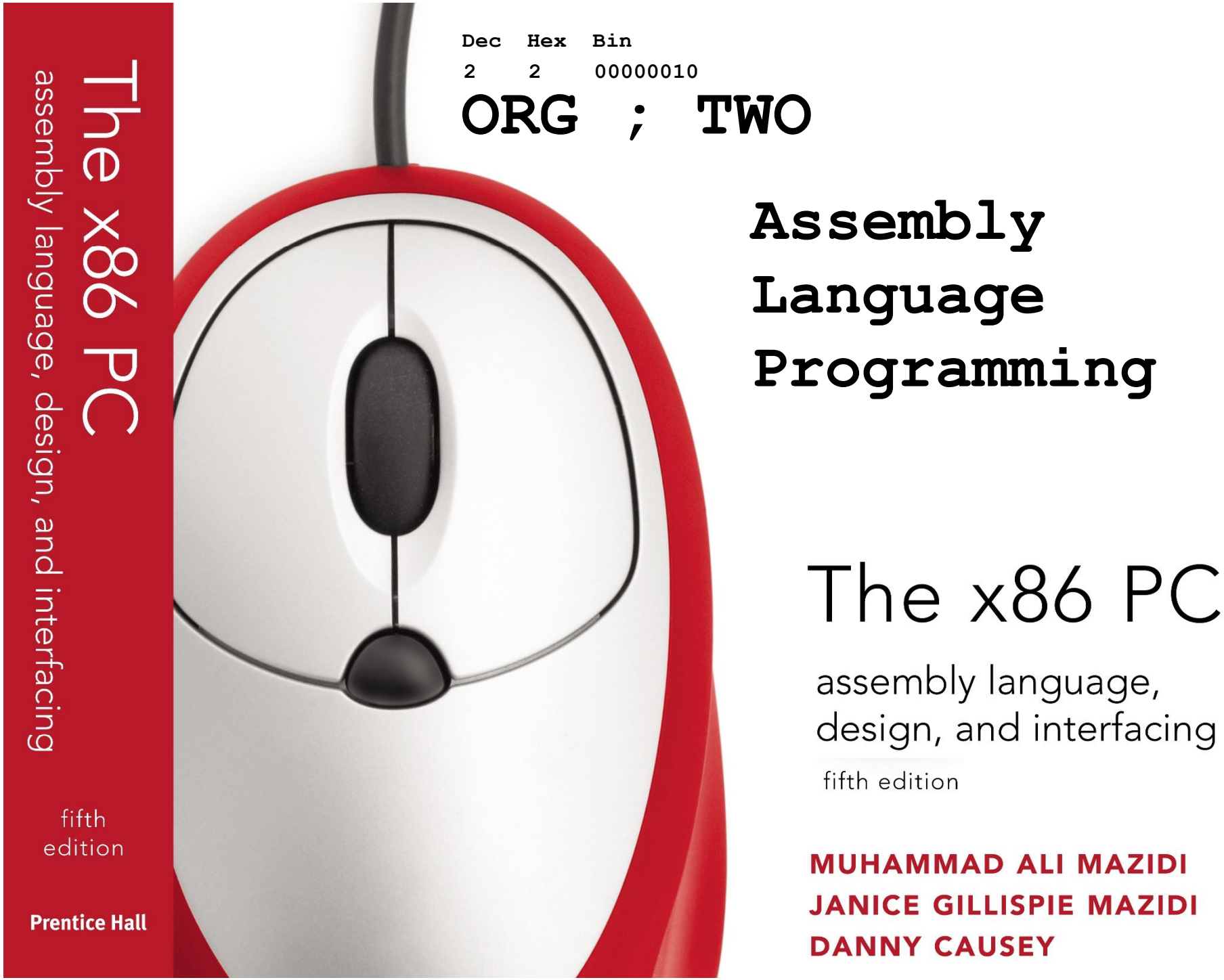


Dec	Hex	Bin
2	2	00000010

ORG ; TWO

Assembly Language Programming

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY

OBJECTIVES

this chapter enables the student 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.

OBJECTIVES (cont)

this chapter enables the student to:

- Code Assembly language data directives for binary, hex, decimal, or ASCII data.
- Write an Assembly language program using either the simplified segment definition or the full segment definition.
- Explore the use of the MASM and emu8086 assemblers.

Assembly Language

- There is a one-to-one relationship between assembly and machine language instructions
- What is found is that a compiled machine code implementation of a program written in a high-level language results in inefficient code
  - More machine language instructions than an assembled version of an equivalent handwritten assembly language program
- Two key benefits of assembly language programming
  - It takes up less memory
  - It executes much faster

2.0: ASSEMBLY LANGUAGE

- An Assembly language program is a series of statements, or lines.
  - Either Assembly language instructions, or statements called *directives*.
    - Directives (pseudo-instructions) give directions to the assembler about how it should translate the Assembly language instructions into machine code.
- Assembly language instructions consist of four fields:

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

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

2.1: DIRECTIVES AND A SAMPLE PROGRAM
assembly language instructions

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

- The label field allows the program to refer to a line of code by name.
  - The label field cannot exceed 31 characters.
    - A label must end with a colon when it refers to an opcode generating instruction.

2.1: DIRECTIVES AND A SAMPLE PROGRAM
assembly language instructions

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

- The mnemonic (instruction) and operand(s) fields together accomplish the tasks for which the program was written.

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

  - The mnemonic opcodes are **ADD** and **MOV**.
  - "**AL,BL**" and "**AX,6764**" are the operands.
    - Instead of a mnemonic and operand, these fields could contain assembler pseudo-instructions, or *directives*.
    - Directives do not generate machine code and are used only by the assembler as opposed to instructions.

2.1: DIRECTIVES AND A SAMPLE PROGRAM

assembly language instructions

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

- Examples of directives are **DB**, **END**, and **ENDP**.

```
;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 OS
            INT    21H          ;
MAIN        ENDP
            END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
assembly language instructions

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

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

2.1: DIRECTIVES AND A SAMPLE PROGRAM
model definition

- After the first two comments is the MODEL directive.
  - This directive selects the size of the memory model.

```
;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 OS
            INT   21H          ;
MAIN        ENDP
            END   MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
model definition

- Among the options for the memory model are SMALL, MEDIUM, COMPACT, and LARGE.

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

2.1: DIRECTIVES AND A SAMPLE PROGRAM segment definition

- Every line of an Assembly language program must correspond to one an x86 CPU segment register.
  - CS (code segment); DS (data segment).
  - SS (stack segment); ES (extra segment).
- The simplified segment definition format uses three simple directives: "**.CODE**" "**.DATA**" "**.STACK**"
  - Which correspond to the CS, DS, and SS registers.

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

  - The stack segment defines storage for the stack.
  - The data segment defines the data the program will use.
  - The code segment contains Assembly language instructions.

2.1: DIRECTIVES AND A SAMPLE PROGRAM
stack segment

- This directive reserves 64 bytes of memory for the stack:

```
;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 OS
            INT   21H          ;
MAIN        ENDP
            END   MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM data segment

- The data segment defines three data items:
  - DATA1, DATA2, and SUM.

```
;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 OS
            INT    21H          ;
MAIN        ENDP
            END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
data segment

- The DB directive is used by the assembler to allocate memory in byte-sized chunks.
  - Each is defined as DB (define byte).
    - Memory can be allocated in different sizes.
  - 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.

