CHAPTER 2

ASSEMBLY LANGUAGE PROGRAMMING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **Explain the difference between Assembly language instructions and pseudo-instructions**
- **Identify the segments of an Assembly language program**
- **Code simple Assembly language instructions**
- **Assemble, link, and run a simple Assembly language program**
- **Code control transfer instructions such as conditional and unconditional jumps and call instructions**
- **Code Assembly language data directives for binary, hex, decimal, or ASCII data**
- **Write an Assembly language program using either the full segment definition or the simplified segment definition**
- **Explain the difference between COM and EXE files and list the advantages of each**

This chapter is an introduction to Assembly language programming with the 80x86. First the basic form of a program is explained, followed by the steps required to edit, assemble, link, and run a program. Next, control transfer instructions such as jump and call are discussed and data types and data directives in 80x86 Assembly language are explained. Then the full segment definition is discussed. Finally, the differences between ".exe" and ".com" files are explained. The programs in this chapter and following ones can be assembled and run on any IBM PC, PS and compatible computer with an 8088/86 or higher microprocessor.

SECTION 2.1: DIRECTIVES AND A SAMPLE PROGRAM

In this section we explain the components of a simple Assembly language program to be assembled by the assembler. A given Assembly language program (see Figure 2-1) is a series of statements, or lines, which are either Assembly language instructions such as ADD and MOV, or statements called directives. *Directives* (also called *pseudo-instructions*) give directions to the assembler about how it should translate the Assembly language instructions into machine code. An Assembly language instruction consists of four fields:

```
[label:]  mnemonic  [operands]  [;comment]
```

Brackets indicate that the field is optional. Do not type in the brackets.

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed 31 characters. Labels for directives do not need to end with a colon. A label must end with a colon when it refers to an opcode generating instruction; the colon indicates to the assembler that this refers to code within this code segment. Appendix C, Section 2 gives more information about labels.

2,3. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Assembly language statements such as

```
ADD   AL,BL
MOV   AX,6764
```

ADD and MOV are the mnemonic opcodes and "AL,BL" and "AX,6764" are the operands. Instead of a mnemonic and operand, these two fields could contain assembler pseudo-instructions, or directives. They are used by the assembler to organize the program as well as other output files. Directives do not generate any machine code and are used only by the assembler as opposed to instructions, which are translated into machine code for the CPU to execute. In Figure 2-1 the commands DB, END, and ENDP are examples of directives.

4. The comment field begins with a ";". Comments may be at the end of a line or on a line by themselves. The assembler ignores comments, but they are indispensable to programmers. Comments are optional, but are highly recommended to make it easier for someone to read and understand the program.

Model definition

The first statement in Figure 2-1 after the comments is the MODEL directive. This directive selects the size of the memory model. Among the options for the memory model are SMALL, MEDIUM, COMPACT, and LARGE.

```
.MODEL SMALL          ;this directive defines the model as small
```

SMALL is one of the most widely used memory models for Assembly language programs and is sufficient for the programs in this book. The small model uses a maximum of 64K bytes of memory for code and another 64K bytes for data. The other models are defined as follows:

```
.MODEL MEDIUM     ;the data must fit into 64K bytes
                  ;but the code can exceed 64K bytes of memory
.MODEL COMPACT    ;the data can exceed 64K bytes
                  ;but the code cannot exceed 64K bytes
.MODEL LARGE      ;both data and code can exceed 64K
                  ;but no single set of data should exceed 64K
.MODEL HUGE       ;both code and data can exceed 64K
                  ;data items (such as arrays) can exceed 64K
.MODEL TINY       ;used with COM files in which data and code
                  ;must fit into 64K bytes
```

Notice in the above list that MEDIUM and COMPACT are opposites. Also note that the TINY model cannot be used with the simplified segment definition described in this section.

Segment definition

As mentioned in Chapter 1, the 80x86 CPU has four segment registers: CS (code segment), DS (data segment), SS (stack segment), and ES (extra segment). Every line of an Assembly language program must correspond to one of these segments. The simplified segment definition format uses three simple directives: ".CODE", ".DATA", and ".STACK", which correspond to the CS, DS, and SS registers, respectively. There is another segment definition style called the *full segment definition*, which is described in Section 2.6.

Segments of a program

Although one can write an Assembly language program that uses only one segment, normally a program consists of at least three segments: the stack segment, the data segment, and the code segment.

```
.STACK    ;marks the beginning of the stack segment
.DATA     ;marks the beginning of the data segment
.CODE     ;marks the beginning of the code segment
```

Assembly language statements are grouped into segments in order to be recognized by the assembler and consequently by the CPU. The stack segment defines storage for the stack, the data segment defines the data that the program will use, and the code segment contains the Assembly language instructions. In Chapter 1 we gave an overview of how these segments were stored in memory. In the following pages we describe the stack, data, and code segments as they are defined in Assembly language programming.

Stack segment

The following directive reserves 64 bytes of memory for the stack:

```
.STACK 64
```

Data segment

The data segment in the program of Figure 2-1 defines three data items: DATA1, DATA2, and SUM. Each is defined as DB (define byte). The DB directive is used by the assembler to allocate memory in byte-sized chunks. Memory

can be allocated in different sizes, such as 2 bytes, which has the directive DW (define word). More of these pseudo-instructions are discussed in detail in Section 2.5. The data items defined in the data segment will be accessed in the code segment by their labels. DATA1 and DATA2 are given initial values in the data section. SUM is not given an initial value, but storage is set aside for it.

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
            .MODEL SMALL
            .STACK 64
            .DATA
DATA1       DB    52H
DATA2       DB    29H
SUM         DB    ?
            .CODE
MAIN        PROC  FAR          ;this is the program entry point
            MOV   AX,@DATA     ;load the data segment address
            MOV   DS,AX        ;assign value to DS
            MOV   AL,DATA1     ;get the first operand
            MOV   BL,DATA2     ;get the second operand
            ADD   AL,BL        ;add the operands
            MOV   SUM,AL       ;store the result in location SUM
            MOV   AH,4CH       ;set up to return to DOS
            INT   21H          ;
MAIN        ENDP
            END   MAIN         ;this is the program exit point
```

Figure 2-1. Simple Assembly Language Program

Code segment definition

The last segment of the program in Figure 2-1 is the code segment. The first line of the segment after the .CODE directive is the PROC directive. A *procedure* is a group of instructions designed to accomplish a specific function. A code segment may consist of only one procedure, but usually is organized into several small procedures in order to make the program more structured. Every procedure must have a name defined by the PROC directive, followed by the assembly language instructions and closed by the ENDP directive. The PROC and ENDP statements must have the same label. The PROC directive may have the option FAR or NEAR. The operating system that controls the computer must be directed to the beginning of the program in order to execute it. DOS requires that the entry point to the user program be a FAR procedure. From then on, either FAR or NEAR can be used. The differences between a FAR and a NEAR procedure, as well as where and why each is used, are explained later in this chapter. For now, just remember that in order to run a program, FAR must be used at the program entry point.

A good question to ask at this point is: What value is actually assigned to the CS, DS, and SS registers for execution of the program? The DOS operating system must pass control to the program so that it may execute, but before it does that it assigns values for the segment registers. The operating system must do this because it knows how much memory is installed in the computer, how much of it is used by the system, and how much is available. In the IBM PC, the operating system first finds out how many kilobytes of RAM memory are installed, allocates some for its own use, and then allows the user program to use the portions that it needs. Various DOS versions require different amounts of memory, and since the user program must be able to run across different versions, one cannot

tell DOS to give the program a specific area of memory, say from 25FFF to 289E2. Therefore, it is the job of DOS to assign exact values for the segment registers. When the program begins executing, of the three segment registers, only CS and SS have the proper values. The DS value (and ES, if used) must be initialized by the program. This is done as follows:

```
MOV AX,@DATA        ;DATA refers to the start of the data segment
MOV DS,AX
```

Remember from Chapter 1 that no segment register can be loaded directly. That is the reason the two lines of code above are needed. You cannot code "MOV DS,@DATA".

After these housekeeping chores, the Assembly language program instructions can be written to perform the desired tasks. In Figure 2-1, the program loads AL and BL with DATA1 and DATA2, respectively, ADDs them together, and stores the result in SUM.

