine code on their program listings and will need to understand a little about instruction formats and execution to interpret and correct errors, it would be a good idea for the instructor to present some material from Chapter 8 (particularly Section 8.8, which explains some of the execution-time error messages) before completing Chapter 7. This is what we do in class, but it seemed awkward to break these chapters up into smaller, intermingled pieces.

Chapter 9 considers the problems involved in communicating between procedures and the programs that call them. It describes several techniques used for solving those problems but focuses mainly on the VAX procedure calling standard. A short section is included on the VAX conventions for linking assembly language with other languages. The information in this section should suffice for many straightforward situations, but it does not cover all argument types.

Chapter 10 presents more about the assembler (and linker): mainly the treatment of expressions and the distinction between absolute and relocatable expressions.

The VAX macro facility is not a particularly powerful or elegant one, but it does include many of the standard features such as argument concatenation, local labels, and a variety of conditional assembly directives. Some of these features and some general points about writing macros are presented in Chapter 11.

Chapter 12 presents the bit and logical instructions and includes a section on the VAX variable length bit field data type and instructions.

The floating point and packed decimal data types and instructions are presented in Chapter 13. The accuracy problems in floating point computations are illustrated.

Chapter 14 describes the character string manipulation instructions, including search, translate, and edit instructions.

Chapter 15 is a brief introduction to the VAX-11 Record Management Services. It will enable the reader to do some I/O without using the macros presented in Chapter 5. A more detailed discussion of I/O devices and operations would involve a lot of discussion of operating systems, so I have chosen not to include that in this book.

The book does not describe all the instructions in the VAX instruction set or all the assembler directives. Some instructions (e.g., quadword arithmetic) are introduced only in the exercises. The VAX instruction set includes some powerful and unusual machine instructions that are designed specifically to efficiently implement some high-level language constructs (e.g., CASE). These are not covered. (All machine instructions are included in the instruction table in Appendix A.) It is expected that students will be able to intelligently consult the relevant manuals to look up additional instructions and directives. The two main reference manuals are:

VAX Architecture Handbook
VAX-11 MACRO Language Reference Manual

The following manuals may also be helpful.

VAX-11 MACRO User's Guide
VAX-11 Linker Reference Manual
VAX-11 Command Language User's Guide
VAX-11 Symbolic Debugger Reference Manual
VAX-11 Record Management Services Reference Manual

This book contains several hundred exercises, ranging from short answer questions to problems that are suitable for programming assignments. Appendix E contains answers to some of the exercises from each chapter.

The type style in which the programming examples in this book were set uses the same symbol for the capital letter "oh" and for the digit "zero." It should be clear from the context which is meant.

Several people helped me, in small and large ways, in the preparation of this book. I would like to thank my colleague Richard Hager for suggesting the idea of writing the book, the many students who gave me lists of typos and errors in the manuscript when it was being used as the text for our assembly language course, instructors Tom Teegarden and John van Zandt for using the manuscript in their classes, and Jack Revelle for writing the editing and formatting software I used to prepare the manuscript, for the use of his computer, and for advice and suggestions throughout the project. Though their contribution to this project was for the most part indirect, several of my colleagues in the computer science group at San Diego State helped out by being such a good bunch of people to work with, and I thank them.

Sara Baase

Contents

Chapter 1

Introduction

1.1 WHAT IS ASSEMBLY LANGUAGE AND WHY STUDY IT?

An assembly language is a programming language in which instructions correspond closely to the individual primitive operations that are carried out by a particular computer. These primitive operations, encoded in a form the computer can act on directly, comprise the machine language of the computer. These are partial, informal descriptions, not rigorous ones. Along with the following discussion and examples, they are intended to give the reader a general idea of what assembly language "looks like," what kinds of instructions it has, and how it differs from high-level languages and machine language. In this book we will be studying the assembly language of the VAX-11 made by Digital Equipment Corporation (DEC for short). Each example shows a statement in a high-level language and possible translations of it into the VAX assembly language (called VAX-11 MACRO) and VAX machine language.

