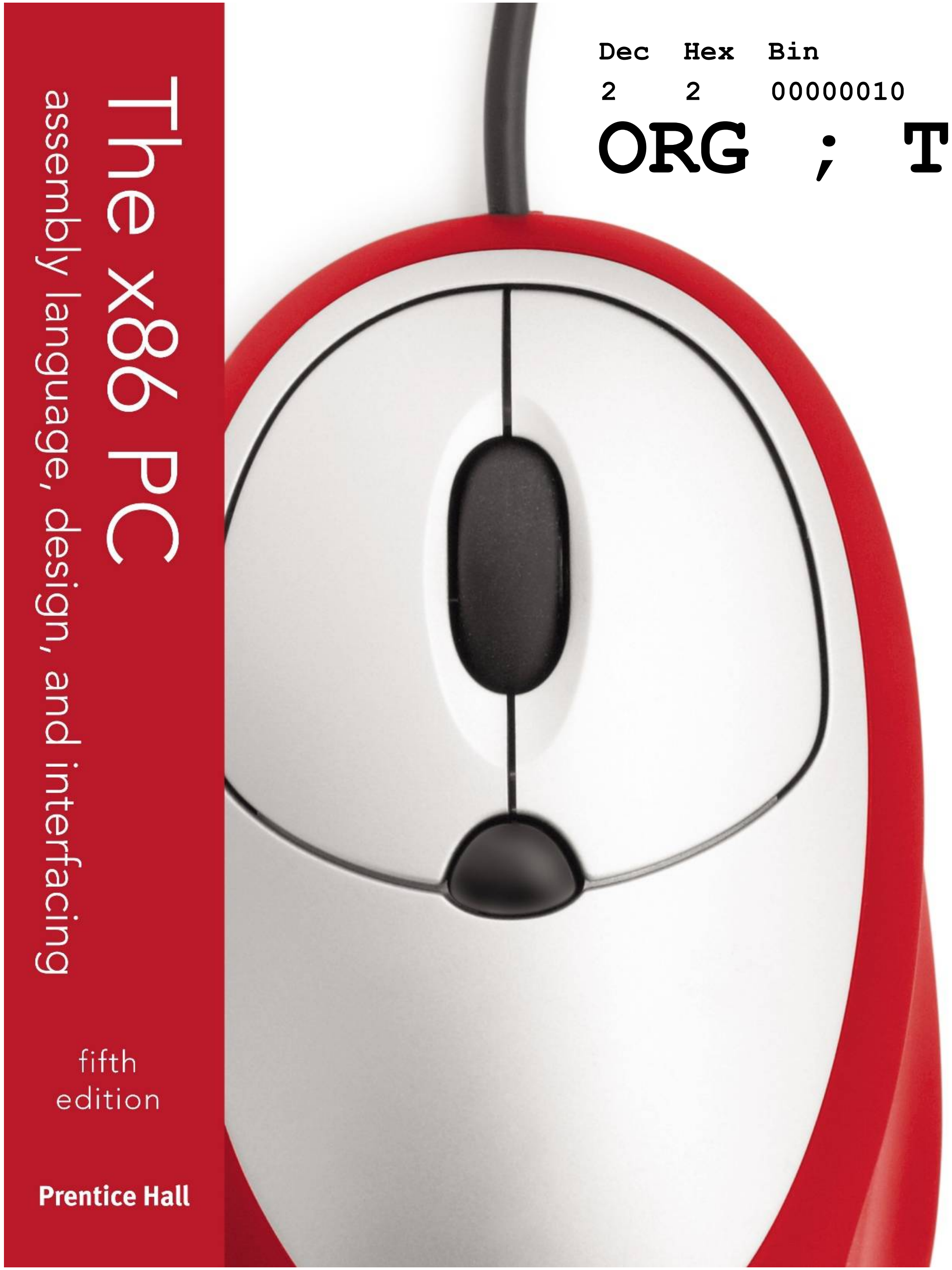
Dec Hex Bin
2 2 00000010

ORG ; THREE

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:

- Flag concepts
- Instruction Types in 8086
- Assembly language program basics.
- Flow charts summary
- Code simple Assembly language instructions.
- Assemble, link, and run a simple Assembly language program.
- Procedures
- Code control transfer instructions such as conditional and unconditional jumps and call instructions.

FLAG REGISTER

- Many Assembly language instructions alter flag register bits & some instructions function differently based on the information in the flag register.
- The flag register is a 16-bit register sometimes referred to as the *status register.*
  - Although 16 bits wide, only some of the bits are used.
    - The rest are either undefined or reserved by Intel.

FLAG REGISTER

- Six flags, called *conditional flags*, indicate some condition resulting after an instruction executes.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	R	R	R	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

R = reserved
U = undefined
OF = overflow flag
DF = direction flag
IF = interrupt flag
TF = trap flag
SF = sign flag
ZF = zero flag
AF = auxiliary carry flag
PF = parity flag
CF = carry flag

  - **These six are CF, PF, AF, ZF, SF, and OF.**
  - **The remaining three, often called *control flags*, control the operation of instructions *before* they are executed.**

bits of the flag register

- Flag register bits used in x86 Assembly language programming, with a brief explanation each:
  - **CF (Carry Flag) -** Set when there is a carry out, from d7 after an 8-bit operation, or d15 after a 16-bit operation.
    - Used to detect errors in unsigned arithmetic operations.
  - **PF (Parity Flag) -** After certain operations, the parity of the result's low-order byte is checked.
    - If the byte has an even number of 1s, the parity flag is set to 1; otherwise, it is cleared.
  - **AF (Auxiliary Carry Flag) -** If there is a carry from d3 to d4 of an operation, this bit is set; otherwise, it is cleared.
    - Used by instructions that perform BCD (binary coded decimal) arithmetic.

bits of the flag register

- Flag register bits used in x86 Assembly language programming, with a brief explanation each:
  - **ZF (Zero Flag) - Set to 1 if the result of an arithmetic or logical operation is zero; otherwise, it is cleared.**
  - **SF (Sign Flag) - Binary representation of signed numbers uses the most significant bit as the sign bit.**
    - **After arithmetic or logic operations, the status of this sign bit is copied into the SF, indicating the sign of the result.**
  - **TF (Trap Flag) - When this flag is set it allows the program to single-step, meaning to execute one instruction at a time.**
    - **Single-stepping is used for debugging purposes.**

bits of the flag register

- Flag register bits used in x86 Assembly language programming, with a brief explanation each:
  - **IF (Interrupt Enable Flag) - This bit is set or cleared to enable/disable only external maskable interrupt requests.**
  - **DF (Direction Flag) - Used to control the direction of string operations.**
  - **OF (Overflow Flag) - Set when the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit.**
    - **Used only to detect errors in signed arithmetic operations.**

flag register and ADD instruction

- Flag bits affected by the ADD instruction:
  - CF (carry flag); PF (parity flag); AF (auxiliary carry flag).
  - ZF (zero flag); SF (sign flag); OF (overflow flag).