```
MOV  AL,DATA1
MOV  BL,DATA2
ADD  AL,BL
MOV  SUM,AL
```

The two last instructions in the shell are:

```
MOV    AH,4CH
INT    21H
```

Their purpose is to return control to the operating system. The last two lines end the procedure and the program, respectively. Note that the label for ENDP (MAIN) matches the label for PROC. The END pseudo-instruction ends the entire program by indicating to DOS that the entry point MAIN has ended. For this reason the labels for the entry point and END must match.

Figure 2-2 shows a sample shell of an Assembly language program. When writing your first few programs, it is handy to keep a copy of this shell on your disk and simply fill it in with the instructions and data for your program.

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
; USING SIMPLIFIED SEGMENT DEFINITION
                .MODEL SMALL
                .STACK 64
                .DATA
                ;
                ;place data definitions here
                ;
                .CODE
MAIN            PROC  FAR           ;this is the program entry point
                MOV   AX,@DATA      ;load the data segment address
                MOV   DS,AX         ;assign value to DS
                ;
                ;place code here
                ;
                MOV   AH,4CH        ;set up to
                INT   21H           ;return to DOS
MAIN            ENDP
                END   MAIN          ;this is the program exit point
```

Figure 2-2. Shell of an Assembly Language Program

Review Questions

1. What is the purpose of pseudo-instructions?
2. _____ are translated by the assembler into machine code, whereas _____ are not.
3. Write an Assembly language program with the following characteristics:
   (a) A data item named HIGH_DAT, which contains 95
   (b) Instructions that move HIGH_DAT to registers AH, BH, and DL
   (c) A program entry point named START
4. Find the errors in the following:

```
        .MODEL ENORMOUS
        .STACK
        .CODE
        .DATA
MAIN    PROC  FAR
        MOV   AX,DATA
        MOV   DS,@DATA
        MOV   AL,34H
        ADD   AL,4FH
        MOV   DATA1,AL
START   ENDP
        END
```

SECTION 2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

Now that the basic form of an Assembly language program has been given, the next question is: How is it created and assembled? The three steps to create an executable Assembly language program are outlined as follows:

Step	Input	Program	Output
1. Edit the program	keyboard	editor	myfile.asm
2. Assemble the program	myfile.asm	MASM or TASM	myfile.obj
3. Link the program	myfile.obj	LINK or TLINK	myfile.exe

The MASM and LINK programs are the assembler and linker programs for Microsoft's MASM assembler. If you are using another assembler, such as Borland's TASM, consult the manual for the procedure to assemble and link a program. Many excellent editors or word processors are available that can be used to create and/or edit the program. The editor must be able to produce an ASCII file. Although filenames follow the usual DOS conventions, the source file must end in ".asm" for the assembler used in this book. This ".asm" source file is assembled by an assembler, such as Microsoft's MASM, or Borland's TASM. The assembler will produce an object file and a list file, along with other

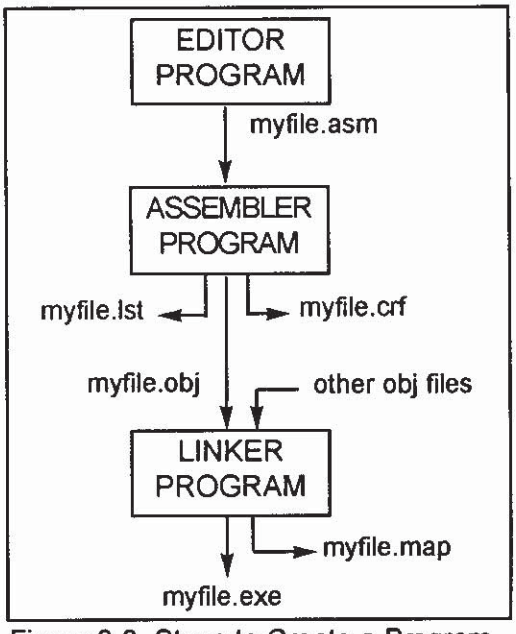

Figure 2-3. Steps to Create a Program

files that may be useful to the programmer. The extension for the object file must be ".obj". This object file is input to the LINK program, which produces the executable program that ends in ".exe". The ".exe" file can be executed by the microprocessor. Before feeding the ".obj" file into LINK, all syntax errors produced by the assembler must be corrected. Of course, fixing these errors will not guarantee that the program will work as intended since the program may contain conceptual errors. Figure 2-3 shows the steps in producing an executable file.

Figure 2-4 shows how an executable program is created by following the steps outlined above, and then run under DEBUG. The portions in bold indicate what the user would type in to perform these steps. Figure 2-4 assumes that the MASM, LINK, and DEBUG programs are on drive C and the Assembly language program is on drive A. The drives used will vary depending on how the system is set up.

```
C>MASM A:MYFILE.ASM <enter>

Microsoft (R) Macro Assembler  Version  5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.

Object filename [A:MYFILE.OBJ]: A: <enter>
Source listing  [NUL.LST]:A:MYFILE.LST  <enter>
Cross-reference [NUL.CRF]: <enter>

      47962 + 413345 Bytes symbol space free

          0 Warning Errors
          0 Severe  Errors


C>LINK A:MYFILE.OBJ <enter>

Microsoft (R) Overlay Linker  Version 3.64
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Run File [A:MYFILE.EXE]:A:<enter>
List File [NUL.MAP]: <enter>
Libraries [.LIB]:<enter>
LINK : warning L4021: no stack segment

 C>DEBUG A:MYFILE.EXE <enter>
 -U CS:0 1 <enter>
 1064:0000 B86610          MOV     AX,1066
 -D 1066:0 F <enter>
 1066:0000 52 29 00 00 00 00 00 00-00 00 00 00 00 00 00 00  R).............
-G <enter>
Program terminated normally
-D 1066:0 F <enter>
1066:0000 52 29 7B 00 00 00 00 00-00 00 00 00 00 00 00 00  R){.............
-Q <enter>
C>
```

Figure 2-4. Creating and Running the .exe File
Note: The parts you type in are printed in **bold**.

.asm and .obj files

The ".asm" file (the source file) is the file created with a word processor or line editor. The MASM (or other) assembler converts the .asm file's Assembly language instructions into machine language (the ".obj" object file). In addition to creating the object program, MASM also creates the ".lst" list file.

.lst file

The ".lst" file, which is optional, is very useful to the programmer because it lists all the opcodes and offset addresses as well as errors that MASM detected. MASM assumes that the list file is not wanted (NUL.LST indicates no list). To get a list file, type in a filename after the prompt. This file can be displayed on the monitor or sent to the printer. The programmer uses it to help debug the program. It is only after fixing all the errors indicated in the ".lst" file that the ".obj" file can be input to the LINK program to create the executable program.

One way to look at the list file is to use the following command at the DOS level. This command will print myfile.lst to the monitor, one sceen at a time.

```
C>type myfile.lst | more
```

Another way to look at the list file is to bring it into a word processor. Then you can read it or print it. There are two assembler directives that can be used to make the ".lst" file more readable: PAGE and TITLE.

PAGE and TITLE directives

The format of the PAGE directive is

```
PAGE [lines],[columns]
```

and its function is to tell the printer how the list should be printed. In the default mode, meaning that the PAGE directive is coded with no numbers coming after it, the output will have 66 lines per page with a maximum of 80 characters per line. In this book, programs will change the default settings to 60 and 132 as follows:

```
PAGE 60,132
```

The range for number of lines is 10 to 255 and for columns is 60 to 132. When the list is printed and it is more than one page, the assembler can be instructed to print the title of the program on top of each page. What comes after the TITLE pseudo-instruction is up to the programmer, but it is common practice to put the name of the program as stored on the disk immediately after the TITLE pseudo-instruction and then a brief description of the function of the program. The text after the TITLE pseudo-instruction cannot be more than 60 ASCII characters.

.crf file

MASM produces another optional file, the cross-reference, which has the extension ".crf". It provides an alphabetical list of all symbols and labels used in the program as well as the program line numbers in which they are referenced. This can be a great help in large programs with many data segments and code segments.

LINKing the program

The assembler (MASM) creates the opcodes, operands, and offset addresses under the ".obj" file. It is the LINK program that produces the ready-to-run version of a program that has the ".exe" (EXEcutable) extension. The LINK program sets up the file so that it can be loaded by DOS and executed.

In Figure 2-4 we used DEBUG to execute the program in Figure 2-1 and analyze the result. In the program in Figure 2-1, three data items are defined in the data segment. Before running the program, one could look at the data in the data segment by dumping the contents of DS:offset as shown in Figure 2-4. Now what is the value for the DS register? This can vary from PC to PC and from DOS to DOS. For this reason it is important to look at the value in "MOV AX,xxxx" as was shown and use that number. The result of the program can be verified after it is run as shown in Figure 2-4. When the program is working successfully, it can be run at the DOS level. To execute myfile.exe, simply type in

```
C>A:myfile
```

However, since this program produces no output, there would be no way to verify the results. When the program name is typed in at the DOS level, as shown above, DOS loads the program in memory. This is sometimes referred to as *mapping*, which means that the program is mapped into the physical memory of the PC.

.map file

When there are many segments for code or data, there is a need to see where each is located and how many bytes are used by each. This is provided by the map file. This file, which is optional, gives the name of each segment, where it starts, where it stops, and its size in bytes. In Chapter 7 the importance of the map will be seen when many separate subroutines (modules) are assembled separately and then linked together.

Review Questions

1. (a) The input file to the MASM assembler program has the extension _____.
   (b) The input file to the LINK program has the extension _____.
2. Select all the file types from the second column that are the output of the program in the first column.

_____ Editor	(a) .obj (b) .asm
_____ Assembler	(c) .exe (d) .lst
_____ Linker	(e) .crf (f) .map

SECTION 2.3: MORE SAMPLE PROGRAMS

Now that some familiarity with Assembly language programming in the IBM PC has been achieved, in this section we look at more example programs in order to allow the reader to master the basic features of Assembly programming. The following pages show Program 2-1 and the list file generated when the program was assembled. After the program was assembled and linked, DEBUG was used to dump the code segment to see what value is assigned to the DS register. Precisely where DOS loads a program into RAM depends on many factors, including the amount of RAM on the system and the version of DOS used. Therefore, remember that the value you get could be different for "MOV AX,xxxx" as well as for CS in the program examples. Do not attempt to modify the segment register contents to conform to those in the examples, or your system may crash!