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The first line of the segment after the .CODE directive is the PROC directive.

```
;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 OS
            INT    21H          ;
MAIN        ENDP
            END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- *A procedure* is a group of instructions designed to accomplish a specific function.
  - A code segment is organized into several small procedures 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 OS requires the entry point to the user program to be a FAR procedure.

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- Before the OS passes control to the program so it may execute, it assigns segment registers values.
  - When the program begins executing, only CS and SS have the proper values.
    - DS (and ES) values are initialized by the program.

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

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The program loads **AL** & **BL** with **DATA1** & **DATA2**, ADDs them together, and stores the result in **SUM**.

```
;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 OS
            INT    21H          ;
MAIN        ENDP
            END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The last instructions, "**MOV AH,4CH**" & "**INT 21H**" return control to the operating system.

```
;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 OS
            INT    21H          ;
MAIN        ENDP
            END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The last two lines end the procedure & program.
  - The label for **ENDP(MAIN)** matches the label for **PROC**.

```
;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 OS
            INT    21H          ;
MAIN        ENDP
            END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- It is handy to keep a sample shell & 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 OS
MAIN        ENDP
            END   MAIN         ;this is the program exit point
```

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

- MASM & LINK are the assembler & linker programs.
  - Many editors or word processors can be used to create and/or edit the program, and produce an ASCII file.
  - The steps to create an executable Assembly language program are 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

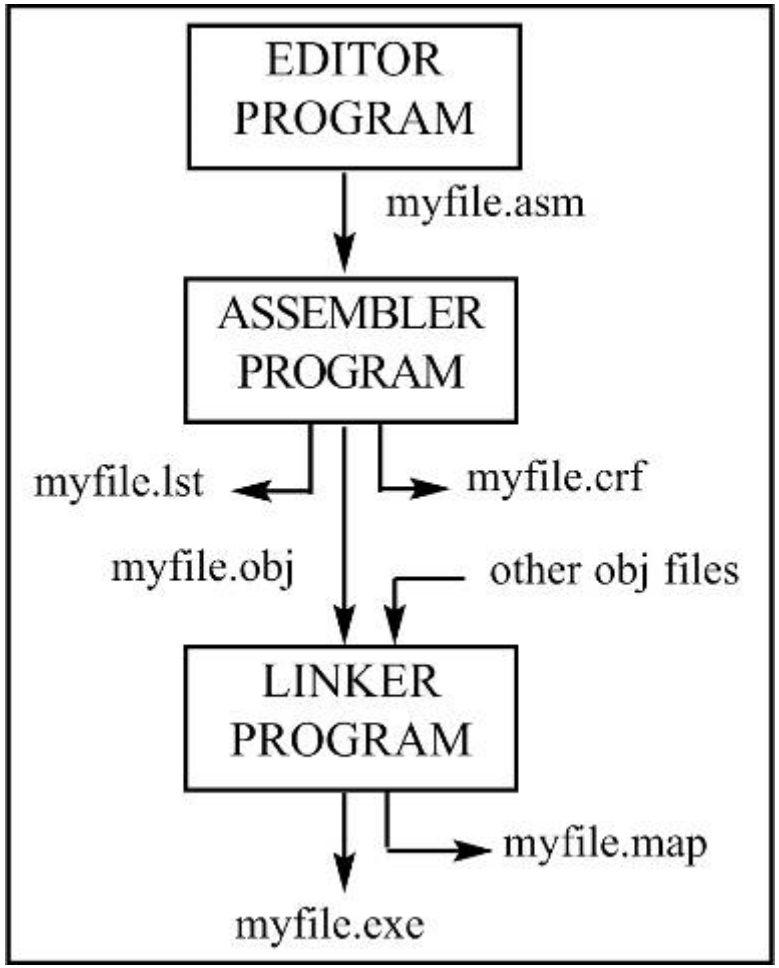
2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

- The source file must end in ".asm“.
  - The ".asm" file is assembled by an assembler, like MASM.
    - The assembler will produce an object file and a list file, along with other files useful to the programmer.
- The extension for the object file must be ".obj".
  - This object file is input to the LINK program, to produce the executable program that ends in ".exe".
  - The ".exe" file can be run (executed) by the microprocessor.

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

Before feeding the ".obj" file into LINK, all syntax errors must be corrected.

Fixing these errors will not guarantee the program will work as intended, as the program may contain conceptual errors.



2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

- Figure 2-4 shows how an executable program is created & run by following the steps outlined above.

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

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

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

       47962 + 413345 Bytes symbol space free

     0 Warning Errors
     0 Severe  Errors


C>LINK C:MYFILE.OBJ <enter>

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

See the entire program listing on page 61 of your textbook.

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM
PAGE and TITLE directives

- The PAGE directive, which tells the printer how the list should be printed, is formatted as:

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

- In default mode, the output will be 66 lines per page & 80 characters per line.
  - In this book, programs change the settings to 60 &132.

```
PAGE 60,132
```

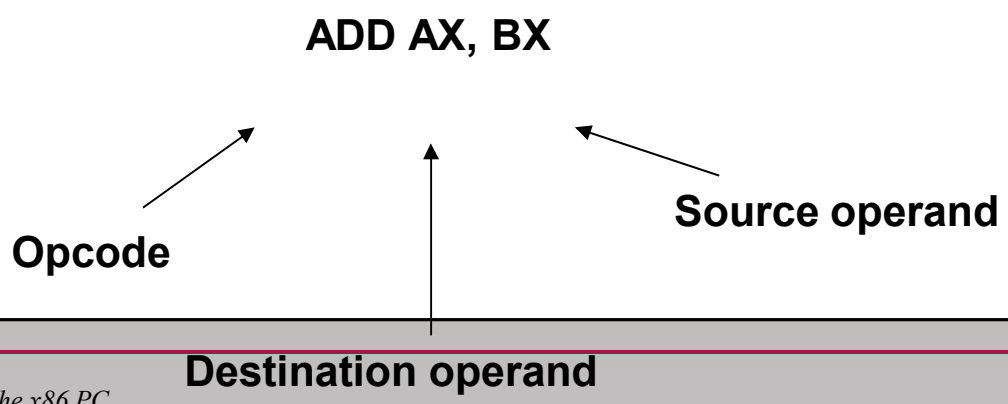
  - The range for number of lines is 10 to 255 and for columns is 60 to 132.

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM
PAGE and TITLE directives

- When the list is printed, the assembler can print the title of the program on top of each page.
  - It is common to put the name of the program immediately after the TITLE pseudo-instruction.
    - And a brief description of the function of the program.
  - The text after the TITLE pseudo-instruction cannot be exceed 60 ASCII characters.

Software

- The sequence of commands used to tell a microcomputer what to do is called a **program**
- Each command in a program is called an **instruction**
- 8088 understands and performs operations for **117 basic instructions**
- The native language of the **IBM PC** is the machine language of the 8088
- A program written in machine code is referred to as **machine code**
- In 8088 assembly language, each of the operations is described by alphanumeric symbols instead of just 0s or 1s.



DEBUG program instruction set (page 825 mzd)

- Debug instructions
- List of commands
  - a Assemble [address] you can type in code this way
  - c range address ; compare c 100 105 200
  - d [range] ; Dump d 150 15A
  - e address [list] ; Enter e 100
  - f Fill range list F 100 500 ‘ ‘
  - g Go [=address] addresses runs the program
  - h Value1 Value2 ; addition and subtraction H 1A 10
  - i Input port I 3F8
  - r Show & change registers Appears to show the same thing as t, but doesn't cause any code to be executed.
  - t=startaddress Trace either from the starting address or current location.
  - u startaddress UnAssemble

Some examples with debug