Example 1-10

Show how the flag register is affected by the addition of 38H and 2FH.

Solution:

```
        MOV   BH,38H          ;BH= 38H
        ADD   BH,2FH          ;add 2F to BH, now BH=67H

    38        0011  1000
+   2F        0010  1111
    67        0110  0111
```

CF = 0 since there is no carry beyond d7

AF = 1 since there is a carry from d3 to d4

PF = 0 since there is an odd number of 1s in the result

ZF = 0 since the result is not zero

SF = 0 since d7 of the result is zero

flag register and ADD instruction

- Flag bits affected by the ADD instruction:
  - CF (carry flag); PF (parity flag); AF (auxiliary carry flag).
  - ZF (zero flag); SF (sign flag); OF (overflow flag).

Example 1-11

Show how the flag register is affected by

```
        MOV    AL,9CH       ;AL=9CH
        MOV    DH,64H       ;DH=64H
        ADD    AL,DH        ;now AL=0
```

Solution:

	9C	1001	1100
+	64	0110	0100
	00	0000	0000

CF = 1 since there is a carry beyond d7
AF = 1 since there is a carry from d3 to d4
PF = 1 since there is an even number of 1s in the result
ZF = 1 since the result is zero
SF = 0 since d7 of the result is zero

flag register and ADD instruction

- It is important to note differences between 8- and 16-bit operations in terms of impact on the flag bits.
  - The parity bit only counts the lower 8 bits of the result and is set accordingly.

Example 1-12

Show how the flag register is affected by

```
        MOV    AX,34F5H      ;AX= 34F5H
        ADD    AX,95EBH      ;now AX= CAE0H
```

Solution:

	34F5	0011	0100	1111	0101
+	95EB	1001	0101	1110	1011
	CAE0	1100	1010	1110	0000

CF = 0 since there is no carry beyond d15

AF = 1 since there is a carry from d3 to d4

PF = 0 since there is an odd number of 1s in the lower byte

ZF = 0 since the result is not zero

SF = 1 since d15 of the result is one

flag register and ADD instruction

- The carry flag is set if there is a carry beyond bit d15 instead of bit d7.
  - Since the result of the entire 16-bit operation is zero (meaning the contents of BX), ZF is set to high.

Example 1-13

Show how the flag register is affected by

```
MOV   BX,AAAAH      ;BX= AAAAH
ADD   BX,5556H      ;now BX= 0000H
```

Solution:

	AAAA	1010	1010	1010	1010
+	5556	0101	0101	0101	0110
	0000	0000	0000	0000	0000

CF = 1 since there is a carry beyond d15

AF = 1 since there is a carry from d3 to d4

PF = 1 since there is an even number of 1s in the lower byte

ZF = 1 since the result is zero

SF = 0 since d15 of the result is zero

flag register and ADD instruction

- Instructions such as data transfers (MOV) affect no flags.

Example 1-14

Show how the flag register is affected by

```
MOV   AX,94C2H     ;AX=94C2H
MOV   BX,323EH     ;BX=323EH
ADD   AX,BX        ;now AX=C700H
MOV   DX,AX        ;now DX=C700H
MOV   CX,DX        ;now CX=C700H
```

Solution:

	94C2		1001	0100	1100	0010
+	323E		0011	0010	0011	1110
	C700		1100	0111	0000	0000

After the ADD operation, the following are the flag bits:

CF = 0 since there is no carry beyond d15

AF = 1 since there is a carry from d3 to d4

PF = 1 since there is an even number of 1s in the lower byte

ZF = 0 since the result is not zero

SF = 1 since d15 of the result is 1

use of the zero flag for looping

- A widely used application of the flag register is the use of the zero flag to implement program loops.
  - A *loop* is a set of instructions repeated a number of times
- More on details on LOOPS later!

use of the zero flag for looping

- As an example, to add 5 bytes of data, a *counter* can be used to keep track of how many times the loop needs to be repeated.
  - **Each time the addition is performed the counter is decremented and the zero flag is checked.**
    - **When the counter becomes zero, the zero flag is set (ZF = 1) and the loop is stopped.**

```
        MOV   CX,05     ;CX holds the loop count
        MOV   BX,0200H  ;BX holds the offset data address
        MOV   AL,00     ;initialize AL
ADD_LP: ADD   AL,[ BX]  ;add the next byte to AL
        INC   BX        ;increment the data pointer
        DEC   CX        ;decrement the loop counter
        JNZ   ADD_LP    ;jump to next iteration if counter
                         not zero
```

use of the zero flag for looping

- Register CX is used to hold the counter.
  - BX is the offset pointer.
    - (SI or DI could have been used instead)

```
        MOV   CX,05      ;CX holds the loop count
        MOV   BX,0200H   ;BX holds the offset data address
        MOV   AL,00      ;initialize AL
ADD_LP: ADD   AL,[ BX]   ;add the next byte to AL
        INC   BX         ;increment the data pointer
        DEC   CX         ;decrement the loop counter
        JNZ   ADD_LP     ;jump to next iteration if counter
                          not zero
```

use of the zero flag for looping

- AL is initialized before the start of the loop
  - In each iteration, ZF is checked by the JNZ instruction
    - JNZ stands for "Jump Not Zero“, meaning that if ZF = 0, jump to a new address.
    - If ZF = 1, the jump is *not* performed, and the instruction below the jump will be executed.

```
        MOV   CX,05     ;CX holds the loop count
        MOV   BX,0200H  ;BX holds the offset data address
        MOV   AL,00     ;initialize AL
ADD_LP: ADD   AL,[BX]   ;add the next byte to AL
        INC   BX        ;increment the data pointer
        DEC   CX        ;decrement the loop counter
        JNZ   ADD_LP    ;jump to next iteration if counter
                         not zero
```

use of the zero flag for looping

- JNZ instruction must come *immediately after* the instruction that decrements CX.
  - JNZ needs to check the effect of "DEC CX" on ZF.
    - If any instruction were placed between them, that instruction might affect the zero flag.