Write, run, and analyze a program that adds 5 bytes of data and saves the result. The data should be the following hex numbers: 25, 12, 15, 1F, and 2B.

```
PAGE      60,132
TITLE     PROG2-1  (EXE)  PURPOSE: ADDS 5 BYTES OF DATA
          .MODEL SMALL
          .STACK 64
;-----------------------------
          .DATA
DATA_IN   DB     25H,12H,15H,1FH,2BH
SUM       DB           ?
;-----------------------------
          .CODE
MAIN      PROC   FAR
          MOV    AX,@DATA
          MOV    DS,AX
          MOV    CX,05                    ;set up loop counter CX=5
          MOV    BX,OFFSET DATA_IN        ;set up data pointer BX
          MOV    AL,0                     ;initialize AL
AGAIN:    ADD    AL,[BX]                  ;add next data item to AL
          INC    BX                       ;make BX point to next data item
          DEC    CX                       ;decrement loop counter
          JNZ    AGAIN                    ;jump if loop counter not zero
          MOV    SUM,AL                   ;load result into sum
          MOV    AH,4CH                   ;set up return
          INT    21H                      ;return to DOS
MAIN      ENDP
          END    MAIN
```

After the program was assembled and linked, it was run using DEBUG:

```
C>debug prog2-1.exe
-u cs:0 19
1067:0000 B86610      MOV   AX,1066
1067:0003 8ED8        MOV   DS,AX
1067:0005 B90500      MOV   CX,0005
1067:0008 BB0000      MOV   BX,0000
1067:000D 0207        ADD   AL,[BX]
1067:000F 43          INC   BX
1067:0010 49          DEC   CX
1067:0013 A20500      MOV   [0005],AL
1067:0016 B44C        MOV   AH,4C
1067:0018 CD21        INT   21
-d 1066:0 f
1066:0000  25 12 15 1F 2B 00 00 00-00 00 00 00 00 00 00 00 %...+.....
-g

Program terminated normally
-d 1066:0 f
1066:0000  25 12 15 1F 2B 96 00 00-00 00 00 00 00 00 00 00 %...+.....
-q
C>
```

Program 2-1

Analysis of Program 2-1

The DEBUG program is explained thoroughly in Appendix A. The commands used in running Program 2-1 were (1) u, to unassemble the code from cs:0 for 19 bytes; (2) d, to dump the contents of memory from 1066:0 for the next F bytes; and (3) g, to go, that is, run the program.

Notice in Program 2-1 that when the program was run in DEBUG, the contents of the data segment memory were dumped before and after execution of the program to verify that the program worked as planned. Normally, it is not necessary to unassemble this much code, but it was done here because in later sec-

```
Turbo Assembler        Version 3.0        06/25/99 12:05:32        Page 1
test21.asm

    1   0000                              .MODEL SMALL
    2   0000                              .STACK 64
    3                                     ;_____________
    4   0000                              .DATA
    5   0000  25 12 15 1F 2B  DATA_IN     DB      25H,12H,15H,1FH,2BH
    6   0005  ??              SUM         DB      ?
    7                                     ;_____________
    8   0006                              .CODE
    9   0000                  MAIN        PROC    FAR
   10   0000  B8 0000s                    MOV     AX,@DATA
   11   0003  8E D8                       MOV     DS,AX
   12   0005  B9 0005                     MOV     CX,05          ;set up loop counter CX=5
   13   0008  BB 0000r                    MOV     BX,OFFSET DATA_IN      ;set up data
   14   000B  B0 00                       MOV     AL,0           ;initialize AL
   15   000D  02 07           AGAIN:      ADD     AL,[BX]        ;add next data item  to AL
   16   000F  43                          INC     BX             ;make BX point to next
   17   0010  49                          DEC     CX             ;decrement loop counter
   18   0011  75 FA                       JNZ     AGAIN          ;jump if counter not zero
   19   0013  A2 0005r                    MOV     SUM,AL         ;load result into sum
   20   0016  B4 4C                       MOV     AH,4CH         ;set up return
   21   0018  CD 21                       INT     21H            ;return to DOS
   22   001A                  MAIN        ENDP
   23                                     END     MAIN

Turbo Assembler        Version 3.0        06/25/99 12:05:32        Page 2
Symbol Table

Symbol Name                    Type   Value

??DATE                         Text   "06/25/99"
??FILENAME                     Text   "test21  "
??TIME                         Text   "12:05:32"
??VERSION                      Number 0300
@32BIT                         Text   0
@CODE                          Text   _TEXT
@CODESIZE                      Text   0
@CPU                           Text   0101H
@CURSEG                        Text   _TEXT
@DATA                          Text   DGROUP
@DATASIZE                      Text   0
@FILENAME                      Text   TEST21
@INTERFACE                     Text   00H
@MODEL                         Text   2
@STACK                         Text   DGROUP
@WORDSIZE                      Text   2
AGAIN                          Near   _TEXT:000D
DATA_IN                        Byte   DGROUP:0000
MAIN                           Far    _TEXT:0000
SUM                            Byte   DGROUP:0005

Groups & Segments              Bit Size Align  Combine Class

DGROUP                         Group
  STACK                        16  0040 Para   Stack   STACK
  _DATA                        16  0006 Word           Public  DATA
_TEXT                          16  001A Word           Public  CODE
```

List File for Program 2-1

tions of the chapter we examine the jump instruction in this program. Also notice that the first 5 bytes dumped above are the data items defined in the data segment of the program and the sixth item is the sum of those five items, so it appears that the program worked correctly (25H + 12H + 15H + 1FH + 2BH = 96H). Program 2-1 is explained below, instruction by instruction.

"MOV CX,05" will load the value 05 into the CX register. This register is used by the program as a counter for iteration (looping).

"MOV BX,OFFSET DATA_IN" will load into BX the offset address assigned to DATA. The assembler starts at offset 0000 and uses memory for the data and then assigns the next available offset memory for SUM (in this case, 0005).

"ADD AL,[BX]" adds the contents of the memory location pointed at by the register BX to AL. Note that [BX] is a pointer to a memory location.

"INC BX" simply increments the pointer by adding 1 to register BX. This will cause BX to point to the next data item, that is, the next byte.

"DEC CX" will decrement (subtract 1 from) the CX counter and will set the zero flag high if CX becomes zero.

"JNZ AGAIN" will jump back to the label AGAIN as long as the zero flag is indicating that CX is not zero. "JNZ AGAIN" will not jump (that is, execution will resume with the next instruction after the JNZ instruction) only after the zero flag has been set high by the "DEC CX" instruction (that is, CX becomes zero). When CX becomes zero, this means that the loop is completed and all five numbers have been added to AL.

Various approaches to Program 2-1

There are many ways in which any program may be written. The method shown for Program 2-1 defined one field of data and used pointer [BX] to access data elements. In the method used below, a name is assigned to each data item that will be accessed in the program. Variations of Program 2-1 are shown below to clarify the use of addressing modes in the context of a real program and also to show that the 80x86 can use any general-purpose register to do arithmetic and logic operations. In earlier-generation CPUs, the accumulator had to be the destination of all arithmetic and logic operations, but in the 80x86 this is not the case. Since the purpose of these examples is to show different ways of accessing operands, it is left to the reader to run and analyze the programs.

```
;from the data segment:
DATA1  DB 25H
DATA2  DB 12H
DATA3  DB 15H
DATA4  DB 1FH
DATA5  DB 2BH
SUM  DB  ?
;from the code segment:
MOV  AL,DATA1      ;MOVE DATA1 INTO AL
ADD  AL,DATA2      ;ADD DATA2 TO AL
ADD  AL,DATA3
ADD  AL,DATA4
ADD  AL,DATA5
MOV  SUM,AL        ;SAVE AL IN SUM
```

There is quite a difference between these two methods of writing the same program. While in the first one the register indirect addressing mode was used to access the data, in the second method the direct addressing mode was used.

Write and run a program that adds four words of data and saves the result. The values will be 234DH, 1DE6H, 3BC7H, and 566AH. Use DEBUG to verify the sum is D364.

```
TITLE          PROG2-2  (EXE)  PURPOSE: ADDS 4 WORDS OF DATA
PAGE   60,132
               .MODEL SMALL
               .STACK 64
;--------------------------------
               .DATA
DATA_IN        DW     234DH,1DE6H,3BC7H,566AH
               ORG    10H
SUM            DW.    ?
;--------------------------------
               .CODE
MAIN           PROC   FAR
               MOV    AX,@DATA
               MOV    DS,AX
               MOV    CX,04                  ;set up loop counter CX=4
               MOV    DI,OFFSET DATA_IN      ;set up data pointer DI
               MOV    BX,00                  ;initialize BX
ADD_LP:        ADD    BX,[DI]                ;add contents pointed at by [DI] to BX
               INC    DI                     ;increment DI twice
               INC    DI                     ;to point to next word
               DEC    CX                     ;decrement loop counter
               JNZ    ADD_LP                 ;jump if loop counter not zero
               MOV    SI,OFFSET SUM          ;load pointer for sum
               MOV    [SI],BX                ;store in data segment
               MOV    AH,4CH                 ;set up return
               INT    21H                    ;return to DOS
MAIN           ENDP
               END    MAIN

After the program was assembled and linked, it was run using DEBUG:
C>debug a:prog2-2.exe
1068:0000  B86610      MOV    AX,1066
-D 1066:0 1F
1066:0000 4D 23 E6 1D C7 3B 6A 56-00 00 00 00 00 00 00 00 M#f.G;jV........
1066:0010 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-G

Program terminated normally
-D 1066:0 1F
1066:0000 4D 23 E6 1D C7 3B 6A 56-00 00 00 00 00 00 00 00 M#f.G;jV........
1066:0010 64 D3 00 00 00 00 00 00-00 00 00 00 00 00 00 00 dS..............
-Q
C>
```

Program 2-2

Analysis of Program 2-2

First notice that the 16-bit data (a word) is stored with the low-order byte first. For example, "234D" as defined in the data segment is stored as "4D23", meaning that the lower address, 0000, has the least significant byte, 4D, and the higher address, 0001, has the most significant byte, 23. This is shown in the DEBUG display of the data segment. Similarly, the sum, D364, is stored as 64D3. As discussed in Chapter 1, this method of low byte to low address and high byte to high address operand assignment is referred to in computer literature as "little endian."