Perhaps the most glaring difference among the three types of languages is that as we move from high-level languages to lower levels, the code gets harder to read (with understanding). The major advantages of high-level languages are that they are easy to read and are machine independent. The instructions are written in a combination of English and ordinary mathematical notation, and programs can be run with minor, if any, changes on different computers. Each computer has its compiler to translate high-level language programs into its machine language.

Some parts of the assembly language instructions are decipherable: in Example 1.1 the variable names appear, and one may guess from their names what some of

EXAMPLE 1.1

FORTRAN

```
COST = BASE + NUM*VAR
```

ASSEMBLY LANGUAGE		*MACHINE CODE*
COST:	.BLKF 1	
BASE:	.BLKF 1	
VAR:	.BLKF 1	
NUM:	.BLKL 1	
	.	
	.	
	.	
CVTLF	NUM,R3	53DBAF4E
MULF2	VAR,R3	53D3AF44
ADDF3	BASE,R3,COST	C4AF53CBAF41

EXAMPLE 1.2

PASCAL

```
IF DIR<0 THEN SUM := SUM + AMNT[I]
```

ASSEMBLY LANGUAGE		*MACHINE CODE*
	TSTB (R2)	6295
	BGEQ NEXT	0418
	ADDL2 (R4)[R5],R6	566445C0
NEXT:	<next instruction>	

the instructions do. MULF2 does multiplication and ADDF3 does addition. The machine code is totally unintelligible without further explanation, and one can see that even knowing all the translation rules wouldn't make reading or writing in machine language an easy or enjoyable task.

The second most visible difference among the different types of languages is that several lines of assembly language instructions are needed to encode one line of a high-level language program. Early computers had very few instructions. The instruction sets included such operations as integer addition and subtraction, sign tests, branching, certain logical operations, input and output, and movement of data. Each instruction performed one operation. (Many modern microprocessors have similarly limited instruction sets.) Integer multiplication and division and floating-point arithmetic had to be programmed using the primitive operations. Now large computers have machine instructions that do integer multiplication and division, floating-point arithmetic, and many more complex logical and data-manipulation operations.

Execution of the Fortran statement in Example 1.1 requires several operations: conversion of the integer datum NUM to floating-point, a multiplication, an addition,

and an assignment of the result to COST. In many modern computers the conversion would require a long sequence of instructions, and each of the other operations would be performed by a separate instruction. The VAX has a very powerful set of instructions, though; the conversion is done in one instruction (CVTLF), and the add instruction adds and stores the result. In the second example, the Pascal **if** statement is translated to a sign test (TSTB), a conditional branch (BGEQ, branch if greater than or equal zero), and an addition. On other computers more instructions might be required to reference the array AMNT, but the VAX has very powerful and flexible ways to refer to data.

The machine code for a whole program segment would be one continuous sequence in the computer's memory; it is broken up into separate lines in each example to show the correspondence between sections of the machine code and the assembly language instructions. Actually, machine code is slightly worse than shown here; it consists of a long string of bits—i.e., 0's and 1's. What we see in the examples is a shorthand notation; each digit and letter represents a group of four bits. The notation used is the hexadecimal number system. Hexadecimal numbers are used extensively for machine code, memory addresses, and data, and will be discussed in Chapter 3.

The assembly and machine language sequences in the examples are not the only possible translations of the Fortran and Pascal statements shown. They depend on some assumptions about the context of the statements. In Example 1.1, for instance, the first four lines allocate memory space for the variables, but similar lines do not appear in Example 1.2. In assembly language, all variable names must be defined; i.e., roughly speaking, a position in memory must be assigned to them. In many cases it is possible to refer to variables by name as in Example 1.1 and in high-level languages. For Example 1.2, we assumed that the statement is in a subroutine or procedure, and DIR, SUM, and AMNT are arguments. Space must be allocated elsewhere for the data the routine acts on, but the variable names (if any) are not available for use here. We may not even use formal argument (i.e., dummy argument) names as in high-level language procedures or subroutines.

