

4

Introduction to IBM PC Assembly Language

Overview

This chapter covers the essential steps in creating, assembling, and executing an assembly language program. By the chapter's end you will be able to write simple but interesting programs that carry out useful tasks, and run them on the computer.

As with any programming language, the first step is to learn the syntax, which for assembly language is relatively simple. Next we show how variables are declared, and introduce basic data movement and arithmetic instructions. Finally, we cover program organization; you'll see that assembly language programs are comprised of code, data, and the stack, just like a machine language program.

Because assembly language instructions are so basic, input/output is much harder in assembly language than in high-level languages. We use DOS functions for I/O; they are easy to invoke and are fast enough for all but the most demanding applications.

An assembly language program must be converted to a machine language program before it can be executed. Section 4.10 explains the steps. To demonstrate, we'll create sample programs. They illustrate some standard assembly language programming techniques and serve as models for the exercises.

4.1 Assembly Language Syntax

Assembly language programs are translated into machine language instructions by an assembler, so they must be written to conform to the assembler's specifications. In this book we use the Microsoft Macro Assembler (MASM). Assembly language code is generally not case sensitive, but we use upper case to differentiate code from the rest of the text.

Statements

Programs consist of statements, one per line. Each statement is either an **instruction**, which the assembler translates into machine code, or an **assembler directive**, which instructs the assembler to perform some specific task, such as allocating memory space for a variable or creating a procedure. Both instructions and directives have up to four fields:

name operation operand(s) comment

At least one blank or tab character must separate the fields. The fields do not have to be aligned in a particular column, but they must appear in the above order.

An example of an instruction is

```
START:            MOV CX,5            ;initialize counter
```

Here, the name field consists of the label START:. The operation is MOV, the operands are CX and 5, and the comment is ;initialize counter.

An example of an assembler directive is

```
MAIN                    PROC
```

MAIN is the name, and the operation field contains PROC. This particular directive creates a procedure called MAIN.

4.1.1 Name Field

The name field is used for instruction labels, procedure names, and variable names. The assembler translates names into memory addresses.

Names can be from 1 to 31 characters long, and may consist of letters, digits, and the special characters ? , @ _ \$ %. Embedded blanks are not allowed. If a period is used, it must be the first character. Names may not begin with a digit. The assembler does not differentiate between upper and lower case in a name.

Examples of legal names

```
COUNTER1
?character
SUM_OF_DIGITS
$1000
DONE?
.TEST
```

Examples of illegal names

TWO WORDS	contains a blank
2abc	begins with a digit
A45.28	. not first character
YOU&ME	contains an illegal character

4.1.2**Operation Field**

For an instruction, the operation field contains a symbolic operation code (opcode). The assembler translates a symbolic opcode into a machine language opcode. Opcode symbols often describe the operation's function; for example, MOV, ADD, SUB.

In an assembler directive, the operation field contains a pseudo-operation code (**pseudo-op**). Pseudo-ops are not translated into machine code; rather, they simply tell the assembler to do something. For example, the PROC pseudo-op is used to create a procedure.

4.1.3**Operand Field**

For an instruction, the operand field specifies the data that are to be acted on by the operation. An instruction may have zero, one, or two operands. For example,

NOP	no operands, does nothing
INC AX	one operand; adds 1 to the contents of AX
ADD WORD1, 2	two operands; adds 2 to the contents of memory word WORD1

In a two-operand instruction, the first operand is the **destination operand**. It is the register or memory location where the result is stored (*note*: some instructions don't store the result). The second operand is the **source operand**. The source is usually not modified by the instruction.

For an assembler directive, the operand field usually contains more information about the directive.

4.1.4**Comment Field**

The comment field of a statement is used by the programmer to say something about what the statement does. A semicolon marks the beginning of this field, and the assembler ignores anything typed after the semicolon. Comments are optional, but because assembly language is so low-level, it is almost impossible to understand an assembly language program without comments. In fact, good programming practice dictates a comment on almost every line. The art of good commentary is developed through practice. Don't say something obvious, like this:

```
MOV CX, 0 ;move 0 to CX
```

Instead, use comments to put the instruction into the context of the program:

```
MOV CX, 0 ;CX counts terms, initially 0
```

It is also permissible to make an entire line a comment, and to use them to create space in a program:

```

;
; initialize registers
;
MOV AX, 0
MOV BX, 0

```


4.2 Program Data

The processor operates only on binary data. Thus, the assembler must translate all data representation into binary numbers. However, in an assembly language program we may express data as binary, decimal, or hex numbers, and even as characters.


Numbers

A binary number is written as a bit string followed by the letter “B” or “b”; for example, 1010B.

A decimal number is a string of decimal digits, ending with an optional “D” or “d”.

 A hex number must begin with a decimal digit and end with the letter “H” or “h”; for example, 0ABCH (the reason for this is that the assembler would be unable to tell whether a symbol such as “ABCH” represents the variable name “ABCH” or the hex number ABC).

Any of the preceding numbers may have an optional sign.
Here are examples of legal and illegal numbers for MASM:

Number	Type
11011	decimal 
11011B	binary
64223	decimal
-21843D	decimal
1,234	illegal—contains a nondigit character
1B4DH	hex
1B4D	illegal hex number—doesn't end in “H”
FFFFH	illegal hex number—doesn't begin with a decimal digit
0FFFFH	hex

Characters

Characters and character strings must be enclosed in single or double quotes; for example, “A” or ‘hello’. Characters are translated into their ASCII codes by the assembler, so there is no difference between using “A” and 41h (the ASCII code for “A”) in a program.