```
0100 mov ax,24b6
0103 mov di, 85c2
0106 mov dx,5f93
0109 mov sp,1236
010c push ax
010d push di
010e int 3
```

Display the stack contents after execution.

-D 1230 123F

Some examples with DEBUG

- 0100 mov al,9c
- 0102 mov dh,64
- 0104 add al,dh
- 0109 int 3

trace these three commands and observe the flags

T=<start trace location>

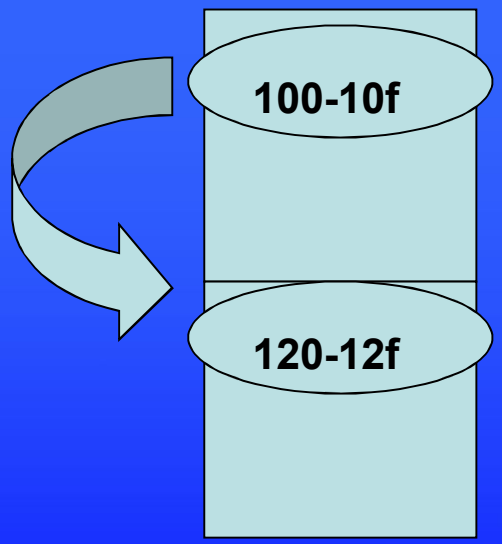
Saving and Loading a file

- After the code has been entered with the A command
- Use CX to store data indicating number of bytes to save. BX is the high word.
- Use N filename.com
- Then W command to write to file.
- L loads this file.

Example

Copy the contents of a block of memory (16 bytes) starting at location 20100h to another block of memory starting at 20120h

```
        MOV AX,2000
        MOV DS,AX
        MOV SI, 100
        MOV DI, 120
        MOV CX, 10
NXTPT:  MOV AH, [SI]
        MOV [DI], AH
        INC SI
        INC DI
        DEC CX
        JNZ  NXTPT
```



2.2: ASSEMBLE, LINK, AND RUN A PROGRAM
PAGE and TITLE directives .crf

- MASM produces another optional file, the *cross-reference*, which has the extension ".crf".
  - An alphabetical list of all symbols & labels in the program.
    - Also program line numbers in which they are referenced.

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM
LINKing the program

- The assembler (MASM) creates the opcodes, operands & offset addresses under the ".obj" file.
- The LINK program produces the ready-to-run program with the ".exe" (EXEcutable) extension.
  - The LINK program sets up the file so it can be loaded by the OS and executed.
- The program can be run at the OS level, using the following command: **`C>myfile`**
  - When the program name is typed in at the OS level, the OS loads the program in memory.
    - Referred to as *mapping*, which means that the program is mapped into the physical memory of the PC.

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM
LINKing the program .map

- 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 optional .map file, which gives the name of each segment, where it starts, where it stops, and its size in bytes.

Download Microsoft Assembler (MASM) and a Tutorial on how to use it from:

http://www.MicroDigitalEd.com

2.3: MORE SAMPLE PROGRAMS

- Program 2-1, and the list file generated when the program was assembled.

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 data item to AL
```

See the entire program listing on page 63 of your textbook.

2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-1

- 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.
  - (3) g, to go. (run the program)

2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-1

- Program 2-1, explained instruction by instruction:
  - "**`MOV CX,05`**" will load the value 05 into the CX register.
    - 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_IN.
    - The assembler starts at offset 0000? and uses memory for the data, 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`**" increments the pointer by adding 1 to BX.
    - This will cause BX to point to the next data item. (next byte)

2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-1

- Program 2-1, explained instruction by instruction:
  - "**`DEC CX`**" will decrement (subtract 1 from) the CX counter and 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 only after the zero flag has been set high by the "DEC CX" instruction (CX becomes zero).
  - When CX becomes zero, this means that the loop is completed and all five numbers have been added to AL.

2.3: MORE SAMPLE PROGRAMS
various approaches to Program 2-1

- Variations of Program 2-1 clarify use of addressing modes, and show that the x86 can use any general-purpose register for arithmetic and logic operations.

```
;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
```

2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-2

- The 16-bit data (a word) is stored with the low-order byte first, referred to as "little endian."

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
```

See the entire program listing on page 66 of your textbook.

2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-2

- 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.
- "**`MOV SI,OFFSET SUM`**" was used to load the pointer for the memory allocated for the label SUM.
- "**`MOV [SI],BX`**" moves the contents of register BX to memory locations with offsets 0010 and 0011.
- Program 2-2 uses the ORG directive to set the offset addresses for data items.
  - This caused SUM to be stored at DS:0010.

2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-3

- Program 2-3 shows the data segment being dumped before and after the program was run.

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 AL to COPY area
```

See the entire program listing on page 67 of your textbook.

2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-3

- C4 was coded in the data segments as 0C4.
  - Indicating that C is a hex number and not a letter.
    - Required if the first digit is a hex digit A through F.
- This program uses registers SI & DI as pointers to the data items being manipulated.
  - The first is a pointer to the data item to be copied.
  - The second points to the location the data is copied to.
- With each iteration of the loop, both data pointers are incremented to point to the next byte.

2.4: CONTROL TRANSFER INSTRUCTIONS FAR and NEAR

- In the sequence of instructions, it is often necessary to transfer program control to a different location.
  - If control is transferred to a memory location within the current code segment, it is NEAR.
    - Sometimes called intrasegment. (within segment)
  - If control is transferred outside the current code segment, it is a FAR jump.
    - Or intersegment. (between segments)

2.4: CONTROL TRANSFER INSTRUCTIONS FAR and NEAR

- As the CS:IP registers always point to the address of the next instruction to be executed, they must be updated when a control transfer 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.

2.4: CONTROL TRANSFER INSTRUCTIONS conditional jumps

- Conditional jumps 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 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.

2.4: CONTROL TRANSFER INSTRUCTIONS conditional jumps

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.

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- All conditional jumps are short jumps.
  - The address of the target must be within -128 to +127 bytes of the IP.
- The conditional jump is a two-byte instruction.
  - One byte is the opcode of the J condition.
  - The second byte is a value between 00 and FF.
    - An offset range of 00 to FF gives 256 possible addresses.
- In a jump backward, the second byte is the 2's complement of the displacement value

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- To calculate the target address, the second byte is added to the IP of the instruction after the jump.

```
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
```

  – "**JNZ AGAIN**" was assembled as "**JNZ 000D**", and 000D is the address of the instruction with the label AGAIN.
    - "**JNZ 000D**" has the opcode 75 and the target address FA.

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- This is followed by "**MOV SUM,AL**", which is located beginning at offset address 0013.

```
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 IP value of **MOV,0013**, is added to FA to calculate the address of label AGAIN, and the carry is dropped.
    - FA is the 2's complement of -6.

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- Calculate a forward jump target address by adding the IP of the following instruction to the operand.
  - The displacement value is positive, as shown.

