1

Introduction to Assembly Language

INTRODUCING ASSEMBLY LANGUAGE

Assembly language unlocks the secrets of your computer's hardware and software. It teaches you about the way the computer's hardware and operating system work together and how application programs communicate with the operating system.

To learn how a computer and its software really work, you need to view them at the machine level. This book is about assembly language programming on the IBM-PC, the PC/XT, and the PC/AT, as the title suggests. But the book is also about operating system concepts.

There is no single assembly language. Each computer or family of computers uses a different set of machine instructions and a different assembly language. Strictly speaking, IBM-PC assembly language consists only of the Intel 8086/8088 instruction set—with enhancements for the 80186, 80826, and 80386. The instructions for the Intel 8088 may be used without modification on all the processors previously listed.

An *assembler* is a program that converts source-code programs into machine language. Several excellent assemblers for the IBM-PC are available that run under the disk operating systems MS-DOS or PC-DOS. Two examples are the

Microsoft Macro Assembler, called MASM, and Borland's Turbo Assembler. The Microsoft assembler, on which this book is based, recognizes 50 or more *directives,* commands that control the way a program is assembled. Most programmers refer to IBM-PC assembly language as both the 8086/8088 instruction set and the complete set of Microsoft Macro Assembler directives. In this book we will do the same. The programs in this book will also work with Borland's Turbo Assembler.

Why Learn Assembly Language?

People learn assembly language for various reasons. The most obvious one may be to learn about the computer's architecture and operating system. You may want to learn more about the computer you work with and about the way computer languages generate machine code. Because of assembly language's close relationship to machine language, it is closely tied to the computer's hardware and software.

You may also want to learn assembly language for its *utility*. Certain types of programming are difficult or impossible to do in high-level languages. For example, direct communication with the computer's operating system may be necessary. A high-speed color graphics program may have to be written using a minimum of memory space. A special program may be needed to interface a printer to a computer. Perhaps you will need to write a telecommunications program for the IBM-PC. Clearly, the list of assembly language applications is endless.

Often there is a need to remove restrictions. High-level languages, out of necessity, impose rules about what is allowed in a program. For example, Pascal does not allow a character value to be assigned to an integer variable. This makes good sense *unless* there is a specific need to do just that. An experienced programmer will find a way around this restriction, but in doing so may end up writing code that is less portable to other computer systems and is difficult to read. Assembly language, in contrast, has very few restrictions or rules; nearly everything is left to the discretion of the programmer. The price for such freedom is the need to handle many details that would otherwise be taken care of by the programming language itself.

Assembly language's usefulness as a *learning tool* should not be underestimated. By having such intimate contact with the operating system, assembly language programmers come to know instinctively how the operating system works. Coupled with a knowledge of hardware and data storage, they gain a tremendous advantage when tackling unusual programming problems. They are able to look at a problem from a different viewpoint than the programmer who knows only high-level languages.

Assembly Language Applications

At first, the assembly language programs presented in this book will seem almost trivial. Those new to assembly language often cannot believe the amount of work required to perform relatively simple tasks. The language requires a great deal

of attention to detail. Most programmers don't write large application programs in assembly language; instead they write short, specific routines.

Often we write subroutines in assembly language and call them from high-level language programs. You can take advantage of the strengths of high-level languages by using them to write applications. Then you can write assembly language subroutines to handle operations that are not available in the high-level language.

Suppose you are writing a business application program in COBOL for the IBM-PC. You then discover that you need to check the free space on the disk, create a subdirectory, write-protect a file, and create overlapping windows, all from within the program. Assuming that your COBOL compiler does not do all of this, you can then write assembly language subroutines to handle these tasks.

Let's use another example. You might be writing a word processor in C or Pascal but find that this language performs badly when updating the screen display. If you know how, you can write routines in assembly language to speed up critical parts of the application and allow the program to perform up to professional standards.

Large application programs written purely in assembly language, however, are somewhat beyond the scope of the person who has just finished this book. There are many people who write complete assembly language application programs for the IBM-PC. The few programmers I know in this group are familiar with several machine architectures and assemblers and have been programming professionally for at least several years. These fortunate individuals still had to start with a basic foundation, and this book is intended to help you acquire just that.

COMPUTER NUMBERING SYSTEMS

The explanation of computer numbering systems in this chapter is brief and is not meant to be a complete tutorial. For a much longer explanation of the process of converting numbers from one format to another, see Appendix A after reading this chapter.

There are four primary types of numbering systems used by programmers today: *binary, octal, decimal,* and *hexadecimal.* Of these, the octal system is rarely used on the IBM-PC. Each system has a *base,* or maximum number of values, that can be assigned to a single digit:

Base	Possible Digits
Binary	0 1
Octal	0 1 2 3 4 5 6 7
Decimal	0 1 2 3 4 5 6 7 8 9
Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F

In the hexadecimal (base 16) numbering system, the letters *A* through *F* represent the decimal values 10 through 15. We will assume that all numbers are decimal unless stated otherwise.