Table 4.1 Data-Defining Pseudo-ops

<i>Pseudo-op</i>	<i>Stands for</i>
DB	define byte
DW	define word
DD	define doubleword (two consecutive words)
DQ	define quadword (four consecutive words)
DT	define tenbytes (ten consecutive bytes)

4.3 Variables

Variables play the same role in assembly language that they do in high-level languages. Each variable has a **data type** and is assigned a memory address by the program. The data-defining pseudo-ops and their meanings are listed in Table 4.1. Each pseudo-op can be used to set aside one or more data items of the given type.

In this section we use DB and DW to define byte variables, word variables, and arrays of bytes and words. The other data-defining pseudo-ops are used in Chapter 18 in connection with multiple-precision and noninteger operations.

4.3.1 Byte Variables

The assembler directive that defines a byte variable takes the following form:

```
name DB initial_value
```

where the pseudo-op DB stands for "Define Byte".

For example,

```
ALPHA DB 4
```

This directive causes the assembler to associate a memory byte with the name ALPHA, and initialize it to 4. A question mark (" ? ") used in place of an initial value sets aside an uninitialized byte; for example,

```
BYT DB ?
```

The decimal range of initial values that can be specified is -128 to 127 if a signed interpretation is being given, or 0 to 255 for an unsigned interpretation. These are the ranges of values that fit in a byte.

4.3.2 Word Variables

The assembler directive for defining a word variable has the following form:

```
name DW initial_value
```

The pseudo-op DW means "Define Word." For example,

```
WRD DW -2
```



as with byte variables, a question mark in place of an initial value means an uninitialized word. The decimal range of initial values that can be specified is -32768 to 32767 for a signed interpretation, or 0 to 65535 for an unsigned interpretation.

4.3.3

Arrays

In assembly language, an **array** is just a sequence of memory bytes or words. For example, to define a three-byte array called B_ARRAY, whose initial values are 10h, 20h, and 30h, we can write,

```
B_ARRAY DB 10H, 20H, 30H
```

The name B_ARRAY is associated with the first of these bytes, B_ARRAY+1 with the second, and B_ARRAY+2 with the third. If the assembler assigns the offset address 0200h to B_ARRAY, then memory would look like this:

Symbol	Address	Contents
B_ARRAY	200h	10h
B_ARRAY+1	201h	20h
B_ARRAY+2	202h	30h

In the same way, an array of words may be defined. For example,

```
W_ARRAY DW 1000, 40, 29887, 329
```

sets up an array of four words, with initial values 1000, 40, 29887, and 329. The initial word is associated with the name W_ARRAY, the next one with W_ARRAY + 2, the next with W_ARRAY + 4, and so on. If the array starts at 0300h, it will look like this:

Symbol	Address	Contents
W_ARRAY	0300h	1000d
W_ARRAY+2	0302h	40d
W_ARRAY+4	0304	29887d
W_ARRAY+6	0306h	329d



High and Low Bytes of a Word

Sometimes we need to refer to the high and low bytes of a word variable. Suppose we define

```
WORD1 DW 1234H
```

The low byte of WORD1 contains 34h, and the high byte contains 12h. The low byte has symbolic address WORD1, and the high byte has symbolic address WORD1+1.

Character Strings

An array of ASCII codes can be initialized with a string of characters. For example,

```
LETTERS      DB      'ABC'
```

is equivalent to

```
LETTERS      DB      41H, 42H, 43H
```

Inside a string, the assembler differentiates between upper and lower case. Thus, the string "abc" is translated into three bytes with values 61h, 62h, and 63h.

It is possible to combine characters and numbers in one definition; for example,

```
MSG DB      'HELLO', 0AH, 0DH, 'S'
```

is equivalent to

```
MSG DB      48H, 45H, 4CH, 4CH, 4FH, 0AH, 0DH, 24H
```



4.4 Named Constants

To make assembly language code easier to understand, it is often desirable to use a symbolic name for a constant quantity.

EQU (Equates)

To assign a name to a constant, we can use the **EQU** (equates) pseudo-op. The syntax is

```
name      EQU      constant
```

For example, the statement

```
LF      EQU      0AH
```

assigns the name LF to 0Ah, the ASCII code of the line feed character. The name LF may now be used in place of 0Ah anywhere in the program. Thus, the assembler translates the instructions

```
MOV DL, 0AH
```

and

```
MOV DL, LF
```

into the same machine instruction.

The symbol on the right of an EQU can also be a string. For example,

```
PROMPT EQU 'TYPE YOUR NAME'
```

Then instead of

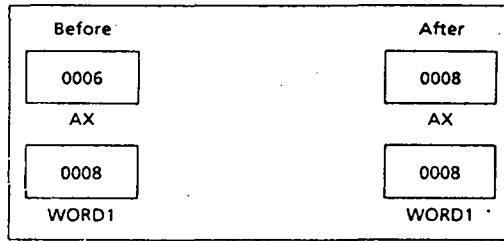
```
MSG DB      'TYPE YOUR NAME'
```

we could say

```
MSG DB      PROMPT
```

Note: no memory is allocated for EQU names.



Figure 4.1 *MOV AX,WORD1*

4.5 A Few Basic Instructions

There are over a hundred instructions in the instruction set for the 8086 CPU; there are also instructions designed especially for the more advanced processors (see Chapter 20). In this section we discuss six of the most useful instructions for transferring data and doing arithmetic. The instructions we present can be used with either byte or word operands.

In the following, WORD1 and WORD2 are word variables, and BYTE1 and BYTE2 are byte variables. Recall from Chapter 3 that AH is the high byte of register AX, and BL is the low byte of BX.

4.5.1 *MOV and XCHG*

The **MOV** instruction is used to transfer data between registers, between a register and a memory location, or to move a number directly into a register or memory location. The syntax is

MOV destination, source

Here are some examples:

MOV AX, WORD1

This reads "Move WORD1 to AX". The contents of register AX are replaced by the contents of memory location WORD1. The contents of WORD1 are unchanged. In other words, a copy of WORD1 is sent to AX (Figure 4.1).