The machine code contains no declarations, variable names, or statement labels (like NEXT in Example 1.2). It contains only executable instructions; references to data and instructions are encoded in ways to be described in Chapter 8.

In situations where programming in a high-level language is not appropriate, it is clear that assembly language is to be preferred to machine language. Assembly language has a number of advantages over machine code aside from the obvious increase in readability. One is that the use of symbolic names for data and instruction labels frees the programmer from computing and recomputing the memory locations whenever a change is made in a program. Another is that assembly languages generally have a feature, called macros, that frees the user from having to repeat similar sections of code used in several places in a program. Assemblers do many bookkeeping and other tasks for the user. Often compilers translate into assembly language rather than machine code.

If one has a choice between assembly language and a high-level language, why choose assembly language? The fact that the amount of programming done in assembly

language is quite small compared to the amount done in high-level languages indicates that one generally doesn't choose assembly language. However, there are situations where it may not be convenient, efficient, or possible to write programs in high-level languages. The first compiler, for example, could not be written in the high-level language it translates because there would be no way to run the compiler. (Actually, nowadays, compilers are usually written in high-level languages.) Programs to control and communicate with peripheral devices (input and output devices) are usually written in assembly language because they use special instructions that are not available in high-level languages, and they must be very efficient. Some systems programs are written in assembly language for similar reasons. In general, since high-level languages are designed without the features of a particular machine in mind and a compiler must do its job in a standardized way to accommodate all valid programs, there are situations where to take advantage of special features of a machine, to program some details that are inaccessible from a high-level language, or perhaps to increase the efficiency of a program,[1] one may reasonably choose to write in assembly language.

Although assembly language programs must be translated into machine code, the translation task is simpler than for a high-level language because of the close correspondence between the assembly and machine language for a particular computer. A program that translates assembly language into machine language is called an *assembler.*

Some of the major reasons for studying assembly language have little to do with its practical use as a programming language. Consider that Fortran was developed in the 1950s when computers were made of vacuum tubes. The same Fortran program that ran on such a machine could also run on a computer in the 1960s made of transistors, on a computer today with integrated circuits, and on a future computer that uses some new technology. Clearly, learning Fortran (and other high-level languages) teaches one virtually nothing about what a computer is and how it actually works. We won't be studying computer hardware here, but the point—that there have been dramatic changes over the years, all virtually invisible to high-level language programmers—is equally true about computer architecture, that is, the conceptual structure and functional characteristics of a computer. A major purpose of studying assembly language is to learn something about computer architecture; in fact, part of Digital Equipment Corporation's definition of architecture is "the attributes of a system as seen by the assembly language programmer." Thus, along with learning how to write programs in assembly language, we will study the structure of the computer, its instruction set, how it decodes and executes instructions, how data are represented, what schemes are used to reference memory locations, and how arguments are passed to and from procedures, or subroutines. All these topics and others covered in assembly language texts help us understand how the computer really works and indirectly help us understand more about the task performed by the compilers that must translate high-level language programs.

[1] Optimizing compilers may eliminate this last reason, as some very good ones produce code that rivals for efficiency the work of experienced assembly language programmers.

Computer architecture and assembly languages differ very much from what they were thirty years ago. There are also significant differences among computers available today from different manufacturers, but of course they also have many features and characteristics in common. Why study the VAX assembly language rather than some other one? To be honest, the choice of what assembly language to learn is usually determined by what computer is available at one's university or place of work, and the decision to acquire that particular machine probably depended on many factors having little to do with the merits of its assembly language. If a programmer must use a VAX and intends to write in assembly language, then he or she will probably write in VAX-11 MACRO.[2] On the other hand, if one's purpose is to learn about the architecture of a modern computer, then there are several to choose from that would serve the purpose. The VAX is not the only choice, but it is a very good one. As we indicated in the examples, the VAX has some instructions and ways of accessing data that are more powerful and flexible than those of other computers. The VAX and its assembly language also have many features that are typical of modern machines. It has, as computer people would put it, a nice architecture. Throughout this book we will often mention similarities and differences between the VAX and other computers.