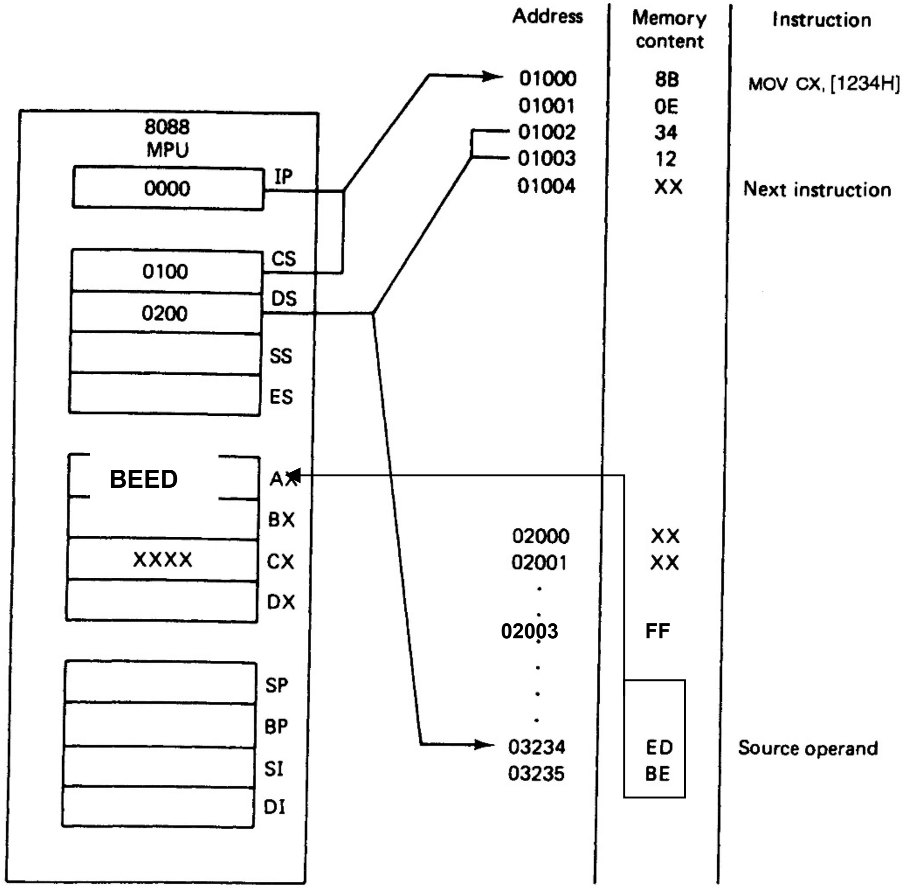
```
        MOV   CX,05      ;CX holds the loop count
        MOV   BX,0200H   ;BX holds the offset data address
        MOV   AL,00      ;initialize AL
ADD_LP: ADD   AL,[ BX]   ;add the next byte to AL
        INC   BX         ;increment the data pointer
        DEC   CX         ;decrement the loop counter
        JNZ   ADD_LP     ;jump to next iteration if counter
                          not zero
```

Addressing Modes

- **Register Addressing Mode**
  - MOV AX, BX
  - MOV ES,AX
  - MOV AL,BH
- **Immediate Addressing Mode**
  - MOV AL,15h
  - MOV AX,2550h
  - MOV CX,625

Direct Addressing Mode

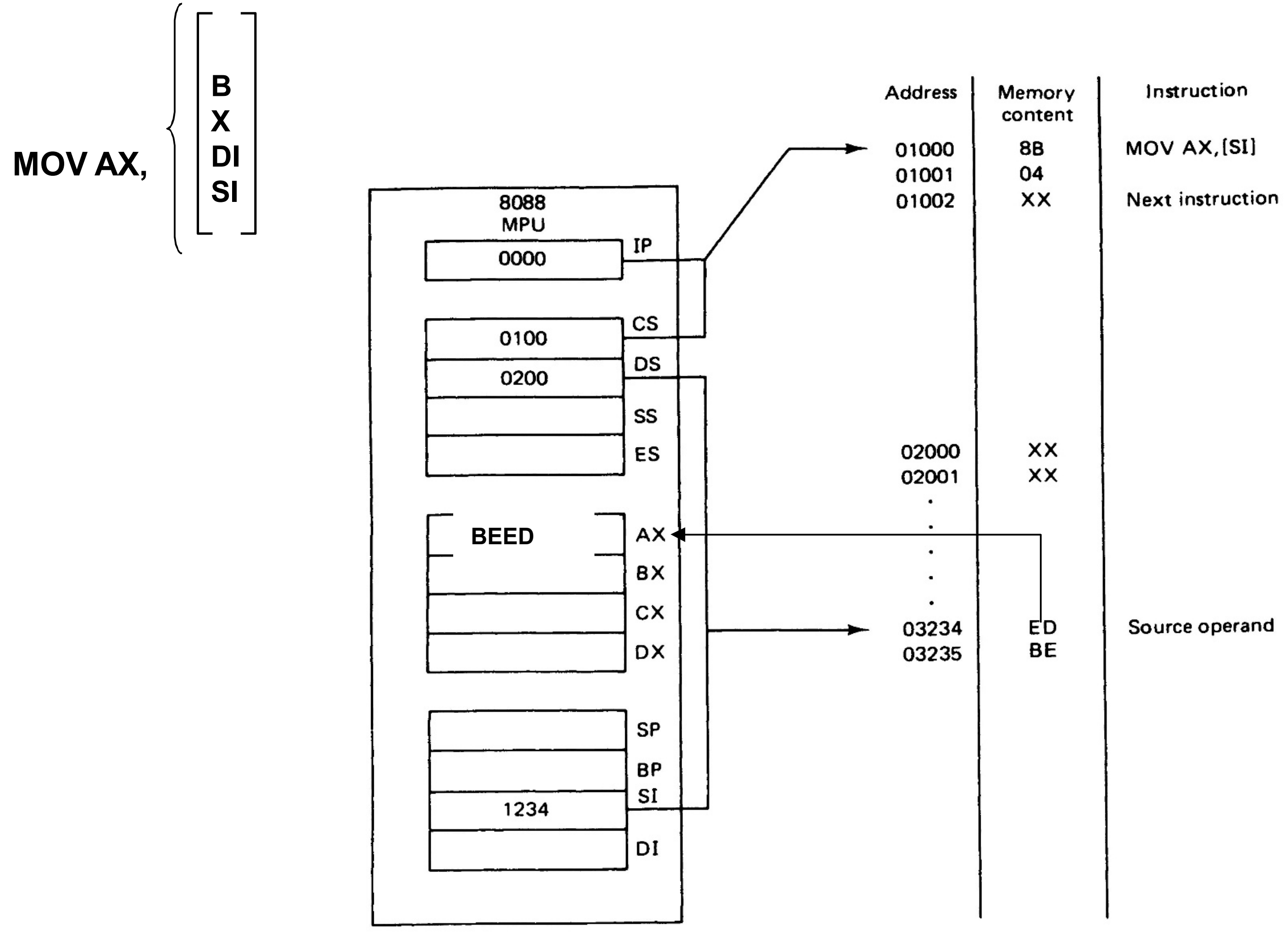
MOV CX, [address]