MOV AX, BX

AX gets what was previously in BX. BX is unchanged.

MOV AH, 'A'

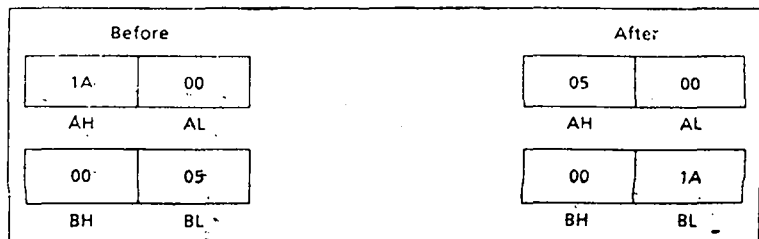
Figure 4.2 *XCHG AH,BL*

Table 4.2 Legal Combinations of Operands for MOV and XCHG

Source Operand	Destination Operand			
	General register	Segment register	Memory location	Constant
General register	yes	yes	yes	no
Segment register	yes	no	yes	no
Memory location	yes	yes	no	no
Constant	yes	no	yes	no

XCHG

Source Operand	Destination Operand	
	General register	Memory location
General register	yes	yes
Memory location	yes	no

This is a move of the number 041h (the ASCII code of "A") into register AH. The previous value of AH is overwritten (replaced by new value).✓

The **XCHG** (exchange) operation is used to exchange the contents of two registers, or a register and a memory location. The syntax is

XCHG destination, source

An example is

XCHG AH, BL

This instruction swaps the contents of AH and BL, so that AH contains what was previously in BL and BL contains what was originally in AH (Figure 4.2). Another example is

XCHG AX, WORD1

which swaps the contents of AX and memory location WORD1.✓

Restrictions on MOV and XCHG

For technical reasons, there are a few restrictions on the use of MOV and XCHG. Table 4.2 shows the allowable combinations. Note in particular that a MOV or XCHG between memory locations is not allowed. For example,

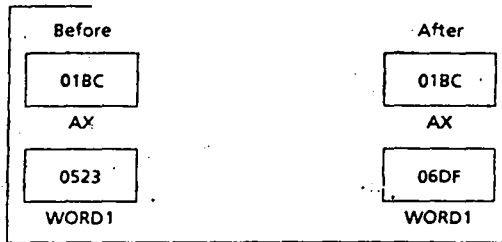
ILLEGAL: MOV WORD1, WORD2

but we can get around this restriction by using a register:

MOV AX, WORD2

MOV WORD1, AX

Figure 4.3 ADD WORD1,AX



4.5.2

ADD, SUB, INC, and DEC

The **ADD** and **SUB** instructions are used to add or subtract the contents of two registers, a register and a memory location, or to add (subtract) a number to (from) a register or memory location. The syntax is

ADD destination, source

SUB destination, source

For example,

ADD WORD1, AX

This instruction, "Add AX to WORD1," causes the contents of AX and memory word WORD1 to be added, and the sum is stored in WORD1. AX is unchanged (Figure 4.3).

SUB AX, DX

In this example, "Subtract DX from AX," the value of DX is subtracted from the value of AX, with the difference being stored in AX. DX is unchanged (Figure 4.4).

Table 4.3 Legal Combinations of Operands for ADD and SUB

Source Operand	Destination Operand	
	General register	Memory location
General register	yes	yes
Memory location	yes	no
Constant	yes	yes

Figure 4.4 SUB AX,DX

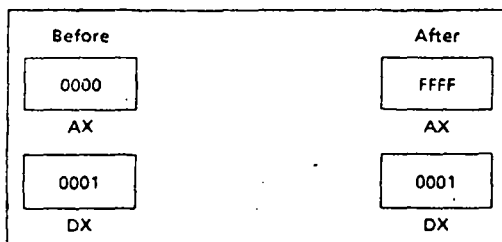
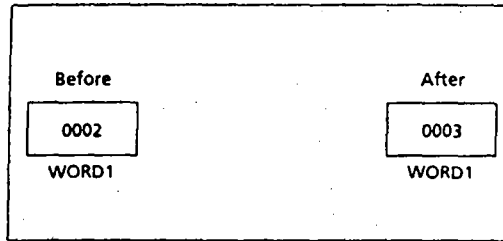


Figure 4.5 INC WORD1



```
ADD BL,5
```

This is an addition of the number 5 to the contents of register BL.

As was the case with MOV and XCHG, there are some restrictions on the combinations of operands allowable with ADD and SUB. The legal ones are summarized in Table 4.3. Direct addition or subtraction between memory locations is illegal; for example,

```
ILLEGAL: ADD BYTE1,BYTE2
```

A solution is to move BYTE2 to a register before adding, thus

```
MOV AL,BYTE2           ;AX gets BYTE2
ADD BYTE1,AL           ;add it to BYTE1
```

INC (increment) is used to add 1 to the contents of a register or memory location and **DEC** (decrement) subtracts 1 from a register or memory location. The syntax is

```
INC destination
DEC destination
```

For example,

```
INC WORD1
```

adds 1 to the contents of WORD1 (Figure 4.5).

```
DEC BYTE1
```

subtracts 1 from variable BYTE1 (Figure 4.6).