```
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,0DFH
0012        88 04    NEXT:    MOV   [SI],AL
```

  - "**JB NEXT**" has the opcode **72**, the target address **06** and is located at **IP = 000A** and **000B**.
    - The jump is 6 bytes from the next instruction, is IP = 000C.
    - Adding gives us 000CH + 0006H = 0012H, which is the exact address of the NEXT label.

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- 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.
  - Any attempt is made to violate this rule will generate a "**`relative jump out of range`**" message.

2.4: CONTROL TRANSFER INSTRUCTIONS unconditional jumps

- An unconditional jump transfers control to the target location label unconditionally, in the following forms:
  - **SHORT JUMP** - in the format "**`JMP SHORT label`**".
    - A jump within -128 to +127 bytes of memory relative to the address of the current IP, opcode EB.
  - **NEAR JUMP** - the default, has the format "**`JMP label`**".
    - A jump within the current code segment, opcode E9.
    - The target address can be any of the addressing modes of direct, register, register indirect, or memory indirect:
  - **Direct JUMP** - exactly like the short jump.
    - Except that the target address can be anywhere in the segment in the range +32767 to -32768 of the current IP.

2.4: CONTROL TRANSFER INSTRUCTIONS unconditional jumps

- An unconditional jump transfers control to the target location label unconditionally, in the following forms:
  - **Register indirect JUMP** - target address is in a register.
    - In "**`JMP BX`**", IP takes the value BX.
  - **Memory indirect JMP** - target address is the contents of two memory locations, pointed at by the register.
    - "**`JMP [DI]`**" will replace the IP with the contents of memory locations pointed at by DI and DI+1.
  - **FAR JUMP** - in the format "**`JMP FAR PTR label`**". register.
    - A jump out of the current code segment
    - IP and CS are both replaced with new values.

2.4: CONTROL TRANSFER INSTRUCTIONS
CALL statements

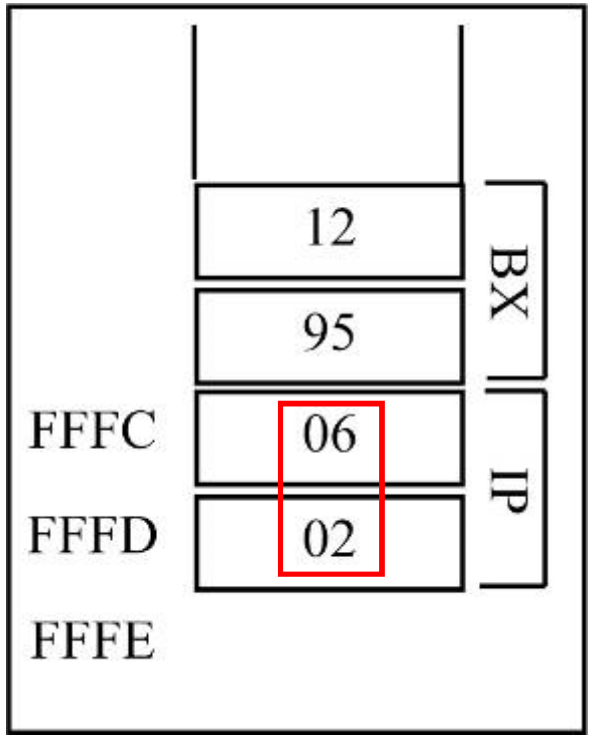
- The CALL instruction is used to call a procedure, to perform tasks that need to be performed frequently.
  - The target address could be in the current segment, in which case it will be a NEAR call or outside the current CS segment, which is a FAR call.
- The microprocessor saves the address of the instruction following the call on the stack.
  - To know where to return, after executing the subroutine.
    - In the NEAR call only the IP is saved on the stack.
    - In a FAR call both CS and IP are saved.

2.4: CONTROL TRANSFER INSTRUCTIONS
CALL statements

- For control to be transferred back to the caller, the last subroutine instruction must be RET (return).
  - For NEAR calls, the IP is restored.
  - For FAR calls, CS & IP are restored.
- Assume SP = FFFEH:

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

  - Since this is a NEAR call, only IP is saved on the stack.
    - The IP address **0206**, which belongs to the "**MOV AX,142F**" instruction, is saved on the stack.



2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

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

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

2.4: CONTROL TRANSFER INSTRUCTIONS
assembly language subroutines

```
            .CODE
MAIN        PROC  FAR          ;THIS IS THE ENTRY POINT FOR OS
            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
```

It is common to have one main program and many subroutines to be called from the main.

Each subroutine can be a separate module, tested separately, then brought together.

If there is no specific mention of FAR after the directive PROC, it defaults to NEAR.

2.4: CONTROL TRANSFER INSTRUCTIONS
rules for names in Assembly language

- The names used for labels in Assembly language programming consist of…
  - Alphabetic letters in both upper- and lowercase.
  - The digits 0 through 9.
  - Question mark (?); Period (.); At (@)
  - Underline (_); Dollar sign ($)
- Each label name must be unique.
  - They may be up to 31 characters long.
- The first character must be an alphabetic or special character.
  - It cannot be a digit.

2.4: CONTROL TRANSFER INSTRUCTIONS
rules for names in Assembly language

- The period can only be used as the first character.
  - This is not recommended since later versions of MASM have several reserved words that begin with a period.

2.5: DATA TYPES AND DATA DEFINITION
x86 data types

- The 8088/86 processor supports many data types.
  - Data types can be 8- or 16-bit, positive or negative.
    - The programmer must break down data larger than 16 bits (0000 to FFFFH, or 0 to 65535 in decimal).
  - A number less than 8 bits wide must be coded as an 8-bit register with the higher digits as zero.
    - A number is less than 16 bits wide must use all 16 bits.

2.5: DATA TYPES AND DATA DEFINITION
ORG origin

- ORG is used to indicate the beginning of the offset address.
  - The number 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.

2.5: DATA TYPES AND DATA DEFINITION
DB define byte

- One of the most widely used data directives, it allows allocation of memory in byte-sized chunks.
  - This is the smallest allocation unit permitted.
  - DB can define numbers in decimal, binary, hex, & ASCII.
    - **D** after the decimal number is optional.
    - **B** (binary) and **H** (hexadecimal) is required.
    - To indicate ASCII, place the string in single quotation marks.
- DB is the only directive that can be used to define ASCII strings larger than two characters.
  - It should be used for all ASCII data definitions.

2.5: DATA TYPES AND DATA DEFINITION
DB define byte

- 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
```

  – **Single** or **double** quotes can be used around ASCII strings.
    - Useful for strings, which should contain a single quote, such as "O'Leary".

2.5: DATA TYPES AND DATA DEFINITION
DB define byte

- List file for DB examples.

```
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    '2591'            ;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
```

2.5: DATA TYPES AND DATA DEFINITION
DUP duplicate

- DUP will duplicate a given number of characters.

```
      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
```

  – Two methods of filling six memory locations with FFH.

2.5: DATA TYPES AND DATA DEFINITION DUP duplicate

- List file for DUP examples.

```
0030                                    ORG     0030H
0030  FF FF FF FF FF FF         DATA7   DB      0FFH,0FFH,0FFH,0FFH,0FFH,0FFH  ; 6 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
                        ]       ]
```

2.5: DATA TYPES AND DATA DEFINITION
DW define word

- DW is used to allocate memory 2 bytes (one word) at a time:

```
            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
```

- List file for DW examples.

```
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
         ????       ]
```

2.5: DATA TYPES AND DATA DEFINITION
EQU equate

- EQU associates a constant value with a data label.
  - When the label appears in the program, its constant value will be substituted for the label.
  - Defines a constant without occupying a memory location.
- 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.
  - In contrast to using DB:

```
COUNT   DB 25
```

2.5: DATA TYPES AND DATA DEFINITION
EQU equate

- When executing the same instruction "MOV CX,COUNT" it will be in the direct addressing mode.
  - EQU can also be used in the data segment:

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

  - Assume a constant (a fixed value) used in many different places in the data and code segments.
    - By use of EQU, one can change it once and the assembler will change all of them.