Example:
MOV AL,[03]

AL=?

Register Indirect Addressing Mode



Example for Register Indirect Addressing

- Assume that DS=1120, SI=2498 and AX=17FE show the memory locations after the execution of:

MOV [SI],AX

DS (Shifted Left) + SI = 13698.

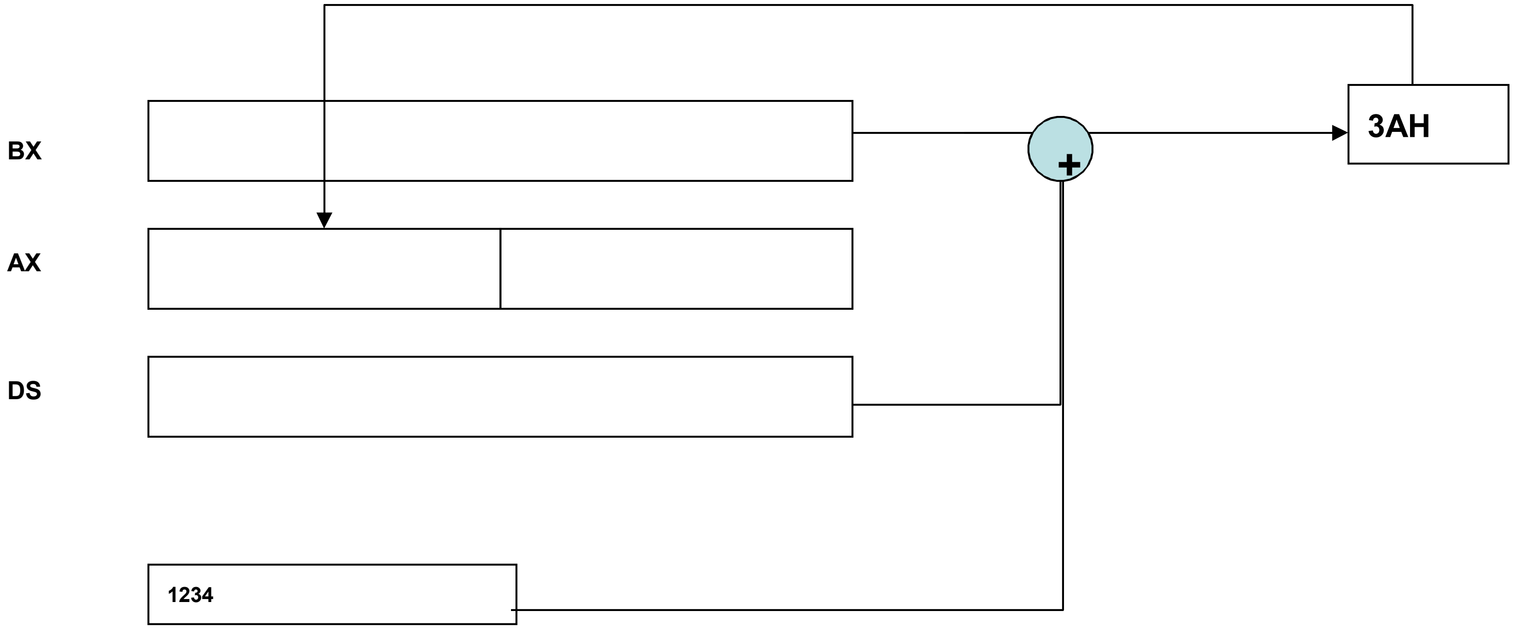
With little endian convention:

Low address 13698 → FE

High Address 13699 → 17

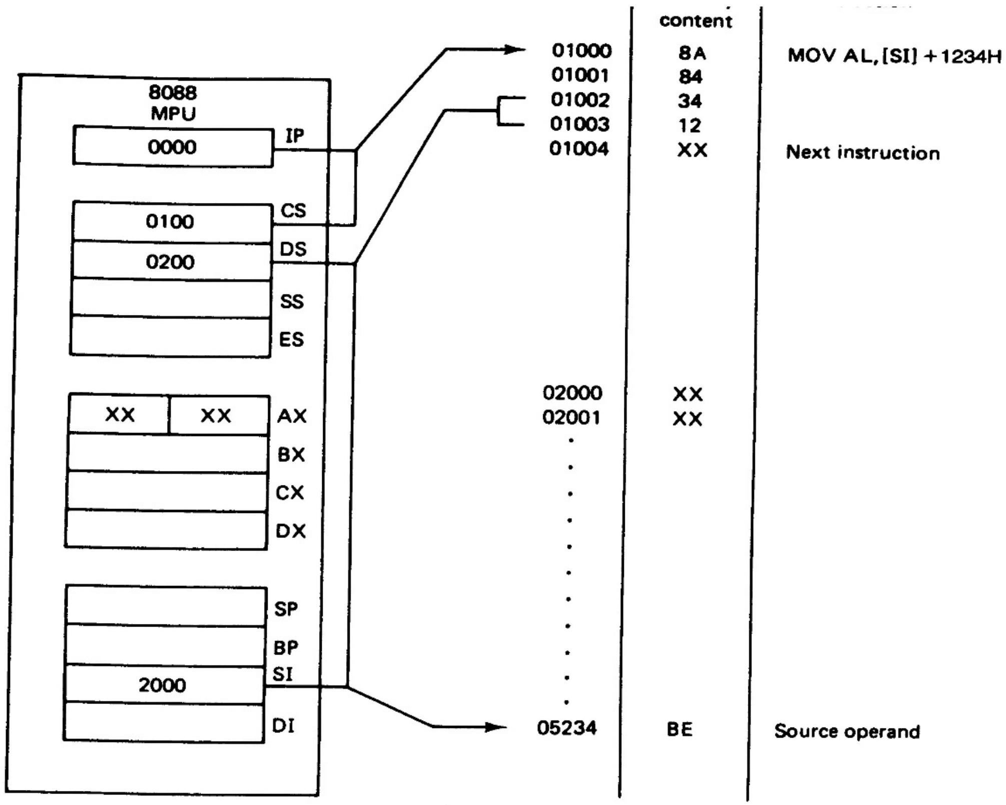
Based-Relative Addressing Mode

MOV AH, [DS:BX / SS:BP] + 1234h



Indexed Relative Addressing Mode

MOV AH, [SI/DI] + 1234h



(a)

Example: What is the physical address MOV [DI-8],BL if DS=200 & DI=30h ?
DS:200 shift left once 2000 + DI + -8 = 2028

Based-Indexed Relative Addressing Mode

- Based Relative + Indexed Relative
- We must calculate the PA (physical address)

	CS					
	SS		BX	SI		8 bit displacement
PA=	DS	:	BP	+ DI	+	16 bit displacement
	ES					

MOV AH,[BP+SI+29]
or
MOV AH,[SI+29+BP]
or
MOV AH,[SI][BP]+29

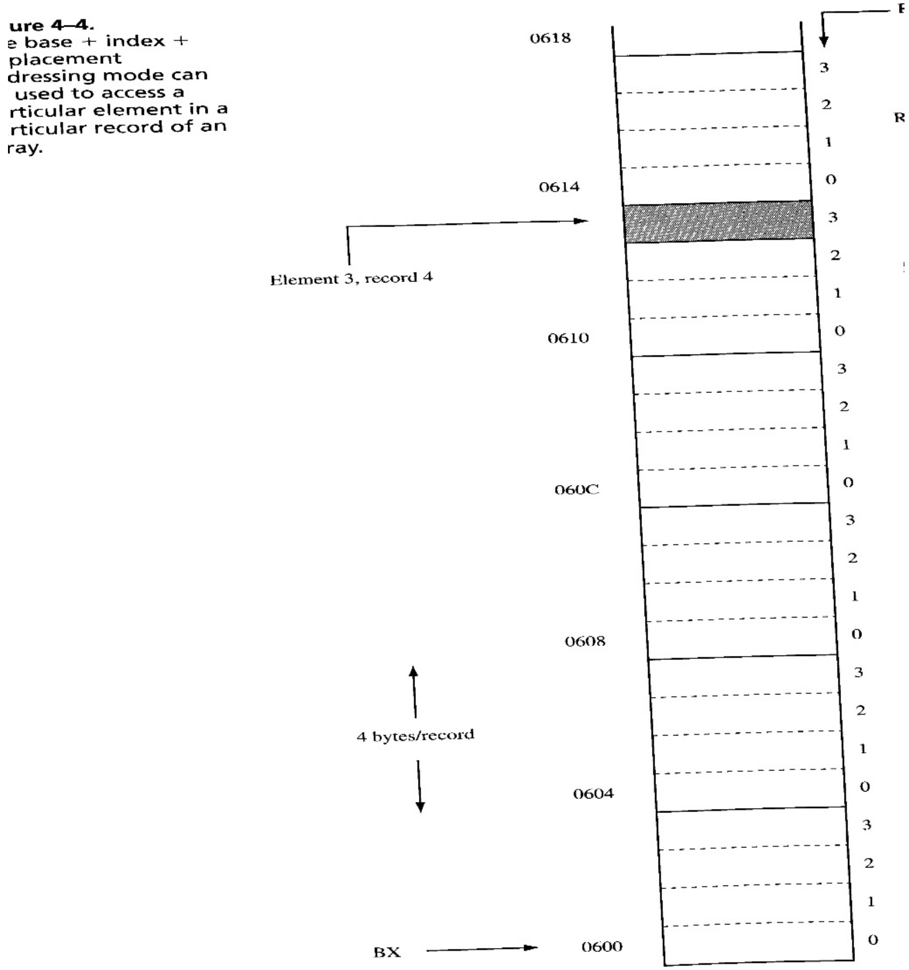
The register order does not matter

Based-Indexed Addressing Mode

```
MOV BX, 0600h
MOV SI, 0010h ; 4 records, 4 elements each.
MOV AL, [BX + SI + 3]
```

OR

```
MOV BX, 0600h
MOV AX, 004h ;
MOV CX,04;
MUL CX
MOV SI, AX
MOV AL, [BX + SI + 3]
```



Summary of the addressing modes

Addressing Mode	Operand	Default Segment
Register	Reg	None
Immediate	Data	None
Direct	[offset]	DS
Register Indirect	[BX] [SI] [DI]	DS DS DS
Based Relative	[BX]+disp [BP]+disp	DS SS
Indexed Relative	[DI]+disp [SI]+disp	DS DS
Based Indexed Relative	[BX][SI or DI]+disp [BP][SI or DI]+disp	DS SS