Radix. When referring to binary, octal, and hexadecimal numbers, a single lowercase letter called a *radix* will be appended to the end of each number to identify its type. For example, hexadecimal 45 will be written as 45h, 76 octal will be 76o or 76q, and binary 11010011 will appear as 11010011b.

Binary Numbers

A computer stores both instructions and data as individual electronic charges. Representing these entities with numbers requires a system geared to the concepts of *on* and *off* or *true* and *false*. Binary is a base 2 numbering system in which each digit is either a 0 or a 1. The digit **1** indicates that the current is on, and the digit **0** indicates that the current is off:

1	0	1	0
on	off	on	off

or

1	0	1	0
true	false	true	false

At one time, computers actually had panels full of mechanical switches that were flipped by hand. Electromechanical relays were soon used instead, and later transistors were implemented. Eventually thousands of individual electronic switches and circuits were engraved on tiny microprocessor chips. Each binary digit (1 or 0) represents an on or off state in an electronic switch.

Computers store all instructions and data as sequences of binary digits, without any distinction between the two. For example, the first three letters of the alphabet would be stored in the IBM-PC as

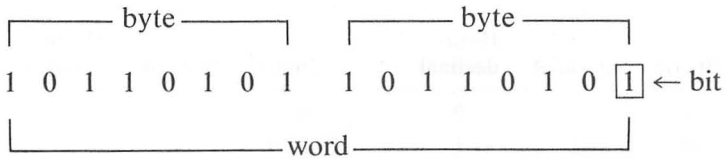
010000010100001001000011 = "ABC"

At the same time, an instruction to add two numbers might be stored in memory as

0000010000000101

Bits and Bytes. Each digit in a binary number is called a *bit*. Eight of these make up a *byte*, which is the basic unit of storage on nearly all computers. Each lo-

cation in the computer's memory holds exactly one byte, or 8 bits. On the IBM-PC and all compatible computers, a byte can hold a single instruction, a character, or a number. The next largest storage type is a *word,* which is 16 bits, or 2 bytes, long:

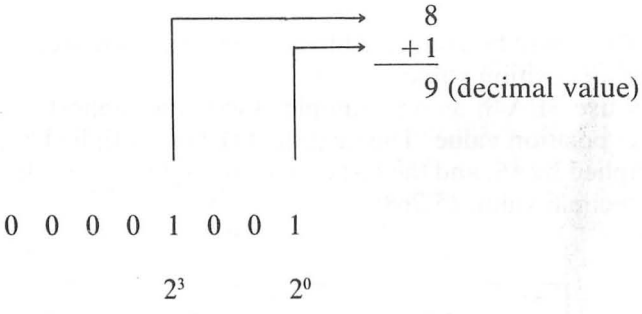


The IBM-PC is called a *16-bit* computer because its instructions can operate on 16-bit quantities.

Converting Binary to Decimal. There are many occasions when we need to find the decimal value of a binary number. Each bit position in a binary number is a power of 2, as the following illustration shows:

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
value:	128	64	32	16	8	4	2	1

In order to find the decimal value, we add the value of each bit that equals 1 to the number's total value. Let's try this with the binary number 0 0 0 0 1 0 0 1:



Hexadecimal Numbers

Large binary numbers are cumbersome to read, so *hexadecimal* numbers are usually used to represent computer memory or instructions. Each digit in a hexadecimal number represents 4 binary bits, and 2 hexadecimal digits represent a byte. In the following example, the binary number 000101100000011110010100 is represented by the hexadecimal number 160794:

```
  1 6      0 7      9 4
00010110 00000111 10010100 = 160794h
```

A single hexadecimal digit may have a value from 0 to 15, so the letters *A* to *F* are used as well as the digits 0 to 9. The letter *A* = 10, *B* = 11, *C* = 12, *D* = 13, *E* = 14, and *F* = 15. The following table shows how each sequence of 4 binary bits translates into a decimal or hexadecimal value:

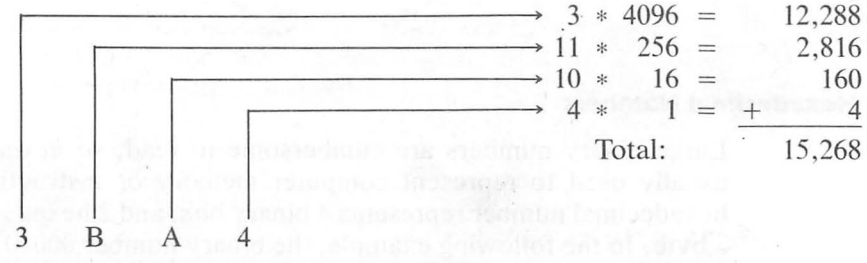
Binary	Decimal	Hexa-decimal	Binary	Decimal	Hexa-decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Hexadecimal Digit Positions. Each hexadecimal digit position represents a power of 16. This is helpful in calculating the value of a hexadecimal number:

	16^3	16^2	16^1	16^0
value:	4096	256	16	1

Numbers may be converted from hexadecimal to decimal by multiplying each digit by its position value.