2.5: DATA TYPES AND DATA DEFINITION
DD define doubleword

- The DD directive is used to allocate memory locations that are 4 bytes (two words) in size.
  - Data is converted to hex & placed in memory locations
    - Low byte to low address and high byte to high address.

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

- List file for DD examples.

```
00A0                          ORG  00A0H
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
```

2.5: DATA TYPES AND DATA DEFINITION
DQ define quadword

- DQ is used to allocate memory 8 bytes (four words) in size, to represent any variable up to 64 bits wide:

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

- List file for DQ examples.

```
00C0                          ORG  00C0H
00C0  C223450000000000   DATA20  DQ  4523C2H     ;HEX
00C8  4948000000000000   DATA21  DQ  'HI'        ;ASCII CHARACTERS
00D0  0000000000000000   DATA22  DQ  ?           ;NOTHING
```

2.5: DATA TYPES AND DATA DEFINITION
DT define ten bytes

- DT is used for memory allocation of packed BCD numbers.
  - This directive allocates 10 bytes.
    - A maximum of 18 digits can be entered.
  - The "**H**" after the data is not needed.

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

- List file for DT examples.

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

2.5: DATA TYPES AND DATA DEFINITION
DQ define ten bytes

- 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
```

2.5: DATA TYPES AND DATA DEFINITION directives

- 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.

```
-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..........
```

2.5: DATA TYPES AND DATA DEFINITION directives

- All of the data directives use the little endian format.
  - For ASCII data, only DB can define data of any length.
    - Use of DD, DQ, or DT directives for ASCII strings of more than 2 bytes gives an assembly error.

```
-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..........
```

2.5: DATA TYPES AND DATA DEFINITION directives

- Review "**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.

```
-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..........
```

2.5: DATA TYPES AND DATA DEFINITION directives

- When DB is used for ASCII numbers, it places them backwards in memory.
  - Review "**DATA4 DB '2591'**" at origin **10H:32**,
    - ASCII for 2, is in memory location 10H;35; for 5, in 11H; etc.

```
-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..........
```

2.6: FULL SEGMENT DEFINITION segment definition

- The SEGMENT and ENDS directives indicate the beginning &ending of a segment, in this format:

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

  - The label, or name, must follow naming conventions and 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 full segment definition, the "**.MODEL**" directive is not used.

2.6: FULL SEGMENT DEFINITION segment definition

- 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.
    - Followed by programs 2-2 and 2-3.

2.6: FULL SEGMENT DEFINITION segment definition

```
;FULL SEGMENT DEFINITION
      ;--- stack segment ---
      name1 SEGMENT
            DB    64 DUP (?)
      name1 ENDS
      ;--- data segment ---
      name2 SEGMENT
      ;place data definitions here
      name2 ENDS
      ;--- code segment ---
name3 SEGMENT
      MAIN  PROC  FAR
            ASSUME ...
            MOV   AX,name2
            MOV   DS,AX
            ...
      MAIN  ENDP
      name3 ENDS
            END   MAIN
```

```
;SIMPLIFIED FORMAT
.MODEL  SMALL
.STACK    64
;
;
;------------------
. DATA
;place data definitions here
;
;------------------
.CODE
MAIN  PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      ...
      ...
MAIN  ENDP
      END   MAIN
```

Figure 2-8

2.6: FULL SEGMENT DEFINITION segment definition

- Program 2-2 rewritten using full 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
```

See the entire program listing on page 78 of your textbook.

2.6: FULL SEGMENT DEFINITION segment definition

- Program 2-3 rewritten using 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
```

See the entire program listing on page 79 of your textbook.

2.6: FULL SEGMENT DEFINITION
stack segment definition

- The stack segment shown 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
```