Figure 4.6 DEC BYTE1

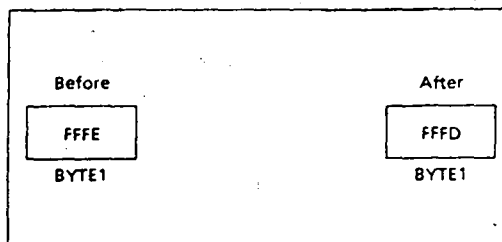
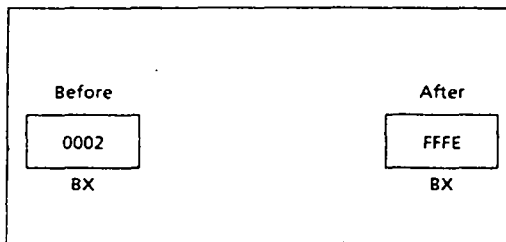


Figure 4.7 NEG BX



4.5.3

NEG

NEG is used to negate the contents of the destination. **NEG** does this by replacing the contents by its two's complement. The syntax is

NEG destination

The destination may be a register or memory location. For example,

NEG BX

negates the contents of **BX** (Figure 4.7).

Type Agreement of Operands

The operands of the preceding two-operand instruction must be of the same type; that is, both bytes or words. Thus an instruction such as

MOV AX, BYTE1 ; illegal

is not allowed. However, the assembler will accept both of the following instructions:

MOV AH, 'A'

and

MOV AX, 'A'

In the former case, the assembler reasons that since the destination **AH** is a byte, the source must be a byte, and it moves 41h into **AH**. In the latter case, it assumes that because the destination is a word, so is the source, and it moves 0041h into **AX**.

4.6

Translation of High-Level Language to Assembly Language

To give you a feeling for the preceding instructions, we'll translate some high-level language assignment statements into assembly language. Only **MOV**, **ADD**, **SUB**, **INC**, **DEC**, and **NEG** are used, although in some cases a better job could be done by using instructions that are covered later. In the discussion, **A** and **B** are word variables.

Statement	Translation
B = A	MOV AX, A ; move A into AX MOV B, AX ; and then into B

As was pointed out earlier, a direct memory-memory move is illegal, so we must move the contents of **A** into a register before moving it to **B**.

```

A = 5 - A          MOV AX, 5          ;put 5 in AX
                   SUB AX, A          ;AX contains 5 - A
                   MOV A, AX          ;put it in A

```

This example illustrates one approach to translating assignment statements: do the arithmetic in a register—for example, AX—then move the result into the destination variable. In this case, there is another, shorter way:

```

                   NEG A              ;A = -A
                   ADD A, 5           ;A = 5 - A

```

The next example shows how to do multiplication by a constant.

```

A = B - 2 x A      MOV AX, B          ;AX has B
                   SUB AX, A          ;AX has B - A
                   SUB AX, A          ;AX has B - 2 x A
                   MOV A, AX          ;move result to A

```

4.7

Program Structure

Chapter 3 noted that machine language programs consist of code, data, and stack. Each part occupies a memory segment. The same organization is reflected in an assembly language program. This time, the code, data, and stack are structured as program segments. Each program segment is translated into a memory segment by the assembler.

We will use the simplified segment definitions that were introduced for the Microsoft Macro Assembler (MASM), version 5.0. They are discussed further in Chapter 14, along with the full segment definitions.

4.7.1

Memory Models

The size of code and data a program can have is determined by specifying a **memory model** using the `.MODEL` directive. The syntax is

```
.MODEL          memory_model
```

The most frequently used memory models are SMALL, MEDIUM, COMPACT, and LARGE. They are described in Table 4.4. Unless there is a lot of code or data, the appropriate model is SMALL. The `.MODEL` directive should come before any segment definition.

Table 4.4 Memory Models


Model	Description
SMALL	code in one segment data in one segment
MEDIUM	code in more than one segment data in one segment
COMPACT	code in one segment data in more than one segment
LARGE	code in more than one segment data in more than one segment no array larger than 64k bytes
HUGE	code in more than one segment data in more than one segment arrays may be larger than 64k bytes

4.7.2

Data Segment

A program's **data segment** contains all the variable definitions. Constant definitions are often made here as well, but they may be placed elsewhere in the program since no memory allocation is involved. To declare a data segment, we use the directive `.DATA`, followed by variable and constant declarations. For example,

```
.DATA
WORD1          DW 2
WORD2          DW 5
MSG            DB 'THIS IS A MESSAGE'
MASK           EQU 10010010B
```



4.7.3.

Stack Segment

The purpose of the **stack segment** declaration is to set aside a block of memory (the stack area) to store the stack. The stack area should be big enough to contain the stack at its maximum size. The declaration syntax is

```
.STACK          size
```

where *size* is an optional number that specifies the stack area size in bytes. For example,

```
.STACK          100H
```

sets aside 100h bytes for the stack area (a reasonable size for most applications). If size is omitted, 1 KB is set aside for the stack area.

4.7.4

Code Segment

The **code segment** contains a program's instructions. The declaration syntax is

```
.CODE name
```

where *name* is the optional name of the segment (there is no need for a name in a `SMALL` program, because the assembler will generate an error).

Inside a code segment, instructions are organized as procedures. The simplest procedure definition is

```
name PROC
;body of the procedure
name ENDP
```

where *name* is the name of the procedure; `PROC` and `ENDP` are pseudo-ops that delineate the procedure.

Here is an example of a code segment definition:

```
.CODE
MAIN PROC
;main procedure instructions
MAIN ENDP
;other procedures go here
```

4.7.5

Putting It Together

Now that you have seen all the program segments, we can construct the general form of a `.SMALL` model program. With minor variations, this form may be used in most applications:

```
.MODEL SMALL
.STACK 100H
.DATA
;data definitions go here
.CODE
MAIN PROC
;instructions go here
MAIN ENDP
;other procedures go here
END MAIN
```

The last line in the program should be the `END` directive, followed by name of the main procedure.