Second, note that the address pointer is incremented twice, since the operand being accessed is a word (two bytes). The program could have used "ADD DI,2" instead of using "INC DI" twice. When storing the result of word addition, "MOV SI,OFFSET SUM" was used to load the pointer (in this case 0010, as defined by ORG 0010H) for the memory allocated for the label SUM,

and then "MOV [SI],BX" was used to move the contents of register BX to memory locations with offsets 0010 and 0011. Again, as was done previously, it could have been coded simply as "MOV SUM,BX", using the direct addressing mode.

Program 2-2 uses the ORG directive. In previous programs where ORG was not used, the assembler would start at offset 0000 and use memory for each data item. The ORG directive can be used to set the offset addresses for data items. Although the programmer cannot assign exact physical addresses, one is allowed to assign offset addresses. The ORG directive in Program 2-2 caused SUM to be stored at DS:0010, as can be seen by looking at the DEBUG display of the data segment.

Write and run a program that transfers 6 bytes of data from memory locations with offset of 0010H to memory locations with offset of 0028H.

```
TITLE          PROG2-3  (EXE)  PURPOSE: TRANSFERS 6 BYTES OF DATA
PAGE  60,132
               .MODEL SMALL
               .STACK 64
               .DATA
               ORG    10H
DATA_IN        DB             25H,4FH,85H,1FH,2BH,0C4H
               ORG    28H
COPY           DB             6 DUP(?)
;--------------------------
               .CODE
MAIN           PROC   FAR
               MOV    AX,@DATA
               MOV    DS,AX
               MOV    SI,OFFSET DATA_IN    ;SI points to data to be copied
               MOV    DI,OFFSET COPY       ;DI points to copy of data
               MOV    CX,06H               ;loop counter = 6
MOV_LOOP: MOV  AL,[SI]              ;move the next byte from DATA area to AL
               MOV    [DI],AL              ;move the next byte to COPY area
               INC    SI                   ;increment DATA pointer
               INC    DI                   ;increment COPY pointer
               DEC    CX                   ;decrement LOOP counter
               JNZ    MOV_LOOP             ;jump if loop counter not zero
               MOV    AH,4CH               ;set up to return
               INT    21H                  ;return to DOS
MAIN           ENDP
               END    MAIN
```

After the program was assembled and linked, it was run using DEBUG:

```
C>debug prog2-3.exe
-u cs:0 1
1069:0000 B86610      MOV   AX,1066
-d 1066:0 2f
1066:0000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:0010 25 4F 85 1F 2B C4 00 00-00 00 00 00 00 00 00 00 %O..+D........
1066:0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-g

Program terminated normally
-d 1066:0 2f
1066:0000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:0010 25 4F 85 1F 2B C4 00 00-00 00 00 00 00 00 00 00 %O..+D........
1066:0020 00 00 00 00 00 00 00 00-25 4F 85 1F 2B C4 00 00 %O..+D........
-q
C>
```

Program 2-3

Analysis of Program 2-3

The DEBUG example shows the data segment being dumped before the program was run and after to verify that the data was copied and that the program ran successfully. Notice that C4 was coded in the data segments as 0C4. This is required by the assembler to indicate that C is a hex number and not a letter. This is required if the first digit is a hex digit A through F.

This program uses two registers, SI and DI, as pointers to the data items being manipulated. The first is used as a pointer to the data item to be copied and the second as a pointer to the location the data item is to be copied to. With each iteration of the loop, both data pointers are incremented to point to the next byte.

Stack segment definition revisited

One of the primary functions of the DOS operating system is to determine the total amount of RAM installed on the PC and then manage it properly. DOS uses the portion it needs for the operating system and allocates the rest. Since memory requirements vary for different DOS versions, a program cannot dictate the exact physical memory location for the stack or any segment. Since memory management is the responsibility of DOS, it will map Assembly programs into the memory of the PC with the help of LINK.

Although in the DOS environment a program can have multiple code segments and data segments, it is strongly recommended that it have only one stack segment, to prevent RAM fragmentation by the stack. It is the function of LINK to combine all different code and data segments to create a single executable program with a single stack, which is the stack of the system. Various options for segment definition are discussed in Chapter 7 and many of these concepts are explained there.

Review Questions

1. What is the purpose of the INC instruction?
2. What is the purpose of the DEC instruction?
3. In Program 2-1, why does the label AGAIN have a colon after it, whereas the label MAIN does not?
4. State the difference between the following two instructions:

```
MOV  BX,DATA1
MOV  BX,OFFSET DATA1
```

5. State the difference between the following two instructions:

```
ADD  AX,BX
ADD  AX,[BX]
```

SECTION 2.4: CONTROL TRANSFER INSTRUCTIONS

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in the 80x86 to achieve this. This section covers the control transfer instructions available in the 8086 Assembly language. Before that, however, it is necessary to explain the concept of FAR and NEAR as it applies to jump and call instructions.

FAR and NEAR

If control is transferred to a memory location within the current code segment, it is NEAR. This is sometimes called *intrasegment* (within segment). If control is transferred outside the current code segment, it is a FAR or intersegment (between segments) jump. Since the CS:IP registers always point to the address of the next instruction to be executed, they must be updated when a control trans-

fer instruction is executed. In a NEAR jump, the IP is updated and CS remains the same, since control is still inside the current code segment. In a FAR jump, because control is passing outside the current code segment, both CS and IP have to be updated to the new values. In other words, in any control transfer instruction such as jump or call, the IP must be changed, but only in the FAR case is the CS changed, too.

Conditional jumps

Conditional jumps, summarized in Table 2-1, have mnemonics such as JNZ (jump not zero) and JC (jump if carry). In the conditional jump, control is transferred to a new location if a certain condition is met. The flag register is the one that indicates the current condition. For example, with "JNZ label", the processor looks at the zero flag to see if it is raised. If not, the CPU starts to fetch and execute instructions from the address of the label. If ZF = 1, it will not jump but will execute the next instruction below the JNZ.

Table 2-1: 8086 Conditional Jump Instructions

Mnemonic	Condition Tested	"Jump IF ..."
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OR) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

Note:
"Above" and "below" refer to the relationship of two unsigned values; "greater" and "less" refer to the relationship of two signed values.

Short jumps

All conditional jumps are short jumps. In a short jump, the address of the target must be within -128 to +127 bytes of the IP. In other words, the conditional jump is a two-byte instruction: one byte is the opcode of the J condition and the second byte is a value between 00 and FF. An offset range of 00 to FF gives 256 possible addresses; these are split between backward jumps (to -128) and forward jumps (to +127).

In a jump backward, the second byte is the 2's complement of the displacement value. To calculate the target address, the second byte is added to the

IP of the instruction after the jump. To understand this, look at the unassembled code of Program 2-1 for the instruction JNZ AGAIN, repeated below.

```
1067:0000 B86610      MOV   AX,1066
1067:0003 8ED8        MOV   DS,AX
1067:0005 B90500      MOV   CX,0005
1067:0008 BB0000      MOV   BX,0000
1067:000D 0207        ADD   AL,[BX]
1067:000F 43          INC   BX
1067:0010 49          DEC   CX
1067:0011 75FA        JNZ   000D
1067:0013 A20500      MOV   [0005],AL
1067:0016 B44C        MOV   AH,4C
1067:0018 CD21        INT   21
```

The instruction "JNZ AGAIN" was assembled as "JNZ 000D", and 000D is the address of the instruction with the label AGAIN. The instruction "JNZ 000D" has the opcode 75 and the target address FA, which is located at offset addresses 0011 and 0012. This is followed by "MOV SUM,AL", which is located beginning at offset address 0013. The IP value of MOV, 0013, is added to FA to calculate the address of label AGAIN (0013 + FA = 000D) and the carry is dropped. In reality, FA is the 2's complement of −6, meaning that the address of the target is −6 bytes from the IP of the next instruction.

Similarly, the target address for a forward jump is calculated by adding the IP of the following instruction to the operand. In that case the displacement value is positive, as shown next. Below is a portion of a list file showing the opcodes for several conditional jumps.

```
0005  8A 47 02 AGAIN:     MOV   AL,[BX]+2
0008  3C 61               CMP   AL,61H
000A  72 06               JB    NEXT
000C  3C 7A               CMP   AL,7AH
000E  77 02               JA    NEXT
0010  24 DF               AND   AL,ODFH
0012  88 04  NEXT:        MOV   [SI],AL
```

In the program above, "JB NEXT" has the opcode 72 and the target address 06 and is located at IP = 000A and 000B. The jump will be 6 bytes from the next instruction, which is IP = 000C. Adding gives us 000CH + 0006H = 0012H, which is the exact address of the NEXT label. Look also at "JA NEXT", which has 77 and 02 for the opcode and displacement, respectively. The IP of the following instruction, 0010, is added to 02 to get 0012, the address of the target location.

It must be emphasized that regardless of whether the jump is forward or backward, for conditional jumps the address of the target address can never be more than −128 to + 127 bytes away from the IP associated with the instruction following the jump (− for the backward jump and + for the forward jump). If any attempt is made to violate this rule, the assembler will generate a "relative jump out of range" message. These conditional jumps are sometimes referred to as SHORT jumps.