Let's use 3BA4h as an example. First, the highest digit (3) is multiplied by 4096, its position value. The next digit (B) is multiplied by 256, the next digit (A) is multiplied by 16, and the last digit is multiplied by 1. The sum of these products is the decimal value 15,268:



(When performing the multiplication, recall that the hexadecimal digit *B* = 11 and the digit *A* = 10.)

Signed Numbers

Binary numbers may be either *signed* or *unsigned*. Oddly, the central processing unit (CPU) performs arithmetic and comparison operations for both types equally well, without knowing which type it's operating on.

An *unsigned* byte uses all 8 bits for its numeric value. For example, 11111111 = 255. Therefore, 255 is the largest value that may be stored in an unsigned byte. The largest 16-bit value that may be stored in an unsigned word is 65,535.

A *signed* byte uses only 7 bits for its value; the highest bit is reserved for the sign. The number may be either positive or negative: If the sign bit equals 1, the number is negative; otherwise, the number is positive:

```
(sign bit)
    |
    1 0 0 0 1 0 1 0    (negative number)
    0 0 0 0 1 0 1 0    (positive number)
```

To calculate the *ones complement* of a number, reverse all of its bits. The ones complement of 11110000b, for example, is 00001111b.

Negative numbers are stored in a special format called a *twos complement*. In order to find out what the number's value really is, you have to calculate its twos complement.

To find the twos complement, calculate the ones complement and add 1. Let's use the negative number 11111010 as an example. Its ones complement is 00000101; if we add 1 to this, we end up with 6:

```
  1 1 1 1 1 0 1 0
  0 0 0 0 0 1 0 1   ←ones complement
+               1   ←add 1
-----------------
  0 0 0 0 0 1 1 0   ←twos complement (6)
```

Now we append the negative sign to the decimal result and see that the original number (11111010) was equal to -6.

Maximum and Minimum Signed Values. The maximum signed byte value is +127 (01111111b, or 7Fh). The maximum signed word value is +32767 (0111111111111111b, or 7FFFh). The minimum signed byte and word values are -128 and -32768, respectively.

CHARACTER STORAGE

Computers can store only binary numbers, so how are characters such as "A" and "$" stored? A standard system used for translating characters into numbers is the *American Standard Code for Information Interchange* (*ASCII*). Another

system, the *Extended Binary Code for Decimal Interchange* (*EBCDIC*), is used on IBM mini and mainframe computers. In each case, a unique numeric value is assigned to each character, including control characters used when printing or transmitting data between computers. ASCII codes are used on nearly all microcomputers, including the IBM-PC and IBM PS/2 series.

A table of ASCII codes is listed on the back inside cover of this book. Standard ASCII codes are actually only 7-bit codes, so the highest possible value is 7Fh. The eighth bit is optional and is used on the IBM-PC to extend the character set. Values 80h–FFh represent graphics symbols and Greek characters. Values 0–1Fh are control codes for printer, communications, and screen output.

All characters, including numbers and letters, are assigned unique ASCII codes. For example, the codes for the letters *ABC* and the numbers 123 follow:

	A	*B*	*C*	1	2	3
ASCII code	41h	42h	43h	31h	32h	33h

Binary Storage. Each letter or digit takes up 1 byte of storage. When we store numbers, however, we can be more efficient. The numeric value 123 can be stored in a single binary byte: 01111011b, or 7Bh. A byte in memory could be a single-byte numeric value or the ASCII representation of a character. The following sequence of memory bytes, for example, could be ASCII characters or four separate binary values:

30h	31h	32h	33h

Differences between instructions and data are purely artificial. High-level languages impose restrictions on the way variables and instructions may be manipulated, but assembly language does not. We might think of these restrictions as *boundaries* designed to help us avoid making fundamental errors. In assembly language there are few boundaries, but we are shouldered with the responsibility of taking care of many details.

ASSEMBLY LANGUAGE: AN INTRODUCTION

Assembly language is a specific set of instructions for a particular computer system. It provides a direct correspondence between symbolic statements and machine language. An *assembler* is a program that translates a program written in assembly language into machine language, which may in turn be executed by the computer. Each type of computer has a different assembly language, because the computer's design influences the instructions it can execute.