4.8

Input and Output Instructions

In Chapter 1, you saw that the CPU communicates with the peripherals through I/O registers called *I/O ports*. There are two instructions, `IN` and `OUT`, that access the ports directly. These instructions are used when fast I/O is essential; for example, in a game program. However, most applications programs do not use `IN` and `OUT` because (1) port addresses vary among computer models, and (2) it's much easier to program I/O with the service routines provided by the manufacturer.

There are two categories of I/O service routines: (1) the Basic Input/Output System (*BIOS*) routines and (2) the DOS routines. The BIOS routines are stored in ROM and interact directly with the I/O ports. In Chapter 12, we use them to carry out basic screen operations such as moving the cursor and scrolling the screen. The DOS routines can carry out more complex tasks; for example, printing a character string; actually they use the BIOS routines to perform direct I/O operations.

The INT Instruction

To invoke a DOS or BIOS routine, the `INT` (interrupt) instruction is used. It has the format

```
INT interrupt_number
```

where `interrupt_number` is a number that specifies a routine. For example, `INT 16h` invokes a BIOS routine that performs keyboard input. Chapter 15 covers the `INT` instruction in more detail. In the following, we use a particular DOS routine, `INT 21h`.

4.8.1

INT 21h

`INT 21h` may be used to invoke a large number of DOS functions (see Appendix C); a particular function is requested by placing a function number in the `AH` register and invoking `INT 21h`. Here we are interested in the following functions:

Function number	Routine
1	single-key input
2	single-character output✓
9	character string output

INT 21h functions expect input values to be in certain registers and return output values in other registers. These are listed as we describe each function.

Function 1: Single-Key Input

Input: AH = 1
Output: AL = ASCII code if character key is pressed
= 0 if non-character key is pressed

To invoke the routine, execute these instructions:

```
MOV AH,1          ;input key function
INT 21h           ;ASCII code in AL
```

The processor will wait for the user to hit a key if necessary. If a character key is pressed, AL gets its ASCII code; the character is also displayed on the screen. If any other key is pressed, such as an arrow key, F1-F10, and so on, AL will contain 0. The instructions following the INT 21h can examine AL and take appropriate action.

Because INT 21h, function 1, doesn't prompt the user for input, he or she might not know whether the computer is waiting for input or is occupied by some computation. The next function can be used to generate an input prompt.

Function 2: Display a character or execute a control function

Input: AH = 2
DL = ASCII code of the display character or control character
Output: AL = ASCII code of the display character or control character

To display a character with this function, we put its ASCII code in DL. For example, the following instructions cause a question mark to appear on the screen:

```
MOV AH,2          ;display character function
MOV DL,'?'        ;character is '?'✓
INT 21h✓         ;display character
```

After the character is displayed, the cursor advances to the next position on the line (if at the end of the line, the cursor moves to the beginning of the next line).

Function 2 may also be used to perform control functions. If DL contains the ASCII code of a control character, INT 21h causes the control function to be performed. The principal control characters are as follows:

ASCII code (Hex)	Symbol	Function
7	BEL	beep (sounds a tone)
8	BS	backspace
9	HT	tab
A	LF	line feed (new line)
D	CR	carriage return (start of current line)

On execution, AL gets the ASCII code of the control character.

1.9

A First Program

Our first program will read a character from the keyboard and display it at the beginning of the next line.

We start by displaying a question mark:

```
MOV AH,2           ;display character function
MOV DL,'?'         ;character is '?'
INT 21h            ;display character
```



The second instruction moves 3Fh, the ASCII code for "?", into DL.

Next we read a character:

```
MOV AH,1           ;read character function
INT 21h            ;character in AL
```

Now we would like to display the character on the next line. Before doing so, the character must be saved in another register. (We'll see why in a moment.)

```
MOV BL,AL          ;save it in BL
```

To move the cursor to the beginning of the next line, we must execute a carriage return and line feed. We can perform these functions by putting the ASCII codes for them in DL and executing INT 21h.

```
MOV AH,2           ;display character function
MOV DL,0DH         ;carriage return
INT 21h            ;execute carriage return
MOV DL,0AH         ;line feed ✓
INT 21h            ;execute line feed
```

The reason why we had to move the input character from AL to BL is that the INT 21h, function 2, changes AL.

Finally we are ready to display the character:

```
MOV DL,BL          ;get character
INT 21h            ;and display it
```

Here is the complete program:

Program Listing PGM4_1.ASM

```
TITLE PGM4_1: ECHO PROGRAM
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
;display prompt
MOV AH,2           ;display character function
```



```

        MOV DL,'?'          ;character is '?'
        INT 21H             ;display it
;input a character
        MOV AH,1            ;read character function
        INT 21H             ;character in AL
        MOV BL,AL           ;save it in BL
;go to a new line
        MOV AH,2            ;display character function
        MOV DL,0DH          ;carriage return
        INT 21H             ;execute carriage return
        MOV DL,0AH          ;line feed
        INT 21H             ;execute line feed
;display character
        MOV DL,BL           ;retrieve character
        INT 21H             ;and display it
;return to DOS
        MOV AH,4CH          ;DOS exit function
        INT 21H             ;exit to DOS
MAIN    ENDP
        END MAIN

```

Because no variables were used, the data segment was omitted.

Terminating a Program

The last two lines in the MAIN procedure require some explanation. When a program terminates, it should return control to DOS. This can be accomplished by executing INT 21h, function 4Ch.