Unconditional jumps

"JMP label" is an unconditional jump in which control is transferred unconditionally to the target location label. The unconditional jump can take the following forms:

1. SHORT JUMP, which is specified by the format "JMP SHORT label". This is a jump in which the address of the target location is within -128 to +127 bytes of memory relative to the address of the current IP. In this case, the opcode is EB and the operand is 1 byte in the range 00 to FF. The operand byte is added to the current IP to calculate the target address. If the jump is backward, the operand is in 2's complement. This is exactly like the J condition case. Coding the directive "short" makes the jump more efficient in that it will be assembled into a 2-byte instruction instead of a 3-byte instruction.
2. NEAR JUMP, which is the default, has the format "JMP label". This is a near jump (within the current code segment) and has the opcode E9. The target address can be any of the addressing modes of direct, register, register indirect, or memory indirect:
   (a) Direct JUMP is exactly like the short jump explained earlier, except that the target address can be anywhere in the segment within the range +32767 to -32768 of the current IP.
   (b) Register indirect JUMP; the target address is in a register. For example, in "JMP BX", IP takes the value BX.
   (c) Memory indirect JMP; the target address is the contents of two memory locations pointed at by the register. Example: "JMP [DI]" will replace the IP with the contents of memory locations pointed at by DI and DI+1.
3. FAR JUMP which has the format "JMP FAR PTR label". This is a jump out of the current code segment, meaning that not only the IP but also the CS is replaced with new values.

CALL statements

Another control transfer instruction is the CALL instruction, which is used to call a procedure. CALLs to procedures are used to perform tasks that need to be performed frequently. This makes a program more structured. The target address could be in the current segment, in which case it will be a NEAR call or outside the current CS segment, which is a FAR call. To make sure that after execution of the called subroutine the microprocessor knows where to come back, the microprocessor automatically saves the address of the instruction following the call on the stack. It must be noted that in the NEAR call only the IP is saved on the stack, and in a FAR call both CS and IP are saved. When a subroutine is called, control is transferred to that subroutine and the processor saves the IP (and CS in the case of a FAR call) and begins to fetch instructions from the new location. After finishing execution of the subroutine, for control to be transferred back to the caller, the last instruction in the called subroutine must be RET (return). In the same way that the assembler generates different opcode for FAR and NEAR calls, the opcode for the RET instruction in the case of NEAR and FAR is different, as well. For NEAR calls, the IP is restored; for FAR calls, both CS and IP are restored. This will ensure that control is given back to the caller. As an example, assume that SP = FFFEH and the following code is a portion of the program unassembled in DEBUG:

```
12B0:0200 BB1295  MOV BX,9512
12B0:0203 E8FA00  CALL 0300
12B0:0206 B82F14  MOV AX,142F
```

Since the CALL instruction is a NEAR call, meaning that it is in the same code segment (different IP, same CS), only IP is saved on the stack. In this case, the IP address of the instruction after the call is saved on the stack as shown in Figure 2-5. That IP will be 0206, which belongs to the "MOV AX,142F" instruction.

The last instruction of the called subroutine must be a RET instruction which directs the CPU to POP the top 2 bytes of the stack into the IP and resume executing at offset address 0206. For this reason, the number of PUSH and POP instructions (which alter the SP) must match. In other words, for every PUSH there must be a POP.

```
12B0:0300  53  PUSH BX
12B0:0301  ...  ..... ..
.....  ...  .....  ..
12B0:0309  5B  POP BX
12B0:030A  C3  RET
```

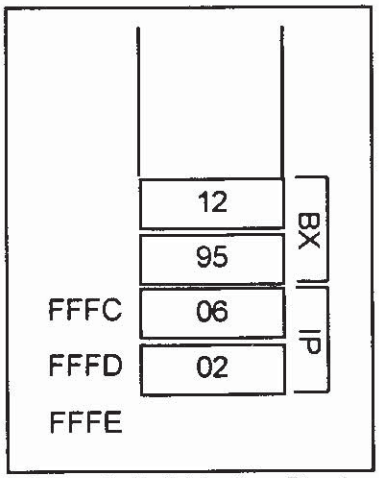

Figure 2-5. IP in the Stack

Assembly language subroutines

In Assembly language programming it is common to have one main program and many subroutines to be called from the main program. This allows you to make each subroutine into a separate module. Each module can be tested separately and then brought together, as will be shown in Chapter 7. The main program is the entry point from DOS and is FAR, as explained earlier, but the subroutines called within the main program can be FAR or NEAR. Remember that NEAR routines are in the same code segment, while FAR routines are outside the current code segment. If there is no specific mention of FAR after the directive PROC, it defaults to NEAR, as shown in Figure 2-6. From now on, all code segments will be written in that format.

Rules for names in Assembly language

By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain. There are several rules that names must follow. First, each label name must be unique. The names used for labels in Assembly language programming consist of alphabetic letters in both upper and lower case, the digits 0 through 9, and the special characters question mark (?), period (.), at (@), underline (_), and dollar sign ($). The first character of the name must be an alphabetic character or special character. It cannot be a digit. The period can only be used as the first character, but this is not recommended since later versions of MASM have several reserved words that begin with a period. Names may be up to 31 characters long. A list of reserved words is given at the end of Appendix C.

Review Questions

1. If control is transferred outside the current code segment, is it NEAR or FAR?
2. If a conditional jump is not taken, what is the next instruction to be executed?
3. In calculating the target address to jump to, a displacement is added to the contents of register _____.
4. What is the advantage in coding the operator "SHORT" in an unconditional jump?
5. A(n) _____ jump is within -128 to +127 bytes of the current IP. A(n) _____ jump is within the current code segment, whereas a(n) _____ jump is outside the current code segment.
6. How does the CPU know where to return to after executing a RET?
7. Describe briefly the function of the RET instruction.
8. State why the following label names are invalid.
   (a) GET.DATA (b) 1_NUM (c) TEST-DATA (d) RET

```
                .CODE
MAIN            PROC  FAR           ;THIS IS THE ENTRY POINT FOR DOS
                MOV   AX,@DATA
                MOV   DS,AX
                CALL  SUBR1
                CALL  SUBR2
                CALL  SUBR3
                MOV   AH,4CH
                INT   21H
MAIN            ENDP
;-------------------------------------------
SUBR1           PROC
                ...
                ...
                RET
SUBR1           ENDP
;-------------------------------------------
SUBR2           PROC
                ...
                ...
                RET
SUBR2           ENDP
;-------------------------------------------
SUBR3           PROC
                ...
                ...
                RET
SUBR3           ENDP
;-------------------------------------------
                END     MAIN          ;THIS IS THE EXIT POINT
```

Figure 2-6. Shell of Assembly Language Subroutines

SECTION 2.5: DATA TYPES AND DATA DEFINITION

The assembler supports all the various data types of the 80x86 microprocessor by providing data directives that define the data types and set aside memory for them. In this section we study these directives and how they are used to represent different data types of the 80x86. The application of these directives becomes clearer in the context of examples in subsequent chapters.

80x86 data types

The 8088/86 microprocessor supports many data types, but none are longer than 16 bits wide since the size of the registers is 16 bits. It is the job of the programmer to break down data larger than 16 bits (0000 to FFFFH, or 0 to 65535 in decimal) to be processed by the CPU. Many of these programs are shown in Chapter 3. The data types used by the 8088/86 can be 8-bit or 16-bit, positive or negative. If a number is less than 8 bits wide, it still must be coded as an 8-bit register with the higher digits as zero. Similarly, if the number is less than 16 bits wide it must use all 16 bits, with the rest being 0s. For example, the number 5 is only 3 bits wide (101) in binary, but the 8088/86 will accept it as 05 or "0000 0101" in binary. The number 514 is "10 0000 0010" in binary, but the 8088/86 will accept it as "0000 0010 0000 0010" in binary. The discussion of signed numbers is postponed until later chapters since their representation and application are unique.

Assembler data directives

All the assemblers designed for the 80x86 (8088, 8086, 80188, 80186, 80286, 80386, 80386SX, 80486, and Pentium) microprocessors have standardized the directives for data representation. The following are some of the data

directives used by the 80x86 microprocessor and supported by all software and hardware vendors of IBM PCs and compatibles.

ORG (origin)

ORG is used to indicate the beginning of the offset address. The number that comes after ORG can be either in hex or in decimal. If the number is not followed by H, it is decimal and the assembler will convert it to hex. Although the ORG directive is used extensively in this book in the data segment to separate fields of data to make it more readable for the student, it can also be used for the offset of the code segment (IP).