16 bit Segment Register Assignments

Segment Registers	CS	DS	ES	SS
Offset Register	IP	SI,DI,BX	SI,DI,BX	SP,BP

Type of Memory Reference	Default Segment	Alternate Segment	Offset
Instruction Fetch	CS	none	IP
Stack Operations	SS	none	SP,BP
General Data	DS	CS,ES,SS	BX, address
String Source	DS	CS,ES,SS	SI, DI, address
String Destination	ES	None	DI

Segment override

Instruction Examples	Override Segment Used	Default Segment
MOV AX,CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

Example for default segments

- The following registers are used as offsets. Assuming that the default segment used to get the logical address, give the segment register associated?

a) BP b)DI c)IP d)SI, e)SP, f) BX

- Show the contents of the related memory locations after the execution of this instruction

  MOV [BP][SI]+10,DX

  if DS=2000, SS=3000,CS=1000,SI=4000,BP=7000,DX=1299 (all hex)

SS(0)=30000
30000+4000+7000+10=3B010

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

Languages in terms of applications

- One of the most beneficial uses of assembly language programming is **real-time applications**.
- Real time means the task required by the application must be completed before any other input to the program that will alter its operation can occur
- For example the device service routine which controls the operation of the floppy disk drive is a good example that is usually written in assembly language
- Assembly language not only good for controlling hardware devices but also **performing pure software operations**
  - searching through a large table of data for a special string of characters
  - Code translation from ASCII to EBCDIC
  - Table sort routines
  - Mathematical routines
- Assembly language: perform real-time operations
- High-level languages: Those operations mostly not critical in time.

Converting Assembly Language Instructions to Machine Code

OPCODE	D	W	MOD	REG	R/M	○○○

- An instruction can be coded with 1 to 6 bytes
- **<u>Byte 1 contains three kinds of information:</u>**
  - Opcode field (6 bits) specifies the operation such as add, subtract, or move
  - Register Direction Bit (D bit)
    - Tells the register operand in REG field in byte 2 is source or destination operand
      - 1:Data flow to the REG field from R/M
      - 0: Data flow from the REG field to the R/M
  - Data Size Bit (W bit)
    - Specifies whether the operation will be performed on 8-bit or 16-bit data
      - 0: 8 bits
      - 1: 16 bits
- **<u>Byte 2 has two fields:</u>**
  - Mode field (MOD) – 2 bits
  - Register field (REG) - 3 bits
  - Register/memory field (R/M field) – 2 bits

Continued

- **REG field is used to identify the register for the first operand**

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Continued

- 2-bit MOD field and 3-bit R/M field together specify the second operand

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

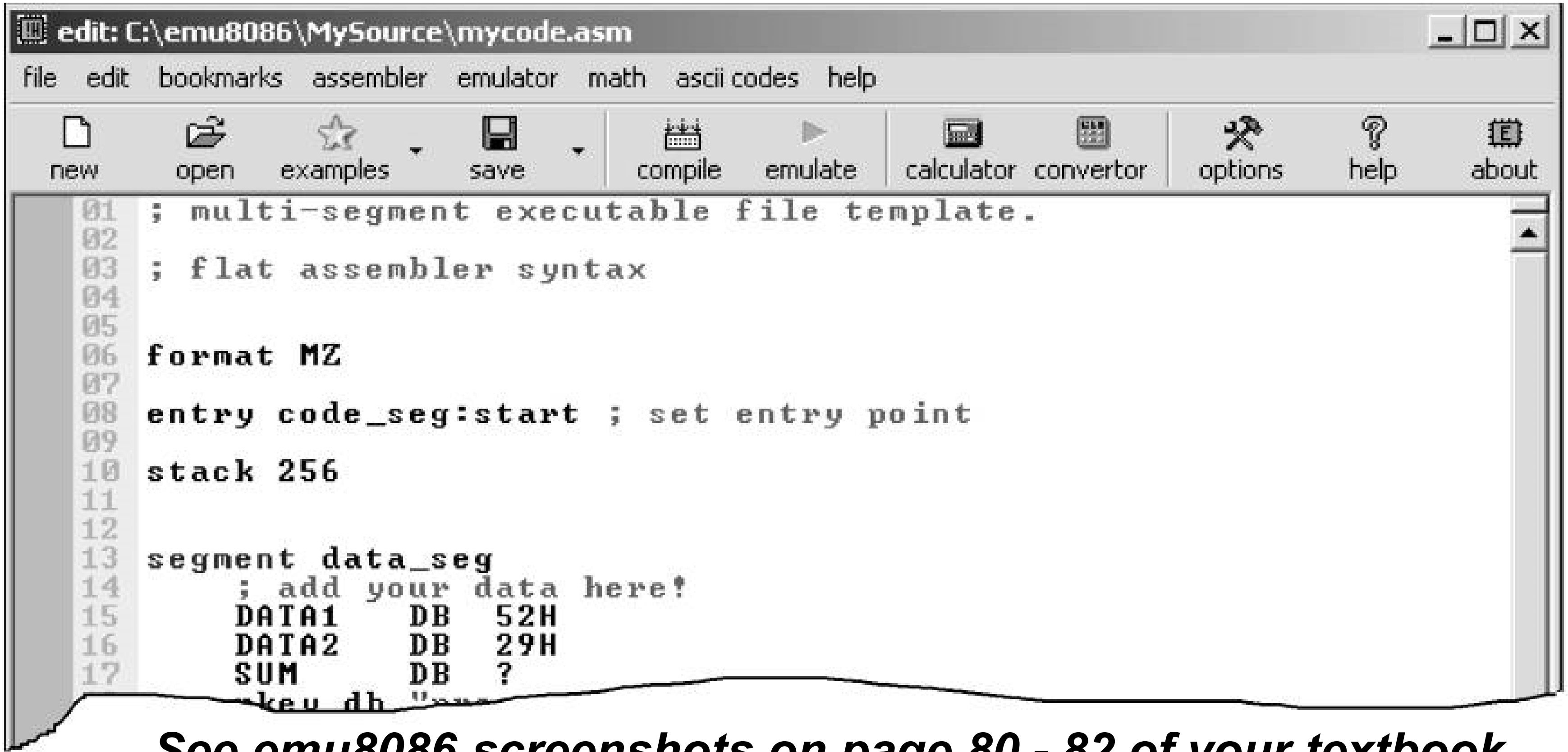
(a)