4.10 Creating and Running a Program

We are now ready to look at the steps involved in creating and running a program. The preceding program is used to demonstrate the process. The four steps are (Figure 4.8):

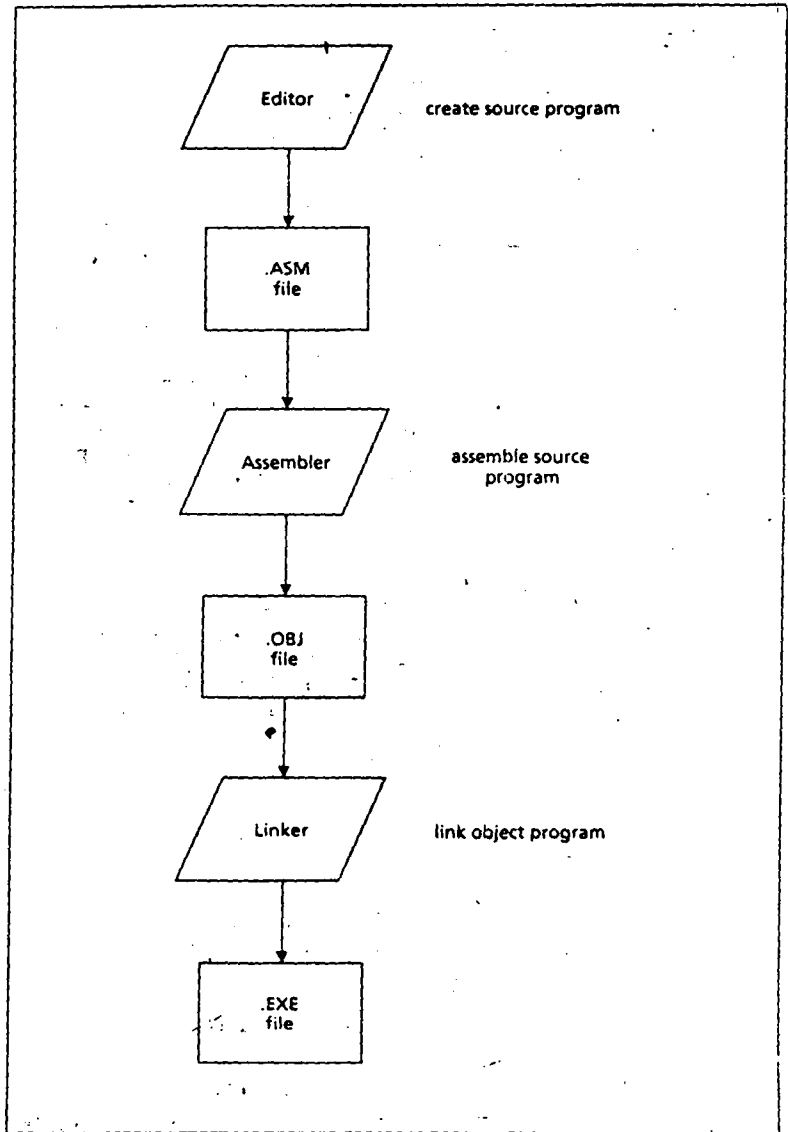
1. Use a text editor or word processor to create a **source program file**.
2. Use an assembler to create a machine language **object file**.
3. Use the LINK program (see description later) to link one or more object files to create a **run file**.
4. Execute the run file.

In this demonstration, the system files we need (assembler and linker, are in drive C and the programmer's disk is in drive A. We make A the default drive so that the files created will be stored on the programmer's disk.

Step 1. Create the Source Program File

We used an editor to create the preceding program, with file name PGM4_1.ASM. The .ASM extension is the conventional extension used to identify an assembly language source file.

Figure 4.8 Programming Steps



Step 2. Assemble the Program

We use the Microsoft Macro Assembler (MASM) to translate the source file PGM4_1.ASM into a machine language object file called PGM4_1.OBJ. The simplest command is (user's response appears in boldface):

```
A>C:MASM PGM4_1;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.
50060 + 418673 Bytes symbol space free
0 Warning Errors
0 Severe Errors
```

After printing copyright information, MASM checks the source file for syntax errors. If it finds any, it will display the line number of each error and a short description. Because there are no errors here, it translates the assembly language code into a machine language object file named PGM4_1.OBJ.

The semicolon after the preceding command means that we *don't* want certain optional files generated. Let's omit it and see what happens:

```
A>C:MASM PGM4_1
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.
Object filename [PGM4_1.OBJ]:
Source listing [NUL.LST]: PGM4_1
Cross-reference [NUL.CRF]: PGM4_1
50060 + 418673 Bytes symbol space free
0 Warning Errors
0 Severe Errors
```

This time MASM prints the names of the files it can create, then waits for us to supply names for the files. The default names are enclosed in square brackets. To accept a name, just press return. The default name NUL means that no file will be created unless the user does specify a name, so we reply with the name PGM4_1.

The Source Listing File

The source listing file (**.LST file**) is a line-numbered text file that displays assembly language code and the corresponding machine code side by side, and gives other information about the program. It is especially helpful for debugging purposes, because MASM's error messages refer to line numbers.

The Cross-Reference File

The cross-reference file (**.CRF file**) is a listing of names that appear in the program and the line numbers on which they occur. It is useful in locating variables and labels in a large program.

Examples of .LST and .CRF files are shown in Appendix D, along with other MASM options.

Step 3. Link the Program

The .OBJ file created in step 2 is a machine language file, but it cannot be executed because it doesn't have the proper run file format. In particular,

1. because it is not known where a program will be loaded in memory for execution, some machine code addresses may not have been filled in.
2. some names used in the program may not have been defined in the program. For example, it may be necessary to create several files for a large program and a procedure in one file may refer to a name defined in another file.

The LINK program takes one or more object files, fills in any missing addresses, and combines the object files into a single executable file (.EXE file). This file can be loaded into memory and run.

To link the program, type

```
A>C:LINK PGM4_1;
```

As before, if the semicolon is omitted, the linker will prompt you for names of the output files generated. See Appendix D.