Machine Instructions

A *machine instruction* is a binary code that has special meaning for a computer's CPU—it tells the CPU to perform a task. The task might be to move a number

from one location to another, compare two numbers, or add two numbers. Each machine instruction is precisely defined when the CPU is constructed, and it is specific to that type of CPU. The following list shows a few sample machine instructions for the IBM-PC:

```
0 0 0 0 0 1 0 0   Add a number to the AL register
1 0 0 0 0 0 0 1   Add a number to a variable
1 0 1 0 0 0 1 1   Move the AX register to another register
```

The *instruction set* is the entire body of machine instructions available for a single CPU, determined by its manufacturer. On IBM-PC and compatible computers, the CPU is one of the following, each made by Intel: iAPX 8088, iAPX 8086, iAPX 80286, or iAPX 80386. All of these CPUs have a common instruction set that will be used in this book. At the same time, the more advanced processors (80286 and 80386) have enhanced instructions that increase their flexibility.

A typical 2-byte IBM-PC machine instruction might be as follows: **B0 05**. The first byte is called the *operation code*, or *op code*, which identifies it as a MOV (move) instruction. The second byte (05) is called the *operand*. The complete instruction moves the number 5 to a register called AL. *Registers* are high-speed storage locations inside the CPU which are used by nearly every instruction. They are identified by two-letter names, such as AH, AL, AX, and so on. We refer to machine instructions using hexadecimal numbers, because they take up less space on the printed page.

Suppose we want to move the number 5 to a different register, the AH register. That machine instruction is **B4 05**. It has a different op code than our first example, but the operand is still the same.

Assembly Language Instructions

Although it is possible to program directly in machine language using numbers, assembly language makes the job easier. The assembly language instruction to move 5 to the AL register is

```
mov   al,5
```

Here it is understood that the first operand (AL) is the destination, and the number 5 is the source.

Assembly language is called a *low-level* language because it is close to machine language in structure and function. We can say that each assembly language instruction corresponds to one machine instruction. In contrast, *high-level* languages such as Pascal, BASIC, FORTRAN, and COBOL contain powerful statements that are translated into many machine instructions by the compiler.

Mnemonic. A *mnemonic* (pronounced ni-'män-ik) is a short alphabetic code that literally "assists the memory" in remembering a CPU instruction. It may be an *instruction* or a *directive*. An example of an instruction is MOV (move). An ex-

ample of an assembler directive is DB (define byte), used to create memory variables.

Operand. An instruction may contain zero, one, or two operands. An operand may be a register, a memory variable, or an immediate value. For example:

```
10      (immediate value or constant)
count   (memory variable)
AX      (register)
```

The choice of operand is usually determined by the addressing mode. The *addressing mode* tells the assembler where to find the data in each operand: in a register, in memory, or as immediate data. Examples of instructions using operands are:

```
push  ax          ; one register
mov   ax,bx       ; two registers
add   count,cl    ; memory variable, register
mov   bx,1000h    ; register, immediate value
```

Comment. The beginning of a comment is marked by a semicolon (;) character. All other characters to the right of the semicolon are ignored by the assembler. You can even begin a line with a semicolon; in this case, the entire line will be treated as a comment. Assembly language statements should contain clear comments in order to help explain details about the program. Samples are as follows:

```
; This entire line is a comment
mov ax,bx    ; copy the BX register into AX
```

Programming Tip: Using a Debugger

In this chapter we begin looking at short assembly language programs and show the contents of registers and memory when they run. The best way to learn about these programs and about the way the computer works is to use a programmer's tool called a *debugger*. A debugger is a program that allows you to examine registers and memory and to step through a program one statement at a time to see what is going on. In assembly language, you will depend upon this ability to see what the CPU is doing. There are a number of debuggers to choose from: DEBUG is a simple, easy-to-use debugger supplied with DOS. The SYMDEB utility is supplied with Microsoft MASM 4.0, but not with version 5.0. It has all the DEBUG commands, with a number of useful enhancements. The CODEVIEW debugger was supplied by Microsoft beginning with MASM version 5.0 and is a full-featured source-level debugger. For a complete tutorial on using DEBUG, see Appendix B.

A Sample Program

Assembly language programs are made up primarily of instructions and operands. *Instructions* tell the CPU to carry out an action, while *variables* are memory locations where data are stored. In assembly language jargon, variables are also known as *memory operands*. *Immediate operands* are constant values, like 5 and 10. Let's write a short assembly language program that adds three numbers together and stores them in a variable called **sum**. The lines are numbered for your convenience.

```
1:   mov   ax,5       ; move 5 into the AX register
2:   add   ax,10h     ; add 10h to the AX register
3:   add   ax,20h     ; add 20h to the AX register
4:   mov   sum,ax     ; store AX in sum
5:   int   20         ; end the program
```

Each line in our example begins with an instruction mnemonic (MOV or ADD), followed by two operands. On the right side, any text following a semicolon is treated as a comment. There is no standard rule regarding the use of uppercase or lowercase letters.