DB (define byte)

The DB directive is one of the most widely used data directives in the assembler. It allows allocation of memory in byte-sized chunks. This is indeed the smallest allocation unit permitted. DB can be used to define numbers in decimal, binary, hex, and ASCII. For decimal, the D after the decimal number is optional, but using B (binary) and H (hexadecimal) for the others is required. Regardless of which one is used, the assembler will convert them into hex. To indicate ASCII, simply place the string in single quotation marks ('like this'). The assembler will assign the ASCII code for the numbers or characters automatically. DB is the only directive that can be used to define ASCII strings larger than two characters; therefore, it should be used for all ASCII data definitions. Following are some DB examples:

```
DATA1   DB   25              ;DECIMAL
DATA2   DB   10001001B       ;BINARY
DATA3   DB   12H             ;HEX
        ORG  0010H
DATA4   DB   '2591'          ;ASCII NUMBERS
        ORG  0018H
DATA5   DB   ?               ;SET ASIDE A BYTE
        ORG  0020H
DATA6   DB   'My name is Joe' ;ASCII CHARACTERS
```

```
0000 19                     DATA1 DB   25               ;DECIMAL
0001 89                     DATA2 DB   10001001B        ;BINARY
0002 12                     DATA3 DB   12H              ;HEX
0010                              ORG  0010H
0010 32 35 39 31            DATA4 DB   '259'            ;ASCII NUMBERS
0018                              ORG  0018H
0018 00                     DATA5 DB   ?                ;SET ASIDE A BYTE
0020                              ORG  0020H
0020 4D 79 20 6E 61 6D      DATA6 DB   'My name is Joe' ;ASCII CHARACTERS
     65 20 69 73 20 4A
     6F 65
```

List File for DB Examples

Either single or double quotes can be used around ASCII strings. This can be useful for strings which should contain a single quote such as "O'Leary".

DUP (duplicate)

DUP is used to duplicate a given number of characters. This can avoid a lot of typing. For example, contrast the following two methods of filling six memory locations with FFH:

```
        ORG   0030H
DATA7   DB    0FFH,0FFH,0FFH,0FFH,0FFH,0FFH  ;FILL 6 BYTES WITH FF
        ORG   38H
DATA8   DB    6 DUP(0FFH)     ;FILL 6 BYTES WITH FF
; the following reserves 32 bytes of memory with no initial value given
        ORG   40H
DATA9   DB    32 DUP (?)      ;SET ASIDE 32 BYTES
;DUP can be used inside another DUP
; the following fills 10 bytes with 99
DATA10  DB    5 DUP (2 DUP (99)) ;FILL 10 BYTES WITH 99
```

```
0030                                    ORG   0030H
0030 FF FF FF FF FF FF          DATA7   DB    0FFH,0FFH,0FFH,0FFH,0FFH,0FFH  ; 6 BYTES = FF
0038                                    ORG   38H
0038 0006[                      DATA8   DB    6 DUP(0FFH)   ;FILL 6 BYTES WITH FF
          FF
                ]
0040                                    ORG   40H
0040 0020 [                     DATA9   DB    32 DUP (?)    ;SET ASIDE 32 BYTES
      ??
                ]
0060                                    ORG   60H
0060 0005[                      DATA10  DB    5 DUP (2 DUP (99))    ;FILL 10 BYTES WITH 99
          0002[
                63
                ]               ]
```

List File for DUP Examples

DW (define word)

DW is used to allocate memory 2 bytes (one word) at a time. DW is used widely in the 8088/8086 and 80286 microprocessors since the registers are 16 bits wide. The following are some examples of DW:

```
        ORG   70H
DATA11  DW    954                   ;DECIMAL
DATA12  DW    100101010100B         ;BINARY
DATA13  DW    253FH                 ;HEX
        ORG   78H
DATA14  DW    9,2,7,0CH,00100000B,5,'HI'  ;MISC. DATA
DATA15  DW    8 DUP (?)                   ;SET ASIDE 8 WORDS
```

```
0070                            ORG     70H
0070 03BA                       DATA11  DW  954                   ;DECIMAL
0072 0954                       DATA12  DW  100101010100B         ;BINARY
0074 253F                       DATA13  DW  253FH                 ;HEX
0078                            ORG     78H
0078 0009 0002 0007 000C        DATA14  DW  9,2,7,0CH,00100000B,5,'HI'  ;MISC. DATA
     0020 0005 4849
0086 0008[                      DATA15  DW  8 DUP (?)             ;SET ASIDE 8 WORDS
          ????                  ]
```

List File for DW Examples

EQU (equate)

This is used to define a constant without occupying a memory location. EQU does not set aside storage for a data item but associates a constant value with a data label so that when the label appears in the program, its constant value will be substituted for the label. EQU can also be used outside the data segment, even in the middle of a code segment. Using EQU for the counter constant in the immediate addressing mode:

```
COUNT  EQU 25
```

When executing the instructions "MOV CX,COUNT", the register CX will be loaded with the value 25. This is in contrast to using DB:

```
COUNT  DB 25
```

When executing the same instruction "MOV CX,COUNT" it will be in the direct addressing mode. Now what is the real advantage of EQU? First, note that EQU can also be used in the data segment:

```
COUNT     EQU     25
COUNTER1  DB      COUNT
COUNTER2  DB      COUNT
```

Assume that there is a constant (a fixed value) used in many different places in the data and code segments. By the use of EQU, one can change it once and the assembler will change all of them, rather than making the programmer try to find every location and correct it.

DD (define doubleword)

The DD directive is used to allocate memory locations that are 4 bytes (two words) in size. Again, the data can be in decimal, binary, or hex. In any case the data is converted to hex and placed in memory locations according to the rule of low byte to low address and high byte to high address. DD examples are:

```
          ORG  00A0H
DATA16    DD   1023                      ;DECIMAL
DATA17    DD   10001001011001011100B     ;BINARY
DATA18    DD   5C2A57F2H                 ;HEX
DATA19    DD   23H,34789H,65533
```

```
00A0
00A0  000003FF                DATA16  DD  1023                      ;DECIMAL
00A4  0008965C                DATA17  DD  10001001011001011100B     ;BINARY
00A8  5C2A57F2                DATA18  DD  5C2A57F2H                 ;HEX
00AC 00000023 00034789        DATA19  DD  23H,34789H,65533
      0000FFFD
```

List File for DD Examples

DQ (define quadword)

DQ is used to allocate memory 8 bytes (four words) in size. This can be used to represent any variable up to 64 bits wide:

```
          ORG  00C0H
DATA20    DQ   4523C2H      ;HEX
DATA21    DQ   'HI'         ;ASCII CHARACTERS
DATA22    DQ   ?            ;NOTHING
```

```
00C0
00C0  C223450000000000    DATA20  DQ  4523C2H      ;HEX
00C8  4948000000000000    DATA21  DQ  'HI'         ;ASCII CHARACTERS
00D0  0000000000000000    DATA22  DQ  ?            ;NOTHING
```

List File for DQ Examples

DT (define ten bytes)

DT is used for memory allocation of packed BCD numbers. The application of DT will be seen in the multibyte addition of BCD numbers in Chapter 3. For now, observe how they are located in memory. Notice that the "H" after the data is not needed. This allocates 10 bytes, but a maximum of 18 digits can be entered.

```
          ORG  00E0H
DATA23    DT   867943569829  ;BCD
DATA24    DT   ?             ;NOTHING
```

```
00E0                              ORG 00E0H
00E0 299856437986000000  DATA23 DT   867943569829        ;BCD
     00
00EA 000000000000000000  DATA24 DT   ?                   ;NOTHING
     00
```

List File for DT Examples

DT can also be used to allocate 10-byte integers by using the "D" option:

```
DEC  DT  65535d     ;the assembler will convert the decimal
                    ;number to hex and store it
```

Figure 2-7 shows the memory dump of the data section, including all the examples in this section. It is essential to understand the way operands are stored in memory. Looking at the memory dump shows that all of the data directives use the little endian format for storing data, meaning that the least significant byte is located in the memory location of the lower address and the most significant byte resides in the memory location of the higher address. For example, look at the case of "DATA20 DQ 4523C2", residing in memory starting at offset 00C0H. C2, the least significant byte, is in location 00C0, with 23 in 00C1, and 45, the most significant byte, in 00C2. It must also be noted that for ASCII data, only the DB directive can be used to define data of any length, and the use of DD, DQ, or DT directives for ASCII strings of more than 2 bytes gives an assembly error. When DB is used for ASCII numbers, notice how it places them backwards in memory. For example, see "DATA4 DB '2591'" at origin 10H: 32, ASCII for 2, is in memory location 10H; 35, ASCII for 5, is in 11H; and so on.

```
-D 1066:0 100
1066:0000 19 89 12 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:0010 32 35 39 31 00 00 00 00-00 00 00 00 00 00 00 00 2591............
1066:0020 4D 79 20 6E 61 6D 65 20-69 73 20 4A 6F 65 00 00 My name is Joe..
1066:0030 FF FF FF FF FF FF 00 00-FF FF FF FF FF FF 00 00 ................
1066:0040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:0060 63 63 63 63 63 63 63 63-63 63 00 00 00 00 00 00 cccccccccc......
1066:0070 BA 03 54 09 3F 25 00 00-09 00 02 00 07 00 0C 00 :.T.?%..........
1066:0080 20 00 05 00 4F 48 00 00-00 00 00 00 00 00 00 00  ...OH...........
1066:0090 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:00A0 FF 03 00 00 5C 96 08 00-F2 57 2A 5C 23 00 00 00 ....\...rW*\#...
1066:00B0 89 47 03 00 FD FF 00 00-00 00 00 00 00 00 00 00 B#E.....IH.....
1066:00C0 C2 23 45 00 00 00 00 00-49 48 00 00 00 00 00 00 ................
1066:00D0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:00E0 29 98 56 43 79 86 00 00-00 00 00 00 00 00 00 00 9.VCy6..........
```

Figure 2-7. DEBUG Dump of Data Segment

Review Questions

1. The _____ directive is always used for ASCII strings longer than 2 bytes.
2. How many bytes are defined by the following?
   ```
   DATA_1    DB    6 DUP (4 DUP (0FFH))
   ```
3. Do the following two data segment definitions result in the same storage in bytes at offset 10H and 11H? If not, explain why.