1.2 SOME TERMINOLOGY

In this section we give definitions and brief explanations of some commonly used terms, many of which should be somewhat familiar to the reader. This is not intended as a complete glossary, but rather as a review of some general terminology.

Data are pieces of information of some kind, often numbers or character strings. The singular form of data is *datum.*

A *bit* is often defined as a binary digit, i.e., a 0 or a 1. It also may mean a place (in a computer memory, for example) where a 0 or a 1 may be stored. In many computers, including the VAX, bits are organized in groups of eight called *bytes.* In such machines the byte is considered the basic unit of memory. Half a byte is defined by some computer makers, including DEC, as a *nibble.* (Yes, computer scientists have a sense of humor.) To *complement* a bit means to reverse its value, i.e., to change a 0 to a 1 and change a 1 to a 0. To complement a bit string means to complement each bit in the string.

A *compiler* is a program that translates a high-level language into assembly language. Compilers usually provide a program listing and error messages for syntax errors in the program being translated.

An *assembler* is also a translation program; its purpose is to translate assembly language into machine language. The input to a compiler or assembler is a program written by a programmer; it is called a *source program* (or *source file* or *source module*). The primary output from the compiler or assembler is called an *object*

[2] PDP-11 assembly language programs may be run on the VAX in what is called compatibility mode, but we will not consider that here.

module (or *object file*). Like a compiler, an assembler also provides error messages and a program listing. The listing shows the machine code produced by the assembler.

Because a program may consist of several modules or procedures assembled separately, and because the assembler doesn't know where in memory a program will be when it runs, the object file is not the final machine-code translation. Another program, called a *linker,* combines the various object modules and puts them in executable form. The output of the linker is called an executable image or execution file.

An *executable instruction,* in assembly language or a high-level language, is an instruction that gets translated into one or more machine language instructions. Other instructions appearing in a program, such as declarations and header statements, provide information or instructions to the assembler or compiler.

As suggested above, a program goes through three stages: assembly, linkage, and execution. When learning and programming assembly language, it is often important to understand which operations take place in which of these stages. Thus we will talk about an operation, or perhaps an error, occurring at *assembly time* or at *execution time. Assembly time* does not mean the amount of time used to translate the program, but the time span, or stage, when assembly takes place. The same is true for *execution time.*

A procedure is a program section that performs a particular task on (zero or more) arguments that are given to it when it is called by another routine. A procedure can be assembled (or compiled) as an independent unit. (In Fortran, procedures are called subroutines.)

We have used the term *module* several times. Since it is used in many contexts, its definition is fairly vague. A *module* is a section of a program, as an abstract entity or in some representation such as source or object code, treated as a unit for some purpose. When we use the term, we will most often mean a section of assembly language source code assembled as a unit. A module would generally be either a main program or a procedure.

Multiprogramming means concurrent execution of several programs. The computer's CPU (central processing unit) executes only one instruction at a time, but it is so fast that, to avoid wasting this valuable resource, several programs are kept available in memory at once so that while one is waiting for a relatively slow operation (maybe one that takes a few dozen milliseconds) to finish, another program may be executing. The slow operation may involve reading data from a disk, or something really slow: a person sitting at a terminal deciding what to do next.

An *operating system* is a collection of programs that controls and allocates the resources of the computer system. It handles the scheduling of all the jobs in the system, the communications necessary for doing input and output, and record and file management. The standard VAX-11 operating system is called VAX/VMS. The letters stand for Virtual Address eXtensions/Virtual Memory System.

Chapter 2

Machine Organization

A computer system generally consists of three subsystems (as illustrated in Fig. 2.1): the memory, the central processing unit, and the I/O subsystem. In this chapter we present an overview of their architecture, or logical structure.