The sample program may be understood with a little imagination. The MOV instruction tells the CPU to move, or copy data, from a *source* operand to a *destination* operand. Line 1 moves 5 into the AX register. Line 2 adds 10h to the AX register, making it equal to 15h. Line 3 adds 20h to AX, making it equal to 35h, and line 4 copies AX into the variable called **sum**. The last line halts the program.

We could have performed the addition using a single high-level language statement, but that's not the point. By writing the program in assembly language, we have communicated with the computer on its own level, and we can see exactly how the computer works.

Assemble and Test the Program. If you have DEBUG available, you can assemble and test our sample program. You will find a debugger to be the perfect tool for learning how assembler instructions work. It prompts for commands using the hyphen (-) character. The DEBUG commands to assemble and test the program are as follows:

```
DEBUG                 (load the DEBUG.COM program)
-A 100                (begin assembly at location 100)
  mov ax,5            (enter the rest of the program)
  add ax,10
  add ax,20
  mov [0120],ax       (SUM is at location 0120)
int 20                (end program)
-                     (press ENTER to end assembly)
-R                    (display registers)
-T                    (trace each instruction)
-T
-T
-G                    (execute the rest of the program)
-Q                    (quit DEBUG, and return to DOS)
```

The **A** (Assemble) command allows you to input a program and have it assembled into machine instructions. After typing the A command, you will be prompted for each assembler instruction.

The **T** (Trace) command executes a single instruction, displays the registers, and stops. The **R** (Register) command displays the registers. The **G** (Go) command executes all remaining instructions in the program, and the **Q** (Quit) command returns you to DOS.

As you assemble each line of the program, you will see a computer address appear at the left side of the screen. These addresses show the location of each instruction. For example:

```
-A  100
5511:0100  mov  ax,5
5511:0103  add  ax,10
.
.
```

After the last MOV instruction, press ENTER to return to DEBUG's command mode. If you would like to see the machine instructions that make up your program, type U for *Unassemble* (this is often called *disassembling* a program). A modified listing of the statements is as follows:

Machine Instruction			Assembly Instruction	
B8	05	00	MOV	AX,0005
05	10	00	ADD	AX,0010
05	20	00	ADD	AX,0020
A3	20	01	MOV	[0120],AX
CD	20		INT	20

DEBUG always displays numbers in hexadecimal. The number [0120] in brackets is DEBUG's way of referring to the contents of a memory location. The variable **sum** is located at address 0120h.

In each machine instruction shown here, the first byte is the op code and the next 2 bytes represent an immediate or memory operand. Of course, not all assembler instructions are 3 bytes long; their lengths vary between 1 and 6 bytes.

Note that 16-bit values are reversed when stored in memory. For example, the first instruction in the preceding table moves 0005h to AX, and the machine instruction reverses the 05h and the 00h. This reversal of bytes is characteristic of all processors in the Intel family and is performed for efficiency reasons. When 16-bit registers are loaded from memory back into registers, the bytes are re-reversed to their original form.

Our entire program could have been created by entering the following sequence of bytes into memory:

B8 05 00 05 10 00 05 20 00 A3 20 01 CD 20

The actual binary storage of the program would be

```
10111000 00000101 00000000 00000101 00010000 00000000
00000101 00100000 00000000 10100011 00100000 00000001
11001101 00100000
```

Believe it or not, early computers had to be programmed just this way, using mechanical switches to represent each binary bit. Most computer programs, fortunately, were very short.

In this book, we will not write programs using binary numbers, because it takes too long to look up the op code for each instruction. But it does illustrate a point: Computer programs are nothing more than meaningful sequences of numbers.

Trace the Program. Use DEBUG's **T** (Trace) command to execute each program statement. After each instruction is executed, DEBUG stops and displays the CPU registers and the next instruction. One advantage to using this procedure is that you can halt the program and return to DOS at any time. You can also change the contents of registers and variables before going on.

A trace of our sample program appears in Figure 1–1. As you look at it, carefully note the value of the AX register as it is changed by each instruction. You are now looking at what the program is doing directly at the machine level. This close contact with the CPU will be necessary throughout your study of assembly language.

Figure 1-1 Sample program trace, using DEBUG.