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

(b)

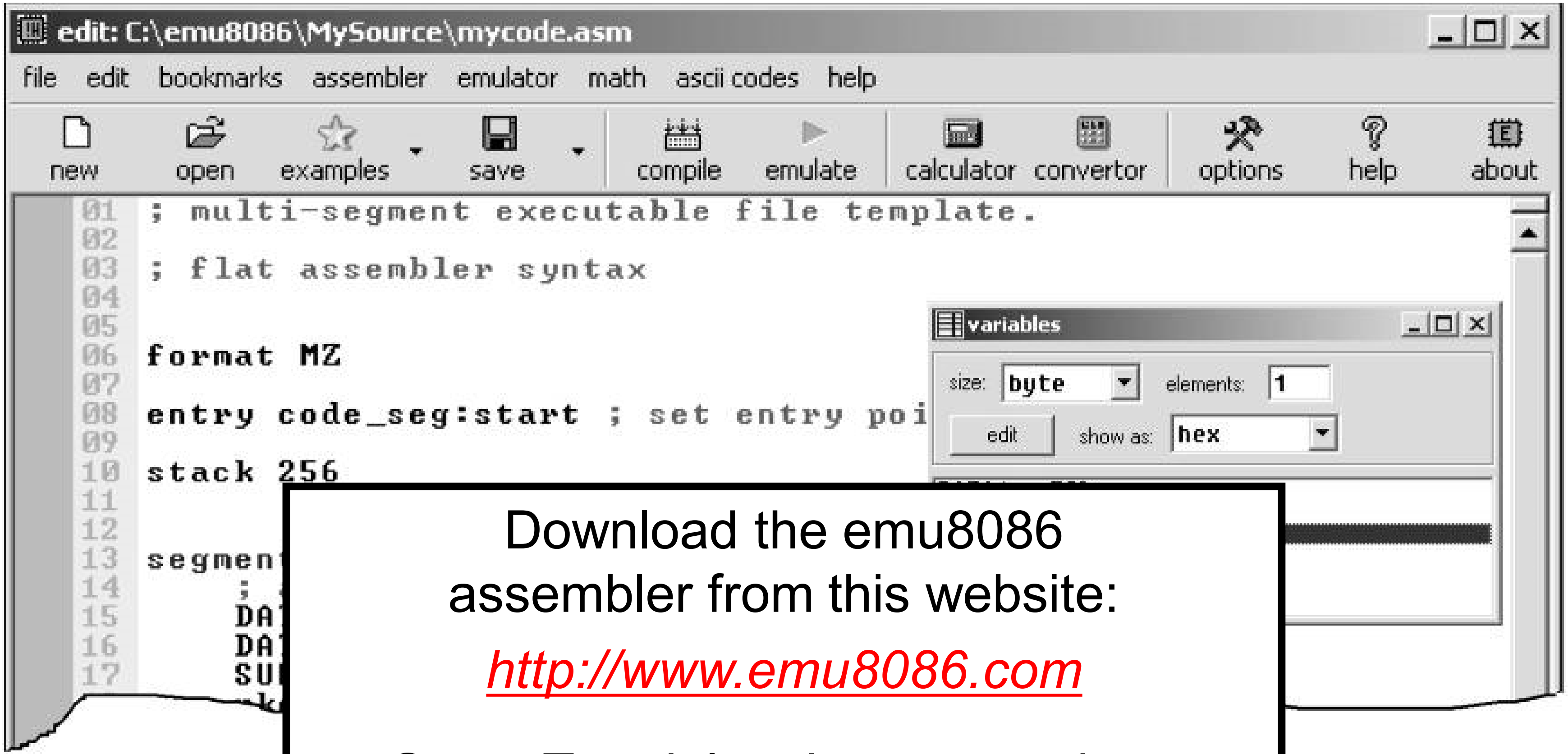
2.6: FULL SEGMENT DEFINITION
the emu8086 assembler

- 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

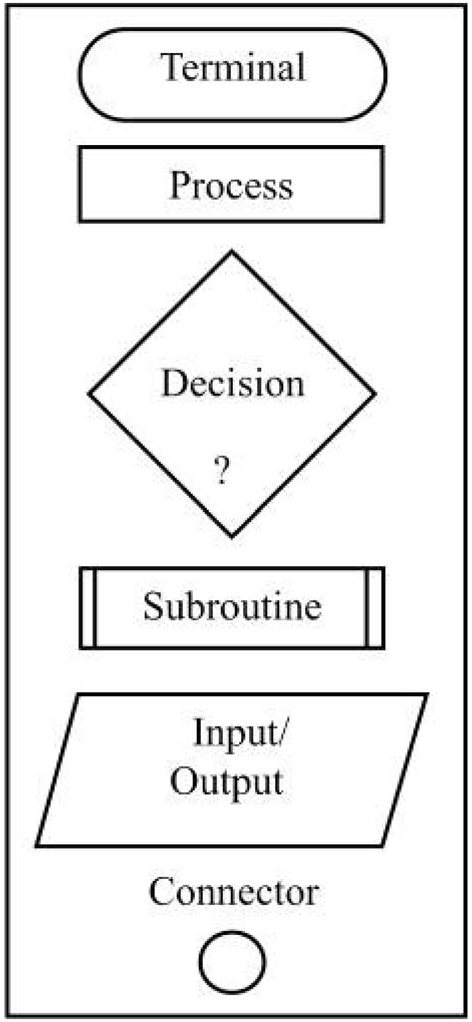
- Principles a structured program should follow:
  - The program should be designed *before* it is coded.
    - By using flowcharting or pseudocode, the design is clear those coding, as well as those maintaining the program later.
  - Use comments within the program and documentation.
    - This will 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