Step 4. Run the Program

To run it, just type the run file name, with or without the .EXE extension.

```
A>PGM4_1
```

```
?A
```

```
A
```

The program prints a "?" and waits for us to enter a character. We enter "A" and the program echoes it on the next line.

4.11 Displaying a String

In our first program, we used INT 21h, functions 1 and 2, to read and display a single character. Here is another INT 21h function that can be used to display a character string:

INT 21h, Function 9: Display a String

Input: DX = offset address of string.
The string must end with a 'S' character.

The "S" marks the end of the string and is not displayed. If the string contains the ASCII code of a control character, the control function is performed.

To demonstrate this function, we will write a program that prints "HELLO!" on the screen. This message is defined in the data segment as

```
MSG DB 'HELLO!S'
```

The LEA Instruction

INT 21h, function 9, expects the offset address of the character string to be in DX. To get it there, we use a new instruction:

LEA destination, source

where destination is a general register and source is a memory location. **LEA** stands for "Load Effective Address." It puts a copy of the source offset address into the destination. For example,

LEA DX, MSG



puts the offset address of the variable MSG into DX.

Because our second program contains a data segment, it will begin with instructions that initialize DS. The following paragraph explains why these instructions are needed.

Program Segment Prefix

When a program is loaded in memory, DOS prefaces it with a 256-byte **program segment prefix (PSP)**. The PSP contains information about the program. So that programs may access this area, DOS places its segment number in both DS and ES before executing the program. The result is that DS does not contain the segment number of the data segment. To correct this, a program containing a data segment begins with these two instructions:

```
MOV AX, @DATA
MOV DS, AX
```

- @Data is the name of the data segment defined by .DATA. The assembler translates the name @DATA into a segment number. Two instructions are needed because a number (the data segment number) may not be moved directly into a segment register.

With DS initialized, we may print the "HELLO!" message by placing its address in DX and executing INT 21h:

```
LEA DX, MSG      ;get message
MOV AH, 9        ;display string function
INT 21h          ;display string
```

Here is the complete program:

Program Listing PGM4_2.ASM

```
TITLE PGM4_2: PRINT STRING PROGRAM
.MODEL SMALL
.STACK 100H
.DATA
MSG DB 'HELLO!'
.CODE
MAIN PROC
;initialize DS
MOV AX, @DATA
MOV DS, AX      ;initialize DS
;display message
LEA DX, MSG     ;get message
MOV AH, 9       ;display string function
INT 21h         ;display message
;return to DOS
MOV AH, 4CH
```



```

        INT 21h          ;DOS exit
MAIN    ENDP
        END    MAIN

```

And here is a sample execution:

```

A> PGM4_2
HELLO!

```

4.12

A Case Conversion Program

We will now combine most of the material covered in this chapter into a single program. This program begins by prompting the user to enter a lowercase letter, and on the next line displays another message with the letter in uppercase. For example,

```

ENTER A LOWERCASE LETTER:
IN UPPER CASE IT IS: A

```

We use EQU to define CR and LF as names for the constants 0DH and 0AH.

```

CR    EQU 0DH
LF    EQU 0AH

```

The messages and the input character can be stored in the data segment like this:

```

MSG1 DB 'ENTER A LOWERCASE LETTER: $'
MSG2 DB CR,LF,'IN UPPER CASE IT IS: '
CHAR DB '?','$'

```

In defining MSG2 and CHAR, we have used a helpful trick: because the program is supposed to display the second message and the letter (after conversion to upper case) on the next line, MSG2 starts with the ASCII codes for carriage return and line feed; when MSG2 is displayed with INT 21h, function 9, these control functions are executed and the output is displayed on the next line. Because MSG2 does not end with '\$', INT 21h goes on and displays the character stored in CHAR.

Our program begins by displaying the first message and reading the character:

```

LEA DX,MSG1          ;get first message
MOV AH,9             ;display string function
INT 21h              ;display first message
MOV AH,1             ;read character function
INT 21h              ;read a small letter into AL

```

Having read a lowercase letter, the program must convert it to upper case. In the ASCII character sequence, the lowercase letters begin at 61h and the uppercase letters start at 41h, so subtraction of 20h from the contents of AL does the conversion:

```

SUB AL,20H           ;convert it to upper case
MOV CHAR,AL          ;and store it

```

Now the program displays the second message and the uppercase letter:

```

LEA DX,MSG2          ;get second message
MOV AH,9             ;display string function
INT 21h              ;display message and uppercase letter

```

Here is the complete program:

Program Listing PGM4_3.ASM

```

TITLE PGM4_3: CASE CONVERSION PROGRAM
.MODEL SMALL
.STACK 100H
.DATA
    CR EQU 0DH
    LF EQU 0AH
MSG1 DB 'ENTER A LOWER CASE LETTER: $'
MSG2 DB 0DH,0AH,'IN UPPER CASE IT IS: '
CHAR DB '?,$'
.CODE
MAIN PROC
;initialize DS
    MOV AX,?DATA    ;get data segment
    MOV DS,AX        ;initialize DS
;print user prompt
    LEA DX,MSG1      ;get first message
    MOV AH,9         ;display string function
    INT 21H          ;display first message
;input a character and convert to upper case
    MOV AH,1         ;read character function
    INT 21H          ;read a small letter into AL
    SUB AL,20H        ;convert it to upper case
    MOV CHAR,AL      ;and store it
;display on the next line
    LEA DX,MSG2      ;get second message
    MOV AH,9         ;display string function
    INT 21H          ;display message and upper case
;DOS exit
    MOV AH,4CH
    INT 21H          ;DOS exit
MAIN ENDP
END MAIN

```

Summary

- Assembly language programs are made up of statements. A statement is either an instruction to be executed by the computer, or a directive for the assembler.
- Statements have name, operation, operand(s), and comment fields.
- A symbolic name can contain up to 31 characters. The characters can be letters, digits, and certain special symbols.
- Numbers may be written in binary, decimal, or hex.
- Characters and character strings must be enclosed in single or double quotes.