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=DF12  BP=0000  SI=0000  DI=0000
DS=1FDD  ES=1FDD  SS=1FDD  CS=1FDD  IP=0100   NV UP EI PL NZ NA PO NC
1FDD:0100  B80500          MOV  AX,0005

-T
AX=0005  BX=0000  CX=0000  DX=0000  SP=DF12  BP=0000  SI=0000  DI=0000
DS=1FDD  ES=1FDD  SS=1FDD  CS=1FDD  IP=0103  NV UP EI PL NZ NA PO NC
1FDD:0103  051000          ADD  AX,0010

-T
AX=0015  BX=0000  CX=0000  DX=0000  SP=DF12  BP=0000  SI=0000  DI=0000
DS=1FDD  ES=1FDD  SS=1FDD  CS=1FDD  IP=0106  NV UP EI PL NZ NA PO NC
1FDD:0106  052000          ADD  AX,0020

-T
AX=0035  BX=0000  CX=0000  DX=0000  SP=DF12  BP=0000  SI=0000  DI=0000
DS=1FDD  ES=1FDD  SS=1FDD  CS=1FDD  IP=0109  NV UP EI PL NZ NA PE NC
1FDD:0109  A32001          MOV  [0120],AX

-G
-D 120,121                (dump memory locations 120 through 121)
1FDD:0120  35 00          (bytes are reversed)
```

```
                    MASM CHARACTER SET

Letters: A-Z, a-z
Digits: 0-9

Special characters:

?                                   ,       (comma)
@                                   "       (double quotes)
_   (Underline symbol)              &
$                                   %
:                                   !
.   (period)                        '       (apostrophe)
[ ]                                 ~       (tilde)
( )                                 |
< >                                 \
{ }                                 =
+                                   #
/                                   ^       (caret)
=                                   ;
                                    `       (accent char)
```

Figure 1-2 The Macro Assembler character set.

BASIC ELEMENTS OF ASSEMBLY LANGUAGE

In this section, we will elaborate on the basic building blocks of IBM-PC assembly language. Assembly language statements are made up of constants, literals, names, mnemonics, operands, and comments. Figure 1–2 shows the assembler's basic character set. These characters may be used to form numbers, names, statements, and comments.

Constant

A *constant* is a value that is either known or calculated at assembly time. A constant may be either a number or a string of characters. It cannot be changed at runtime. A *variable,* on the other hand, is a storage location that may be changed at runtime. The following are examples of constants:

```
'ABC'
2134
 5*6
(1+2)/3
```

Integer. An integer is made up of numeric digits with no decimal point, followed by an optional radix character (d = decimal, h = hexadecimal, o, q = octal, b

= binary). The radix character may be in uppercase or lowercase. If the radix is omitted, the number is assumed to be a decimal number. Examples:

Example	Radix
11110000b	Binary
200	Decimal (default)
300d	Decimal
4A6Bh	Hexadecimal
2047q	Octal
2047o	Octal

Real Number. A *real number* contains digits, a single decimal point, an optional exponent, and an optional leading sign. The syntax is

$$\left[\left\{\begin{matrix}+\\-\end{matrix}\right\}\right] \textit{digits.digits} \left[\text{E}\left[\left\{\begin{matrix}+\\-\end{matrix}\right\}\right] \textit{digits}\right]$$

Examples

```
2.3
+200.576E+05
0.22222E-5
-6.0e3
```

Syntax Notation. In the preceding example and in all future syntax definitions, an optional element will be enclosed in brackets. Braces will identify a required selection. For example, if a leading sign is used in a real number, you must choose either a plus (+) or minus (-) sign. Required reserved words are in uppercase. Lowercase words in italics are predefined terms, such as *identifier, operand,* and *register.*

Character or String Constant. A single ASCII character or a string of characters may be enclosed in single or double quotation marks:

```
"a"
'B'
"STACK OVERFLOW"
'012#?%%A'
```

A *character constant* is 1 byte long. The length of a *string constant* is determined by its number of characters. The following constant is 5 bytes long:

```
'ABCDE'
```

An apostrophe may be enclosed in double quotation marks, or double quotation marks may be enclosed in single quotation marks. Single or double quotation marks may be embedded in their own string by repeating them immediately. All of the following are valid:

"That's not all..."
'The file "FIRST" was not found'
'That''s not all...'
"The file ""FIRST"" was not found"

Statement

An assembly language *statement* consists of a name, a mnemonic, operands, and a comment. Statements generally fall into two classes—instructions and directives. *Instructions* are executable statements, and *directives* are statements that provide information to assist the assembler in producing executable code. The general format of a statement is:

```
[name] [mnemonic] [operands] [;comment]
```

Statements are *free-form,* meaning that they may be written in any column with any number of spaces between each operand. Blank lines are permitted anywhere in a program. A statement must be written on a single line and may not extend past column 128.

A *directive,* or *pseudo-op,* is a statement that affects either the program listing or the way machine code is generated. For example, the DB directive tells MASM to create storage for a variable named **count** and initialize it to 50:

```
count db 50
```

An *instruction* is executed by the microprocessor at runtime. Instructions fall into general types: program control, data transfer, arithmetic, logical, and input-output. Instructions are always translated directly into machine code by the assembler. In fact, a feature of assembly language is that each assembler instruction translates directly into a single machine language instruction.

Name

A *name* identifies a label, variable, symbol, or reserved word. It may contain any of the following characters:

A–Z a–z 0–9 ? _ @ $.

Programmer-chosen names must adhere to the following restrictions:

- Only the first 31 characters are recognized.
- There is no distinction between upper- and lowercase letters.
- The first character may not be a digit.
- If it is used, the period (.) may be used only as the first character.
- A programmer-chosen name may not be the same as an assembler reserved word.

A name used before an assembler *directive* identifies a location or numeric value within the program. For example, the following memory variable has been given the name **count**:

```
count   db   50
```

If a name appears next to a program instruction, it is called a *label.* Labels serve as place markers whenever transfers of control are required within programs. In the following example, **start_program** is a label identifying a location in an assembly language program:

```
start_program:   mov ax,0
                 mov bx,0
```

A label can also appear on a blank line:

```
Label1: mov ax,0
```

Reserved Name. A reserved name, or *keyword,* has a special predefined meaning. Reserved names are *case insensitive*—they may contain uppercase letters, lowercase letters, or a combination of the two. Examples of reserved names in IBM-PC assembly language are MOV, PROC, TITLE, ADD, AX, and END. For example, the word MOV is an assembly language instruction mnemonic. It cannot be used as a label before an instruction, because it has a predefined meaning.

Another Program Example

We're going to write a program called ECHO, which inputs a character from the keyboard and echoes it on the screen. It demonstrates the way DOS handles input-output.

DOS has a set of *function calls* that provide a convenient set of input-output instructions for your programs. Each is activated by using the **INT 21h** instruction and by placing a function selector number in the AH register. First, to input a character from the keyboard, we use the following two instructions:

```
mov  ah,1   ; DOS function #1: keyboard input
int  21h    ; call DOS to do the work
```

The program will stop and wait for a key to be pressed; then the key's ASCII code will be placed in the AL register. Next, we will use DOS function 2 to display a character on the console. The ASCII code of the character must be in DL. We want to display the character currently in AL, so we move it to DL where DOS can display it:

```
mov  ah,2    ; DOS function #1: console output
mov  dl,al   ; display the character in AL
int  21h     ; call DOS
```

Complete Program. The complete program will now be listed, along with the DEBUG commands to be used when assembling and testing it. Remember not to use the **h** radix on numbers in DEBUG, because all numbers are assumed to be in hexadecimal anyway. Do not type the comments on the right side:

```
DEBUG
-A 100
mov ah,1
int 21          (input character will be in AL)
mov ah,2
mov dl,al       (move the character to DL)
int 21
int 20          (end the program)
                (press ENTER on the blank line)
-U              (unassemble the program)
-G              (Go-execute the program)
-Q              (Quit-return to DOS)
```

The Procedure Trace Command. To trace a program containing INT instructions, you have to use the P (Procedure trace) command instead of the T command. The T command would trace into DOS's input-output subroutines, and you would probably never get back to your own program. A sample trace of the program using DEBUG is shown in Figure 1–3.

POINTS TO REMEMBER

The first purpose of this chapter has been to explain what assembly language is, why it is used, and what types of programs assembly programmers write. Second, we have briefly covered the binary and hexadecimal numbering systems, because they are the foundation of data representation on the IBM-PC. When writing and debugging programs in assembly language, you must be able to *think* in binary and hexadecimal.

A tremendous amout of time and work goes into mastering IBM-PC assembly language. Many skills must be mastered before the process of programming in assembler seems natural. The IBM-PC appears deceptively simple, yet its as-

```
-R
AX=0000  BX=0000  CX=000C  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D7D  ES=3D7D  SS=3D7D  CS=3D7D  IP=0100   NV UP EI PL NZ NA PO NC
3D7D:0100 B401          MOV     AH,01
-P

AX=0100  BX=0000  CX=000C  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D7D  ES=3D7D  SS=3D7D  CS=3D7D  IP=0102   NV UP EI PL NZ NA PO NC
3D7D:0102 CD21          INT     21
-P

*
AX=012A  BX=0000  CX=000C  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D7D  ES=3D7D  SS=3D7D  CS=3D7D  IP=0104   NV UP EI PL NZ NA PO NC
3D7D:0104 B402          MOV     AH,02
-P

AX=022A  BX=0000  CX=000C  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D7D  ES=3D7D  SS=3D7D  CS=3D7D  IP=0106   NV UP EI PL NZ NA PO NC
3D7D:0106 88C2          MOV     DL,AL
-P

AX=022A  BX=0000  CX=000C  DX=002A  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D7D  ES=3D7D  SS=3D7D  CS=3D7D  IP=0108   NV UP EI PL NZ NA PO NC
3D7D:0108 CD21          INT     21
-P