   ```
            ORG  10H
   DATA_1   DB   72
   DATA_2   DB   04H
   ```

   ```
            ORG  10H
   DATA_1   DW   7204H
   ```

4. The DD directive is used to allocate memory locations that are ____ bytes in length. The DQ directive is used to allocate memory locations that are _____ bytes in length.
5. State briefly the purpose of the ORG directive.
6. What is the advantage in using the EQU directive to define a constant value?
7. How many bytes are set aside by each of the following directives?
   (a) ASC_DATA DB '1234' (b) HEX_DATA DW 1234H
8. Does the little endian storage convention apply to the storage of ASCII data?

SECTION 2.6: FULL SEGMENT DEFINITION

The way that segments have been defined in the programs above is a newer definition referred to as *simple segment definition*. It is supported by Microsoft's MASM 5.0 and higher, Borland's TASM version 1 and higher, and many other compatible assemblers. The older, more traditional definition is called the *full segment definition*. Although the simplified segment definition is much easier to understand and use, especially for beginners, it is essential to master full segment definition since many older programs use it.

Segment definition

The "SEGMENT" and "ENDS" directives indicate to the assembler the beginning and ending of a segment and have the following format:

```
label   SEGMENT     [options]
        ;place the statements belonging to this segment here
label   ENDS
```

The label, or name, must follow naming conventions (see the end of Section 2.4) and must be unique. The [options] field gives important information to the assembler for organizing the segment, but is not required. The ENDS label must be the same label as in the SEGMENT directive. In the full segment definition, the ".MODEL" directive is not used. Further, the directives ".STACK", ".DATA", and ".CODE" are replaced by SEGMENT and ENDS directives that surround each segment. Figure 2-8 shows the full segment definition and simplified format, side by side. This is followed by Programs 2-2 and 2-3, rewritten using the full segment definition.

Stack segment definition

The stack segment shown below contains the line: "DB 64 DUP (?)" to reserve 64 bytes of memory for the stack. The following three lines in full segment definition are comparable to ".STACK 64" in simple definition:

```
STSEG       SEGMENT      ;the "SEGMENT" directive begins the segment
            DB 64 DUP (?) ;this segment contains only one line
STSEG       ENDS         ;the "ENDS" segment ends the segment
```

```
;FULL SEGMENT DEFINITION                ;SIMPLIFIED FORMAT
        ;—— stack segment ——                    .MODEL  SMALL
        name1  SEGMENT                          .STACK   64
               DB      64 DUP (?)       ;
        name1  ENDS                     ;
        ;—— data segment ——             ;————————————————————
        name2  SEGMENT                          . DATA
        ;data definitions are placed here ;data definitions are placed here
        name2  ENDS                     ;
        ;—— code segment ——             ;————————————————————
        name3  SEGMENT                          .CODE
        MAIN   PROC  FAR                MAIN    PROC  FAR
               ASSUME ...                       MOV   AX,@DATA
               MOV    AX,name2                  MOV   DS,AX
               MOV    DS,AX                     ...
               ...                              ...
        MAIN   ENDP                     MAIN    ENDP
        name3  ENDS                              END   MAIN
               END    MAIN
```

Figure 2-8. Full versus Simplified Segment Definition

```
TITLE          PURPOSE: ADDS 4 WORDS OF DATA
PAGE  60,132
STSEG          SEGMENT
               DB      32 DUP (?)
STSEG          ENDS
DTSEG          SEGMENT
DATA_IN        DW      234DH,1DE6H,3BC7H,566AH
               ORG     10H
SUM            DW      ?
DTSEG          ENDS
;————————————————————
CDSEG          SEGMENT
MAIN           PROC  FAR
               ASSUME CS:CDSEG,DS:DTSEG,SS:STSEG
               MOV   AX,DTSEG
               MOV   DS,AX
               MOV   CX,04              ;set up loop counter CX=4
               MOV   DI,OFFSET DATA_IN  ;set up data pointer DI
               MOV   BX,00              ;initialize BX
ADD_LP:        ADD   BX,[DI]            ;add contents pointed at by [DI] to BX
               INC   DI                 ;increment DI twice
               INC   DI                 ;to point to next word
               DEC   CX                 ;decrement loop counter
               JNZ   ADD_LP             ;jump if loop counter not zero
               MOV   SI,OFFSET SUM      ;load pointer for sum
               MOV   [SI],BX            ;store in data segment
               MOV   AH,4CH             ;set up return
               INT   21H                ;return to DOS
MAIN           ENDP
CDSEG          ENDS
               END   MAIN
```

Program 2-2, rewritten with full segment definition

```
TITLE       PURPOSE: TRANSFERS 6 BYTES OF DATA
PAGE        60,132
STSEG       SEGMENT
            DB          32 DUP (?)
STSEG       ENDS
;-------------------------------
DTSEG       SEGMENT
            ORG   10H
DATA_IN     DB          25H,4FH,85H,1FH,2BH,0C4H
            ORG   28H
COPY        DB          6 DUP(?)
DTSEG       ENDS
;-------------------------------
CDSEG       SEGMENT
MAIN        PROC  FAR
            ASSUME CS:CDSEG,DS:DTSEG,SS:STSEG
            MOV   AX,DTSEG
            MOV   DS,AX
            MOV   SI,OFFSET DATA_IN   ;SI points to data to be copied
            MOV   DI,OFFSET COPY      ;DI points to copy of data
            MOV   CX,06H              ;loop counter = 6
MOV_LOOP:   MOV   AL,[SI]             ;move the next byte from DATA area to AL
            MOV   [DI],AL             ;move the next byte to COPY area
            INC   SI                  ;increment DATA pointer
            INC   DI                  ;increment COPY pointer
            DEC   CX                  ;decrement LOOP counter
            JNZ   MOV_LOOP            ;jump if loop counter not zero
            MOV   AH,4CH              ;set up to return
            INT   21H                 ;return to DOS
MAIN        ENDP
CDSEG       ENDS
            END   MAIN
```

Program 2-3, rewritten with full segment definition

Data segment definition

In full segment definition, the SEGMENT directive names the data segment and must appear before the data. The ENDS segment marks the end of the data segment:

```
DTSEG       SEGMENT     ;the SEGMENT directive begins the segment
            ;define your data here
DTSEG       ENDS        ;the ENDS segment ends the segment
```

Code segment definition

The code segment also begins with a SEGMENT directive and ends with a matching ENDS directive:

```
CDSSEG      SEGMENT     ;the SEGMENT directive begins the segment
            ;your code is here
CDSEG       ENDS        ;the ENDS segment ends the segment
```

In full segment definition, immediately after the PROC directive is the ASSUME directive, which associates segment registers with specific segments by

assuming that the segment register is equal to the segment labels used in the program. If an extra segment had been used, ES would also be included in the ASSUME statement. The ASSUME statement is needed because a given Assembly language program can have several code segments, one or two or three or more data segments and more than one stack segment, but only one of each can be addressed by the CPU at a given time since there is only one of each of the segment registers available inside the CPU. Therefore, ASSUME tells the assembler which of the segments defined by the SEGMENT directives should be used. It also helps the assembler to calculate the offset addresses from the beginning of that segment. For example, in "MOV AL,[BX]" the BX register is the offset of the data segment.

Upon transfer of control from DOS to the program, of the three segment registers, only CS and SS have the proper values. The DS value (and ES, if used) must be initialized by the program. This is done as follows in full segment definition:

```
MOV AX,DTSEG        ;DTSEG is the label for the data segment
MOV DS,AX
```

SECTION 2.7: EXE VS. COM FILES

All program examples so far were designed to be assembled and linked into EXE files. This section looks at the COM file, which like the EXE file contains the executable machine code and can be run at the DOS level. At the end of this section, the process of conversion from one file to the other is shown.

Why COM files?

There are occasions where, due to a limited amount of memory, one needs to have very compact code. This is the time when the COM file is useful. The fact that the EXE file can be of any size is one of the main reasons that EXE files are used so widely. On the other hand, COM files are used because of their compactness since they cannot be greater than 64K bytes. The reason for the 64K-byte limit is that the COM file must fit into a single segment, and since in the 80x86 the size of a segment is 64K bytes, the COM file cannot be larger than 64K. To limit the size of the file to 64K bytes requires defining the data inside the code segment and also using an area (the end area) of the code segment for the stack. One of the distinguishing features of the COM file program is the fact that in contrast to the EXE file, it has no separate data segment definition. One can summarize the differences between COM and EXE files as shown in Table 2-2.

Table 2-2: EXE vs. COM File Format

EXE File	COM File
unlimited size	maximum size 64K bytes
stack segment is defined	no stack segment definition
data segment is defined	data segment defined in code segment
code, data defined at any offset address	code and data begin at offset 0100H
larger file (takes more memory)	smaller file (takes less memory)

Another reason for the difference in the size of the EXE and COM files is the fact that the COM file does not have a header block. The header block, which occupies 512 bytes of memory, precedes every EXE file and contains information such as size, address location in memory, and stack address of the EXE module.