- Directives DB and DW are used to define byte and word variables, respectively. EQU can be used to give names to constants.
- A program generally contains a code segment, a data segment, and a stack segment.
- MOV and XCHG are used to transfer data. There are some restrictions for the use of these instructions; for example, they may not operate directly between memory locations.
- ADD, SUB, INC, DEC, and NEG are some of the basic arithmetic instructions.
- There are two ways to do input and output on the IBM PC: (1) by direct communication with I/O devices, (2) by using BIOS or DOS interrupt routines.
- The direct method is fastest, but is tedious to program and depends on specific hardware circuits.
- Input and output of characters and strings may be done by the DOS routine INT 21h.
- INT 21h, function 1, causes a keyboard character to be read into AL.
- INT 21h, function 2, causes the character whose ASCII code is in DL to be displayed. If DL contains the code of a control character, the control function is performed.
- INT 21h, function 9, causes the string whose offset address is in DX to be displayed. The string must end with a "\$" character.

Glossary

array	A sequence of memory bytes or words
assembler directive	Directs the assembler to perform some specific task
code segment	Part of the program that holds the instructions
.CRF file	A file created by the assembler that lists names that appear in a program and line numbers where they occur
data segment	Part of the program that holds variables
destination operand	First operand in an instruction—receives the result
.EXE file	Same as run file
instruction	A statement that the assembler translates to machine code
.LST file	A line-numbered file created by the assembler that displays assembly language code, machine code, and other information about a program
memory model	Organization of a program that indicates the amount of code and data

object file	The machine language file created by the assembler from the source program file
Program segment prefix, PSP	The 256-byte area that precedes the program in memory—contains information about the program
pseudo-op	Assembler directive
run file	The executable machine language file created by the LINK program
source operand	Second operand in an instruction—usually not changed by the instruction
source program file	A program text file created with a word processor or text editor
stack segment	Part of the program that holds the run-time stack
variable	Symbolic name for a memory location that stores data

New Instructions

ADD	INT	NEG
DEC	LEA	SUB
INC	MOV	XCHG

New Pseudo-Ops

.CODE	.MODEL	EQU
.DATA	.STACK	

Exercises

- Which of the following names are legal in IBM PC assembly language?
 - TWO_WORDS
 - ?1
 - Two words
 - .@?
 - \$145
 - LET'S_GO
 - T =
- Which of the following are legal numbers? If they are legal, tell whether they are binary, decimal, or hex numbers.
 - 246
 - 246h
 - 1001
 - 1,101
 - 2A3h
 - FFEH

- g. 0Ah
 - h. Bh
 - i. 1110b
3. If it is legal, give data definition pseudo-ops to define each of the following.
- a. A word variable A initialized to 52
 - b. A word variable WORD1, uninitialized
 - c. A byte variable B, initialized to 25h
 - d. A byte variable C1, uninitialized
 - e. A word variable WORD2, initialized to 65536
 - f. A word array ARRAY1, initialized to the first five positive integers (i.e. 1-5)
 - g. A constant BELL equal to 07h
 - h. A constant MSG equal to 'THIS IS A MESSAGE\$'
4. Suppose that the following data are loaded starting at offset 0000h:
- | | | |
|---|----|---------|
| A | DB | 7 |
| B | DW | 1ABCh |
| C | DB | 'HELLO' |
- a. Give the offset address assigned to variables A, B, and C.
 - b. Give the contents of the byte at offset 0002h in hex.
 - c. Give the contents of the byte at offset 0004h in hex.
 - d. Give the offset address of the character "O" in "HELLO."
5. Tell whether each of the following instructions is legal or illegal. W1 and W2 are word variables, and B1 and B2 are byte variables.
- a. MOV DS, AX
 - b. MOV DS, 1000h
 - c. MOV CS, ES
 - d. MOV W1, DS
 - e. XCHG W1, W2
 - f. SUB 5, B1
 - g. ADD B1, B2
 - h. ADD AL, 25h
 - i. MOV W1, B1
6. Using only MOV, ADD, SUB, INC, DEC, and NEG, translate the following high-level language assignment statements into assembly language. A, B, and C are word variables.
- a. $A = B - A$
 - b. $A = -(A + 1)$
 - c. $C = A + B$
 - d. $B = 3 \times B + 7$
 - e. $A = B - A - 1$
7. Write instructions (not a complete program) to do the following.
- a. Read a character, and display it at the next position on the same line.
 - b. Read an uppercase letter (omit error checking), and display it at the next position on the same line in lower case.

Programming Exercises

8. Write a program to (a) display a "?", (b) read two decimal digits whose sum is less than 10, (c) display them and their sum on the next line, with an appropriate message.

Sample execution:

```
?27
```

```
THE SUM OF 2 AND 7 IS 9
```

9. Write a program to (a) prompt the user, (b) read first, middle, and last initials of a person's name, and (c) display them down the left margin.

Sample execution:

```
ENTER THREE INITIALS: JFK
```

```
J
```

```
F
```

```
K
```

10. Write a program to read one of the hex digits A-F, and display it on the next line in decimal.

Sample execution:

```
ENTER A HEX DIGIT: C
```

```
IN DECIMAL IT IS 12
```

11. Write a program to display a 10 × 10 solid box of asterisks.
Hint: declare a string in the data segment that specifies the box, and display it with INT 21h, function 9h.
12. Write a program to (a) display "?", (b) read three initials, (c) display them in the middle of an 11 × 11 box of asterisks, and (d) beep the computer.