2.1 MEMORY AND DATA ORGANIZATION

Physical Memory

The *physical memory,* also called *main memory* or *main storage,* of a computer is where instructions and data that the processor can directly fetch and execute or manipulate are stored. The VAX uses MOS (metal oxide semiconductor) memory, which consists of chips containing thousands of tiny electric circuits each of which may be open or closed at any time. One state is taken to represent 0 and the other to represent 1. In the past, computer memories have been made of other materials, including, for example, magnetic rings, called cores, that could be magnetized in one of two directions, and thus also could represent one bit. In the VAX (and many other computers) the bits are logically grouped in units of eight called *bytes.*

On the VAX-11/780 it takes an average of a little under 300 nanoseconds to transfer an operand from memory to the central processor. On the VAX-11/750 it takes about 400 nanoseconds. (One nanosecond is 10^{-9} second.) To achieve access times this low, both the 780 and the 750 use a special high-speed memory, called a *cache,* where they store what they are currently working on.

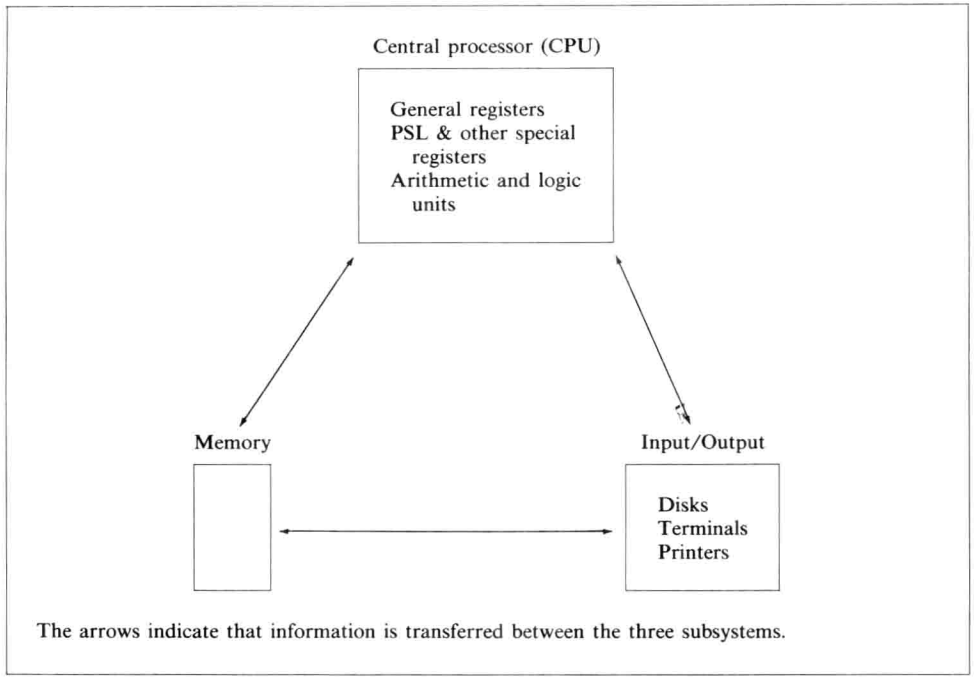


Figure 2.1 The subsystems of a computer

A VAX-11/780 may have up to eight megabytes (roughly eight million bytes) of main memory; a 750 may have up to two megabytes.

The Logical Structure of Memory

Conceptually, memory consists of a sequence of bytes numbered from 0 to the maximum available in the given installation. The number of a byte is called its *address.*

The bits in a byte are numbered right to left, beginning with 0. Thus the rightmost, or least significant, bit is bit number 0 and the leftmost, or most significant, bit is bit number 7. The bits are numbered right to left so that the bit number is the power of 2 represented by that bit when the datum is interpreted as a binary number.

Since each bit may have one of two values, 0 or 1, 256 (2^8) configurations are possible in one byte. A byte may be used to store a character or a small integer. (Characters are encoded in seven bits; the eighth, or leftmost, is always zero.)

Bytes are too small a unit of memory for storing large integers or floating-point numbers. They are grouped in various ways to provide larger units of storage