2.6: 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
```

- The code segment also begins and ends with SEGMENT and ENDS directives:

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

2.6: FULL SEGMENT DEFINITION
code segment definition

- Immediately after PROC, the ASSUME directive, associates segments with specific registers.
  - By assuming 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.
  - ASSUME tells the assembler which of the segments, defined by SEGMENT, should be used.
    - Also helps the assembler to calculate the offset addresses from the beginning of that segment.
- In "**`MOV AL, [BX]`** " the BX register is the offset of the data segment.

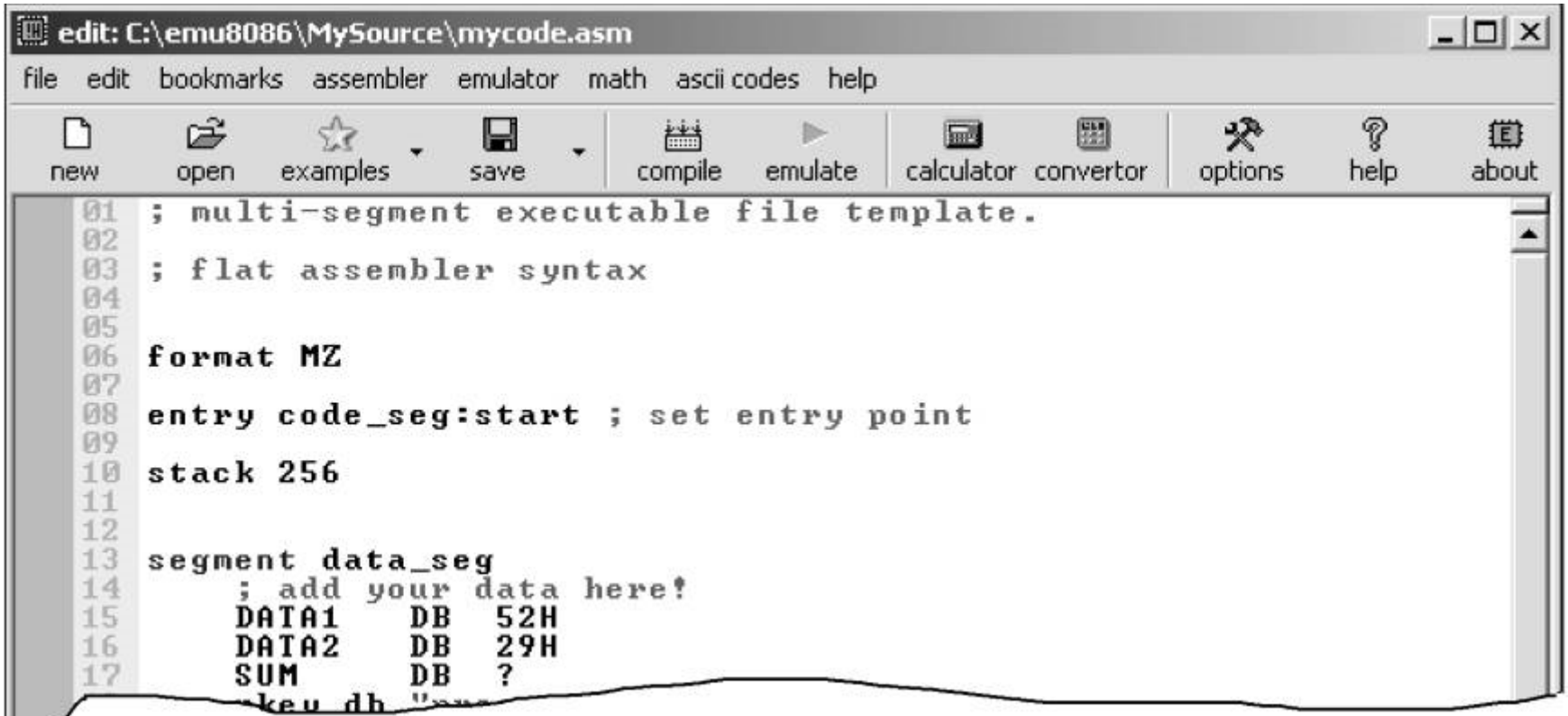
2.6: FULL SEGMENT DEFINITION
code segment definition

- On transfer of control from OS to the program, of the three segment registers, only CS and SS have the proper values.
  - The DS value (and ES) must be initialized by the program.

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

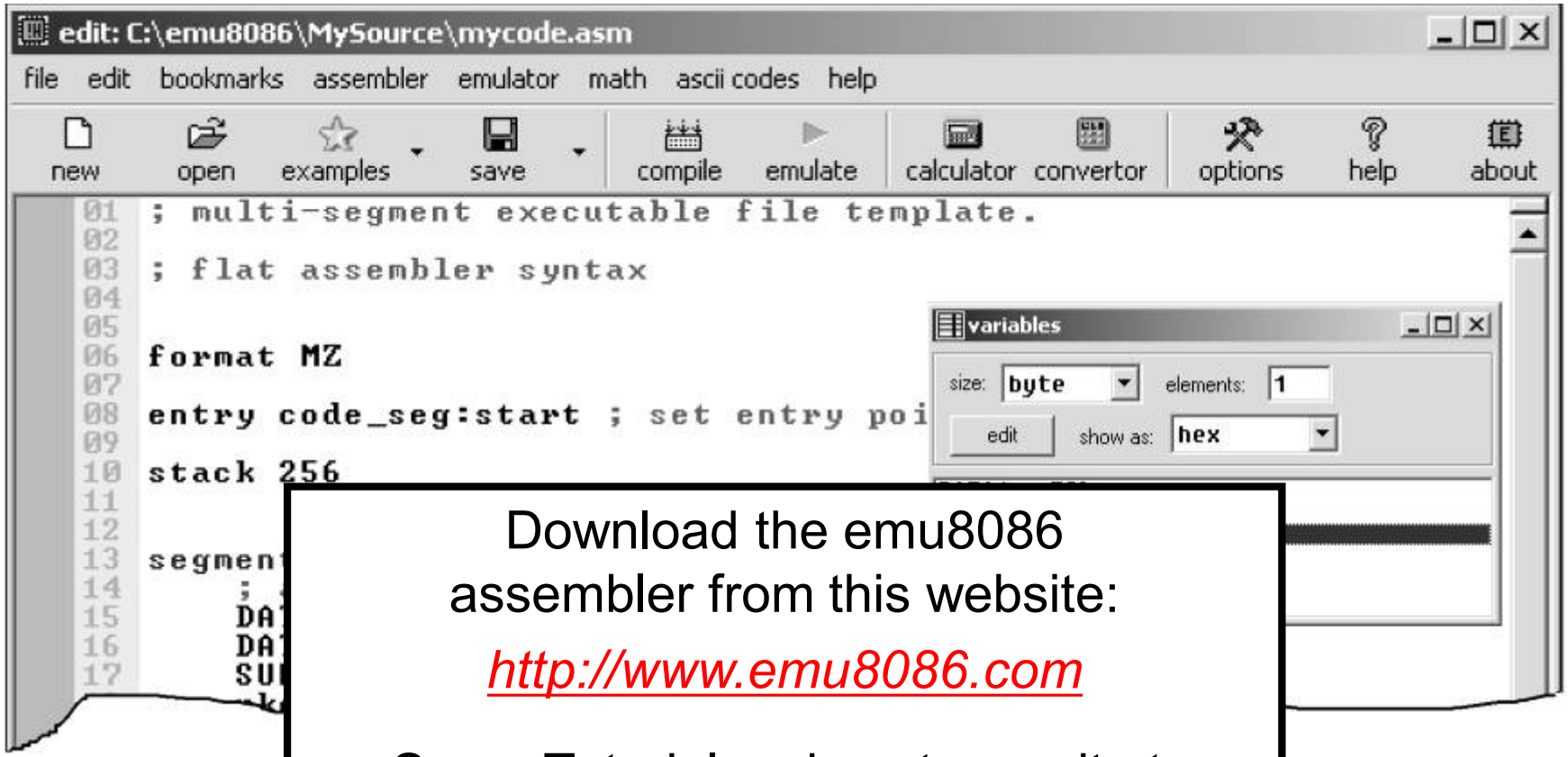
2.6: FULL SEGMENT DEFINITION the emu8086 assembler

- A simple, popular assembler for 8086 Assembly language programs is called emu8086.



See emu8086 screenshots on page 80 - 82 of your textbook.

2.6: FULL SEGMENT DEFINITION
the emu8086 assembler



Download the emu8086 assembler from this website:

http://www.emu8086.com

See a Tutorial on how to use it at:

http://www.MicroDigitalEd.com

2.6: FULL SEGMENT DEFINITION
EXE vs. COM files

- The EXE file is used widely as it can be of any size.
  - There are occasions when, due to a limited amount of memory, one needs to have very compact code.
- COM files must fit in a single segment.
  - The x86 segment size is 64K bytes, thus the COM file cannot be larger than 64K.
- To limit the size to 64K requires defining the data inside the code segment and using the end area of the code segment for the stack.
  - In contrast to the EXE file, the COM file has no separate data segment definition.

2.6: FULL SEGMENT DEFINITION
EXE vs. COM files

- The header block, which occupies 512 bytes of memory, precedes every EXE file.
  - It contains information such as size, address location in memory, and stack address of the EXE module.
  - The COM file does not have a header block.

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)

2.7: FLOWCHARTS AND PSEUDOCODE