*
AX=022A  BX=0000  CX=000F  DX=002A  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3D7D  ES=3D7D  SS=3D7D  CS=3D7D  IP=010A   NV UP EI PL NZ NA PO NC
3D7D:010A CD20          INT     20
-P
```

Figure 1-3 *Trace output of the ECHO program, using DEBUG.*

sembly language is surprisingly complex. Within the next 10 years, you will probably learn three or four different assembly languages. You can take heart in knowing that many skills learned here will apply to other assemblers.

Debuggers. Use one of the debugging programs mentioned in this chapter (DEBUG, SYMDEB, or CODEVIEW), or another if one is available. There are also several excellent shareware debuggers available on electronic bulletin boards, including the nationwide Compuserve network. The reward for this effort is threefold: You will master specific skills in assembly language programming, you will gain a great deal of confidence in your ability to tackle systems-level programming on the IBM-PC, and you will gain insight into how high-level language compilers generate machine instructions.

We have introduced the following DEBUG commands:

A 100 — *Assemble* instructions into machine language, beginning at location 100.

G — *Go* (execute) from the current location to the end of the program.

P	*Procedure trace* a single instruction or DOS function call. This prevents you from having to trace instructions within DOS functions.
Q	*Quit* DEBUG and return to DOS.
R	*Register* display.
T	*Trace* a single instruction, and show the contents of the registers after it is executed.
U	*Unassemble*. Disassemble memory into assembly language instructions.

REVIEW QUESTIONS

1. Why should computer students learn assembly language?

2. Before reading this chapter, how did you think assembly language was used? Has your impression changed?

3. Can you recall a debugging problem that occurred when you were unable to proceed without help because you didn't know enough about the IBM-PC's operating system?

4. Think of at least two things that you cannot do in a high-level language that you would like to be able to do in assembly language.

5. Find a book, article, or other publication that uses octal notation.

6. Using a high-level language, show an example of converting data from character format to numeric format and vice versa.

7. Write the hexadecimal and binary representations of the characters 'XY', using the ASCII table at the end of this book.

8. Suppose your computer has 24-bit registers, and the highest bit of a signed number holds the sign. What would be the largest *positive* number, expressed in both binary and hexadecimal, that a register could hold?

9. How many bits does a word of storage contain on the IBM-PC?

10. What are the decimal values of the following signed numbers?
    a. 10000000b
    b. 01111111b

11. How does a low-level language differ from a high-level language?

12. Define the following terms in your own words:
    a. machine instruction
    b. instruction mnemonic
    c. operand
    d. register
    e. disassemble

13. In the following machine instruction, which byte probably contains the op code?

    05 0A 00

14. Which DEBUG command lets you do each of the following?
    a. Execute a single instruction, stop, and display all registers.
    b. Execute the remainder of a program.
    c. Begin assembling statements that will be converted to machine language.
    d. Return to DOS.

PROGRAMMING EXERCISES

DEBUG Exercises

For each of the following program excerpts, draw a diagram showing the movement of data as each instruction executes. Write down the final contents of the AX, BX, CX, and DX registers. Then assemble and trace the instructions using DEBUG, and verify your answers.

Exercise 1

```
mov   ah,7F
mov   ax,1234
mov   bh,al
mov   bl,ah
int   20
```

Exercise 2

Assuming that the numbers in this program excerpt are signed, explain what has happened to the signed result in the AL register. Use decimal values in your explanation.

```
mov   al,81
add   al,0FE
int   20
```

Exercise 3

(The following SUB instruction subtracts the second operand from the first.)

```
mov   al,FFFE
sub   al,2
mov   bl,8C
mov   bh,2D
add   bx,ax
int   20
```

Assuming that all values in the preceding program are signed, write down the signed decimal value in AX at the end of the program.

Exercise 4

(The NOT instruction reverses all bits in a number.)

```
mov   bl,FE
not   bl
add   bl,1
int   20
```

Exercise 5

(The INC instruction adds 1 to a number, and the DEC instruction subtracts 1.)

```
mov   ax,0
dec   ax
dec   ax
add   ax,2
inc   ax
int   20
```

ANSWERS TO REVIEW QUESTIONS

6. BASIC: N = ASC("B") : C$ = CHR$(66)
   Pascal: n := ord('Z'); ch := char(90);

7. a. 5859h b. 0101100001011001b

8. 011111111111111111111111b or 7FFFFFh

9. A word is 16 bits.

10. a. -128 b. +127

11. Each instruction in a low-level language matches one machine instruction; each instruction in a high-level language is translated into many machine instructions.

12. See the definitions in the chapter.

13. The first byte: 05

14. a. T (Trace) b. G (Go) c. A (Assemble) d. Q (Quit)