Program 2-4, written in COM format, adds two words of data and saves the result. This format is very similar to many programs written on the 8080/85 microprocessors, the generation before the 8088/86. This format of first having the code and then the data takes longer to assemble; therefore, it is strongly recommended to put the data first and then the code, but the program must bypass the data area by the use of a JUMP instruction, as shown in Program 2-5.

```
TITLE   PROG2-4 COM PROGRAM TO ADD TWO WORDS
PAGE   60,132
CODSG          SEGMENT
               ORG    100H
               ASSUME  CS:CODSG,DS:CODSG,ES:CODSG
;——THIS IS THE CODE AREA
PROGCODE   PROC  NEAR
               MOV    AX,DATA1        ;move the first word into AX
               MOV    SUM,AX          ;move the sum
               MOV    AH,4CH          ;return to DOS
               INT    21H
PROGCODE   ENDP
;——THIS IS THE DATA AREA
DATA1          DW     2390
DATA2          DW     3456
SUM            DW     ?
;-------------------------------
CODSG          ENDS
               END    PROGCODE
```

Program 2-4

```
TITLE          PROG2-5 COM PROGRAM TO ADD TWO WORDS
PAGE           60,132
CODSG          SEGMENT
               ASSUME  CS:CODSG,DS:CODSG,ES:CODSG
               ORG    100H
START:         JMP    PROGCODE    ;go around the data area
;——THIS IS THE DATA AREA
DATA1          DW     2390
DATA2          DW     3456
SUM            DW     ?
;——THIS IS THE CODE AREA
PROGCODE:      MOV    AX,DATA1    ;move the first word into AX
               ADD    AX,DATA1    ;add the second word
               MOV    SUM,AX      ;move the sum
               MOV    AH,4CH
               INT    21H
;-----------------------------------
CODSB          ENDS
               END    START
```

Program 2-5

Converting from EXE to COM

For the sake of memory efficiency, it is often desirable to convert an EXE file into a COM file. The source file must be changed to the COM format shown above, then assembled and linked as usual. Then it must be input to a utility program called EXE2BIN that comes with DOS. Its function is to convert the EXE file to a COM file. For example, to convert an EXE file called PROG1.EXE in drive A, assuming that the EXE2BIN utility is in drive C, do the following:

```
C>EXE2BIN A:PROG1,A:PROG1.COM
```

Notice that there is no extension of EXE for PROG1 since it is assumed that one is converting an EXE file. Keep in mind that for a program to be converted into a COM file, it must be in the format shown in Programs 2-4 and 2-5.

SUMMARY

An Assembly language program is composed of a series of statements that are either instructions or pseudo-instructions, also called directives. Instructions are translated by the assembler into machine code. Pseudo-instructions are not translated into machine code: They direct the assembler in how to translate the instructions into machine code. The statements of an Assembly language program are grouped into segments. Other pseudo-instructions, often called data directives, are used to define the data in the data segment. Data can be allocated in units ranging in size from byte, word, doubleword, and quadword to 10 bytes at a time. The data can be in binary, hex, decimal, or ASCII.

The flow of a program proceeds sequentially, from instruction to instruction, unless a control transfer instruction is executed. The various types of control transfer instructions in Assembly language include conditional and unconditional jumps, and call instructions.

PROBLEMS

1. Rewrite Program 2-3 to transfer one word at a time instead of one byte.
2. List the steps in getting a ready-to-run program.
3. Which program produces the ".exe" file?
4. Which program produces the ".obj" file?
5. True or false: The ".lst" file is produced by the assembler regardless of whether or not the programmer wants it.
6. The source program file must have the ".asm" extension in some assemblers. such as MASM. Is this true for the assembler you are using?
7. Circle one: The linking process comes (after, before) assembling.
8. In some applications it is common practice to save all registers at the beginning of a subroutine. Assume that SP = 1288H before a subroutine CALL. Show the contents of the stack pointer and the exact memory contents of the stack after PUSHF for the following:

```
1132:0450 CALL PROC1
1132:0453 INC BX

 PROC1     PROC
           PUSH AX
           PUSH BX
           PUSH CX
           PUSH DX
           PUSH  SI
           PUSH DI
           PUSHF
           ....
 PROC1     ENDP
```

9. To restore the original information inside the CPU at the end of a CALL to a subroutine, the sequence of POP instructions must follow a certain order. Write the sequence of POP instructions that will restore the information in Problem 8. At each point, show the contents of the SP.

10. When a CALL is executed, how does the CPU know where to return?
11. In a FAR CALL, _____ and _____ are saved on the stack, whereas in a NEAR CALL, _____ is saved on the stack.
12. Compare the number of bytes of stack taken due to NEAR and FAR CALLs.
13. Find the contents of the stack and stack pointer after execution of the CALL instruction shown next.

```
CS  : IP
2450:673A CALL SUM
2450:673D DEC AH
```

SUM is a near procedure. Assume the value SS:1296 right before the execution of CALL.

14. The following is a section of BIOS of the IBM PC which is described in detail in Chapter 3. All the jumps below are short jumps, meaning that the labels are in the range -128 to +127.

```
IP    Code
E06C 733F   JNC  ERROR1
...  ...  ...
E072 7139   JNO  ERROR1
...  ...  ...
E08C 8ED8  C8:  MOV  DS,AX
...  ...  ...
E0A7 EBE3   JMP  C8
...  ...  ...
E0AD F4  ERROR1: HLT
```

Verify the address calculations of:
(a) JNC ERROR1 (b) JNO ERROR1 (c) JMP C8

15. Find the precise offset location of each ASCII character or data in the following:

```
        ORG   20H
DATA1   DB    '1-800-555-1234'
        ORG   40H
DATA2   DB    'Name: John Jones'
        ORG   60H
DATA3   DB    '5956342'
        ORG   70H
DATA4   DW    2560H,1000000000110B
DATA5   DW    49
        ORG   80H
DATA6   DD    25697F6EH
DATA7   DQ    9E7BA21C99F2H
        ORG   90H
DATA8   DT    439997924999828
DATA9   DB    6 DUP (0EEH)
```

16. The following program contains some errors. Fix the errors and make the program run correctly. Verify it through the DEBUG program. This program adds four words and saves the result.

```
TITLE  PROBLEM (EXE)  PROBLEM 16 PROGRAM
PAGE   60,132
       .MODEL SMALL
       .STACK 32
;-------------------------
       .DATA
DATA   DW    234DH,DE6H,3BC7H,566AH
       ORG   10H
SUM    DW    ?
;-------------------------
       .CODE
START: PROC  FAR
       MOV   AX,DATA
       MOV   DS,AX
       MOV   CX,04      ;SET UP LOOP COUNTER CX=4
```

```
        MOV   BX,0              ;INITIALIZE BX TO ZERO
        MOV   DI,OFFSET DATA    ;SET UP DATA POINTER BX
LOOP1:  ADD   BX,[DI]  ;ADD CONTENTS POINTED AT BY [DI] TO BX
        INC   DI                ;INCREMENT DI
        JNZ   LOOP1             ;JUMP IF COUNTER NOT ZERO
        MOV   SI,OFFSET RESULT  ;LOAD POINTER FOR  RESULT
        MOV   [SI],BX           ;STORE THE SUM
        MOV   AH,4CH
        INT   21H
START   ENDP
        END   STRT
```

ANSWERS TO REVIEW QUESTIONS

SECTION 2.1: DIRECTIVES AND A SAMPLE PROGRAM

1. Pseudo-instructions direct the assembler as to how to assemble the program.
2. Instructions, pseudo-instructions or directives
3.
```
                .MODEL SMALL
                .STACK 64
                .DATA
HIGH_DAT        DB 95
                .CODE
                START  PROC FAR
                MOV AX,@DATA
                MOV DS,AX
                MOV AH,HIGH_DAT
                MOV BH,AH
                MOV DL,BH
                MOV AH,4CH
                INT 21H
START           ENDP
                END START
```
4. (1) there is no ENORMOUS model
   (2) ENDP label does not match label for PROC directive
   (3) .CODE and .DATA directives need to be switched
   (4) "MOV AX,DATA" should be "MOV AX,@DATA"
   (5) "MOV DS,@DATA" should be "MOV DS,AX"
   (6) END must have the entry point label "MAIN"

SECTION 2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

1. (a) MASM must have the ".asm" file as input
   (b) LINK must have the ".obj" file as input
2. Editor outputs : (b) .asm
   Assembler outputs: (a) .obj, (d) .lst, and (e) .crf files
   Linker outputs: (c) .exe and (f) .map files

SECTION 2.3: MORE SAMPLE PROGRAMS

1. increments the operand, that is, it causes 1 to be added to the operand
2. decrements the operand, that is, it causes 1 to be subtracted from the operand
3. a colon is required after labels referring to instructions; colons are not placed after labels for directives
4. the first moves the contents of the word beginning at offset DATA1, the second moves the offset address of DATA1
5. the first adds the contents of BX to AX, the second adds the contents of the memory location at offset BX.

SECTION 2.4: CONTROL TRANSFER INSTRUCTIONS

1. far 2. the instruction right below the jump 3. IP
4. the machine code for the instruction will take up 1 less byte
5. short, near, far
6. the contents of CS and IP were stored on the stack when the call was executed
7. it restores the contents of CS:IP and returns control to the instruction immediately following the CALL
8. (a) GET.DATA, invalid because "." is only allowed as the first character
   (b) 1_NUM, because the first character cannot be a number
   (c) TEST-DATA, because "-" is not allowed
   (d) RET, is a reserved word

SECTION 2.5: DATA TYPES AND DATA DEFINITION

1. DB 2. 24
3. no because of the little endian storage conventions, which will cause the word "7204H" to be stored with the lower byte (04) at offset 10H and the upper byte at offset 11H; DB allocates each byte as it is defined
4. 4, 8 5. it is used to assign the offset address
6. if the value is to be changed later, it can be changed in one place instead of at every occurrence
7. (a) 4 (b) 2 8. no