structured programming

- Structured programming uses three basic types of program control structures:
  - Sequence.
  - Control.
  - Iteration.

2.7: FLOWCHARTS AND PSEUDOCODE
structured programming

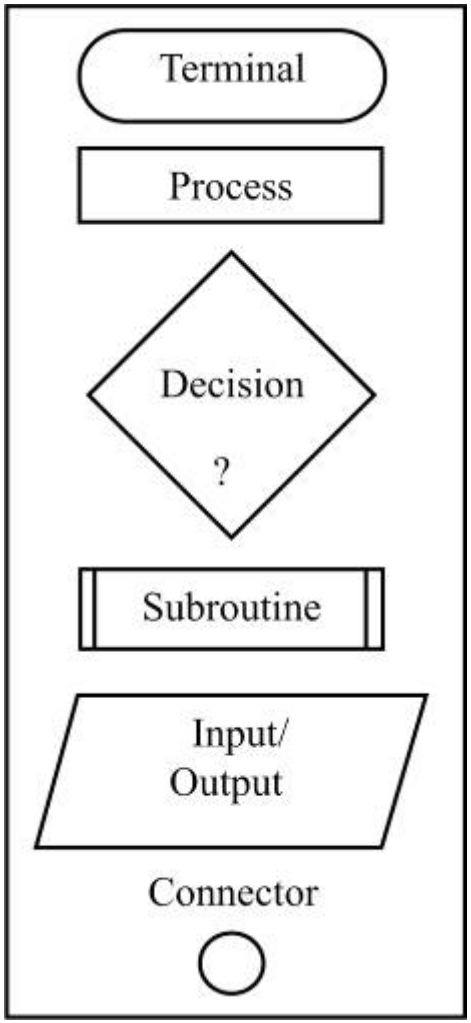
- Principles a structured program should follow:
  - The program should be designed *before* it is coded.
    - By using flowcharting or pseudocode, the design is clear those coding, as well as those maintaining the program later.
  - Use comments within the program and documentation.
    - This will help other figure out *what* the program does and *how* it does it.
  - The main routine should consist primarily of calls to subroutines that perform the work of the program.
    - Sometimes called top-down programming.
    - Using subroutines to accomplish repetitive tasks saves time in coding, and makes the program easier to read.

2.7: FLOWCHARTS AND PSEUDOCODE

- Principles a structured program should follow:
  - Data control is *very* important.
    - The programmer should document the purpose of each variable, and which subroutines might alter its value.
    - Each subroutine should document its input/output variables, and which input variables might be altered within it.

2.7: FLOWCHARTS AND PSEUDOCODE
flowcharts

- Flowcharts use graphic symbols to represent different types of program operations.
  - The symbols are connected together to show the flow of execution of the program.
    - Flowcharting has been standard industry practice for decades.
  - Flowchart templates help you draw the symbols quickly and neatly.

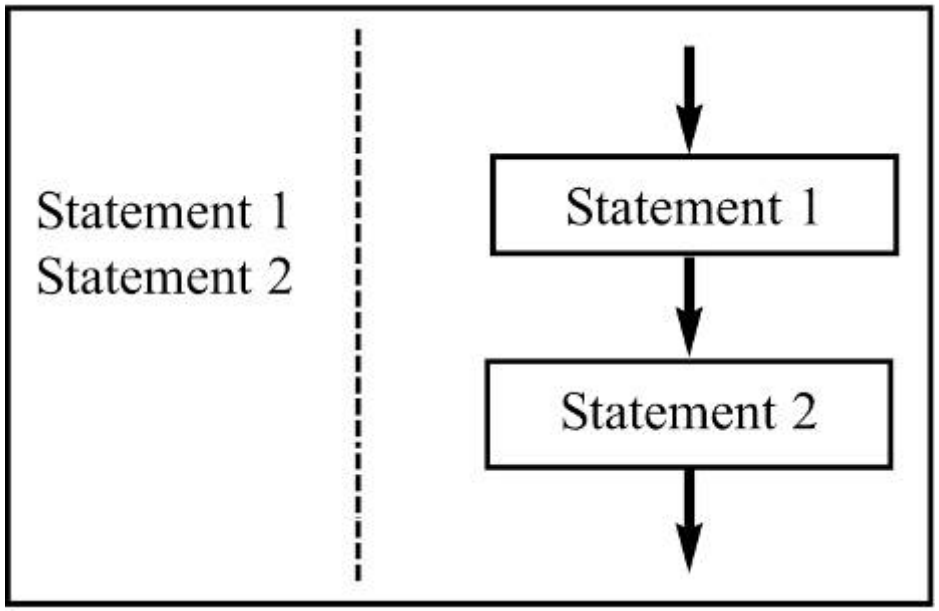


2.7: FLOWCHARTS AND PSEUDOCODE

pseudocode

- An alternative to flowcharts, *pseudocode*, involves writing brief descriptions of the flow of the code.
  - **SEQUENCE** is executing instructions one after the other.

Figure 2-15
SEQUENCE
Pseudocode vs. Flowchart



2.7: FLOWCHARTS AND PSEUDOCODE

pseudocode

- An alternative to flowcharts, *pseudocode*, involves writing brief descriptions of the flow of the code.
  - **IF-THEN-ELSE** and IF-THEN are control programming structures, which can indicate one statement or a group of statements.

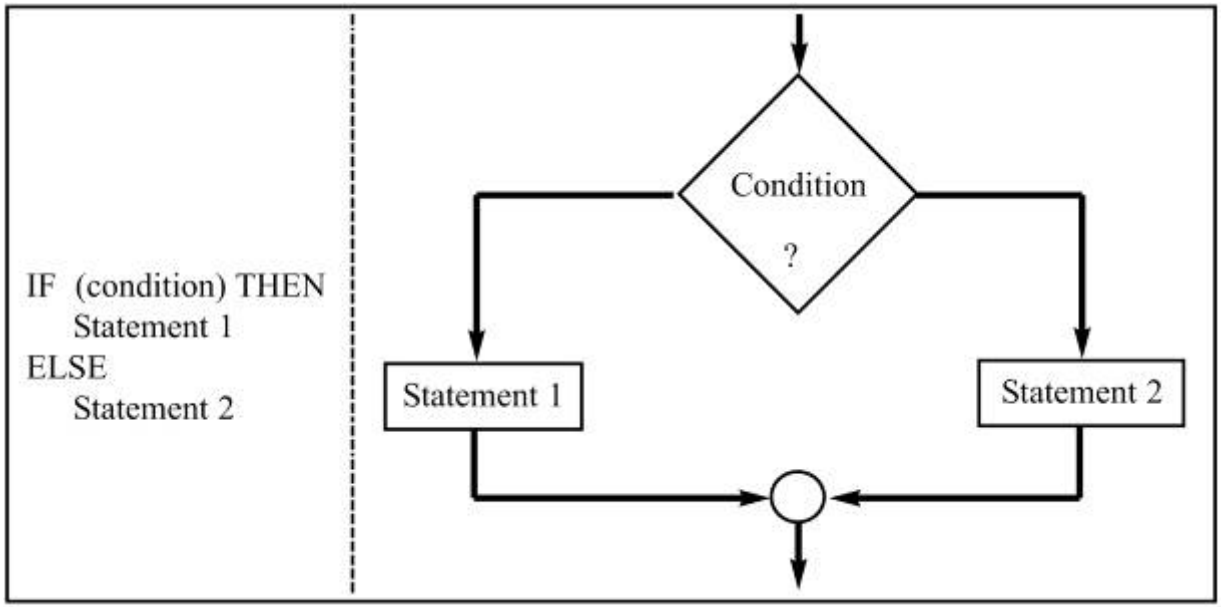


Figure 2-16
IF-THEN-ELSE
Pseudocode vs. Flowchart

2.7: FLOWCHARTS AND PSEUDOCODE

pseudocode

- An alternative to flowcharts, *pseudocode*, involves writing brief descriptions of the flow of the code.
  - IF-THEN-ELSE and **IF-THEN** are control programming structures, which can indicate one statement or a group of statements.

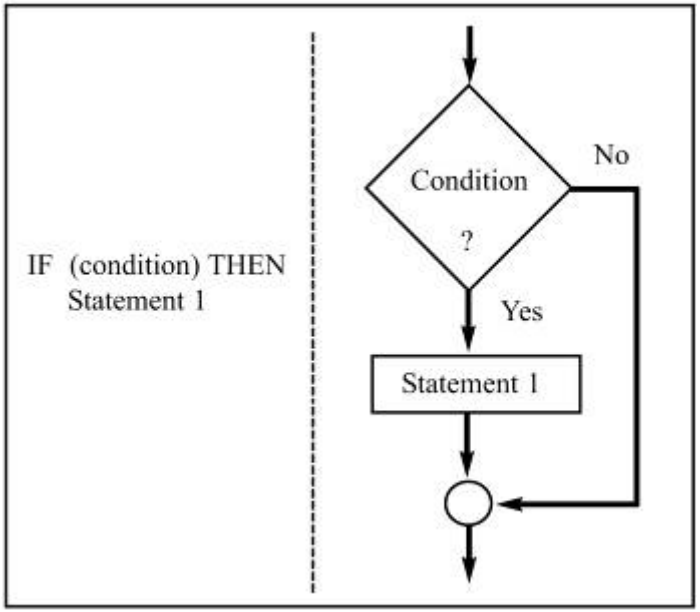


Figure 2-17
IF-THEN
Pseudocode vs. Flowchart

2.7: FLOWCHARTS AND PSEUDOCODE

pseudocode

- An alternative to flowcharts, *pseudocode*, involves writing brief descriptions of the flow of the code.
  - **REPEAT-UNTIL** and WHILE-DO are iteration control structures, which execute a statement or group of statements repeatedly.

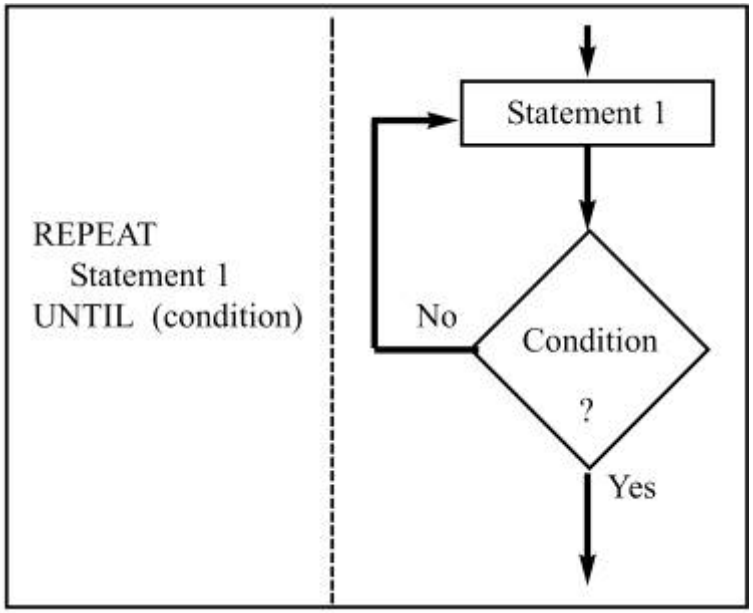


Figure 2-18
REPEAT-UNTIL
Pseudocode vs. Flowchart

REPEAT-UNTIL structure always executes the statement(s) at least once, and checks the condition after each iteration.

2.7: FLOWCHARTS AND PSEUDOCODE

pseudocode

- An alternative to flowcharts, *pseudocode*, involves writing brief descriptions of the flow of the code.
  - REPEAT-UNTIL and **WHILE-DO** are iteration control structures, which execute a statement or group of statements repeatedly.

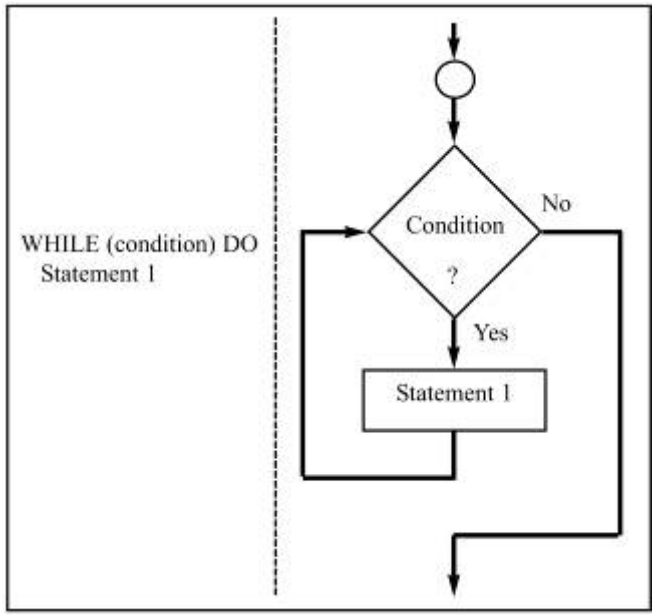


Figure 2-19
WHILE-DO
Pseudocode vs. Flowchart

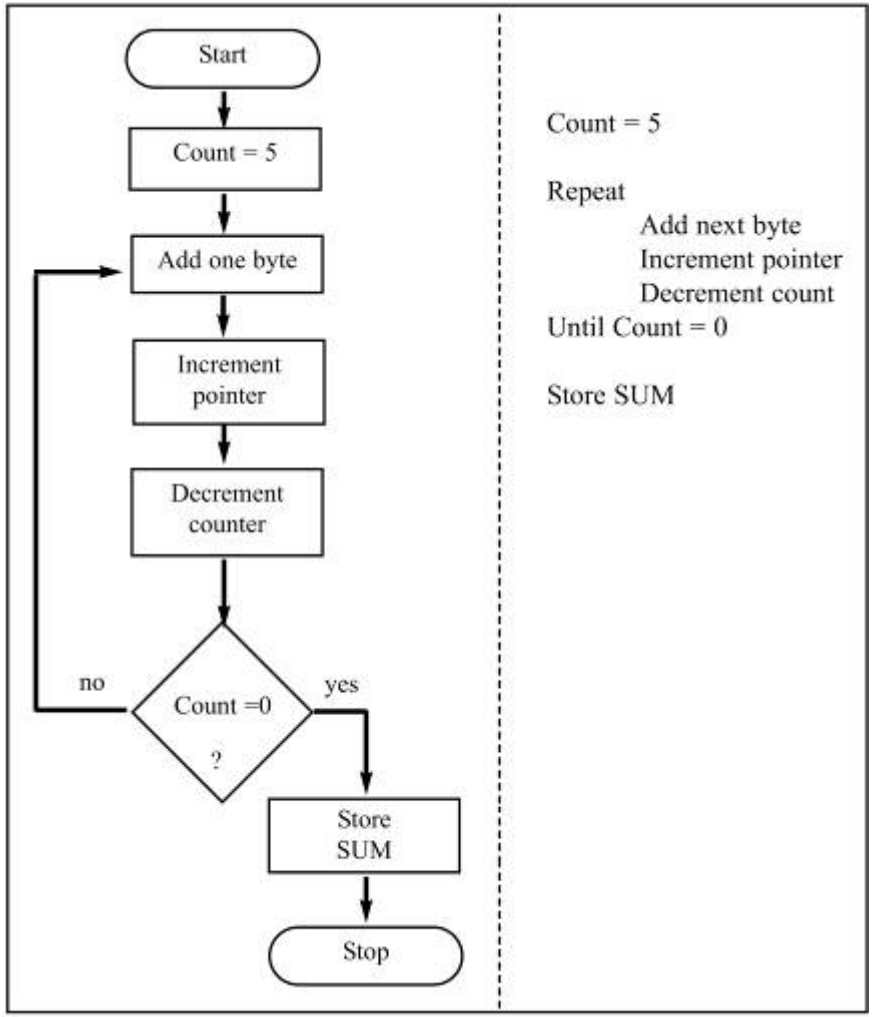
WHILE-DO may not execute the statement(s) at all, as the condition is checked at the beginning of each iteration.

2.7: FLOWCHARTS AND PSEUDOCODE control structures

```
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]                          data item to AL
```

Flowchart vs. pseudocode for Program 2-1, showing steps for initializing/decrementing counters.

Housekeeping, such as initializing the data segment register in the MAIN procedure are not included in the flowchart or pseudocode.



2.7: FLOWCHARTS AND PSEUDOCODE
control structures

- The purpose of flowcharts or pseudocode is to show the program flow, and what the program does.
  - Pseudocode gives the same information as a flowchart, in a more compact form.
    - Often written in layers, in a top-down manner.
  - Code specific to a certain language or operating platform is not described in the pseudocode or flowchart.
    - Ideally, one could take a flowchart or pseudocode and code the program in any language.

```
Dec  Hex  Bin
2    2    00000010
ENDS ; TWO
```

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY

The x86 PC
assembly language, design, and interfacing
fifth edition

Prentice Hall