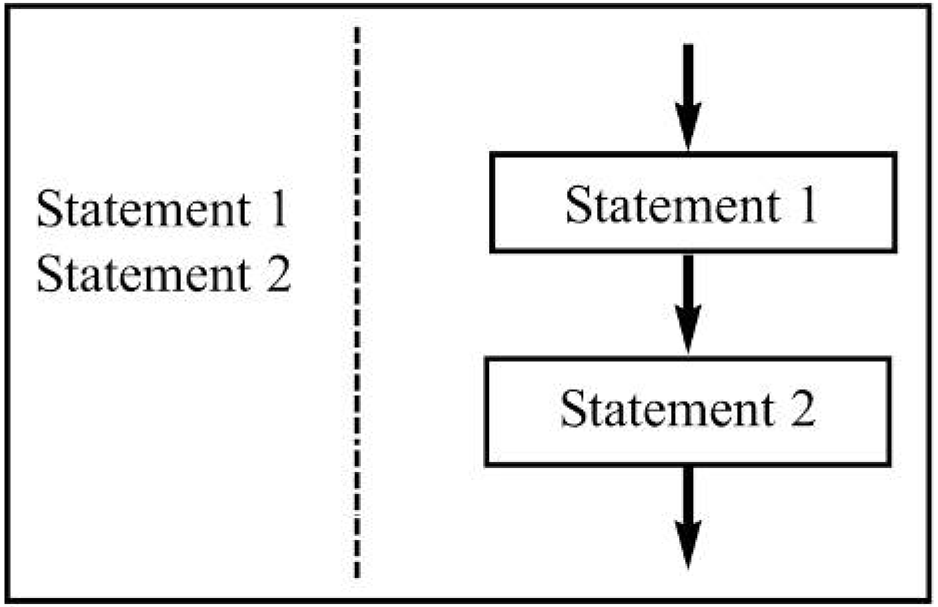
- Flowcharts use graphic symbols to represent different types of program operations.
  - The symbols are connected together to show the flow of execution of the program.
    - Flowcharting has been standard industry practice for decades.
  - Flowchart templates help you draw the symbols quickly and neatly.



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.

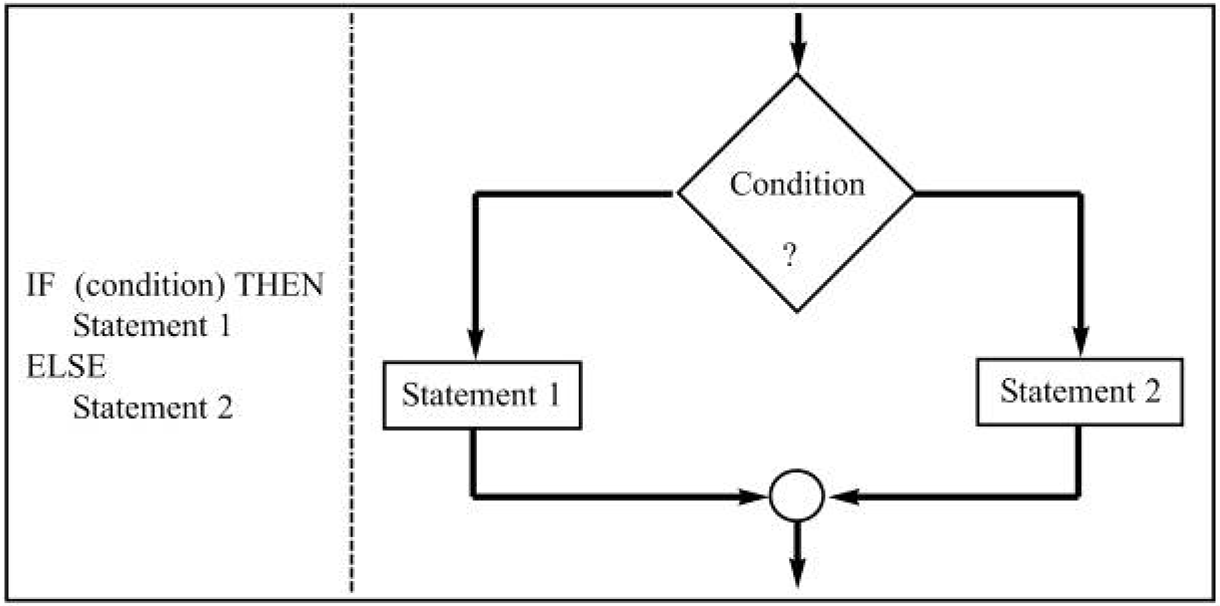


Figure 2-16
IF-THEN-ELSE
Pseudocode vs. Flowchart

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.

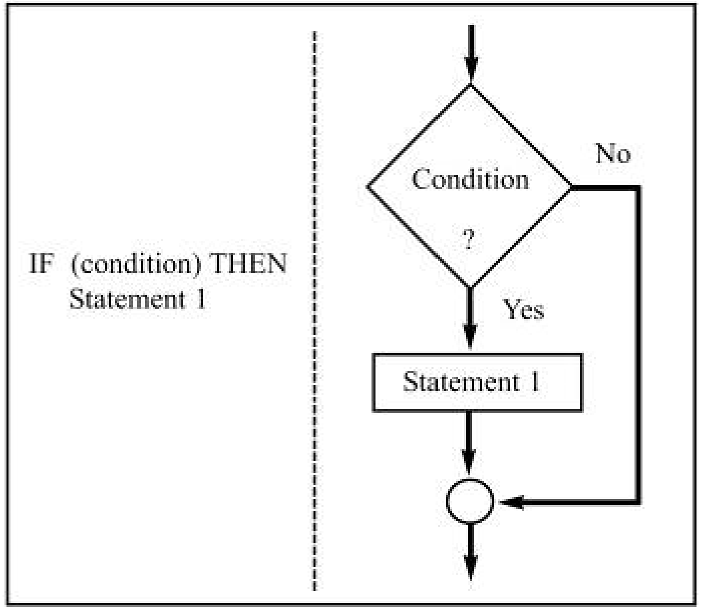
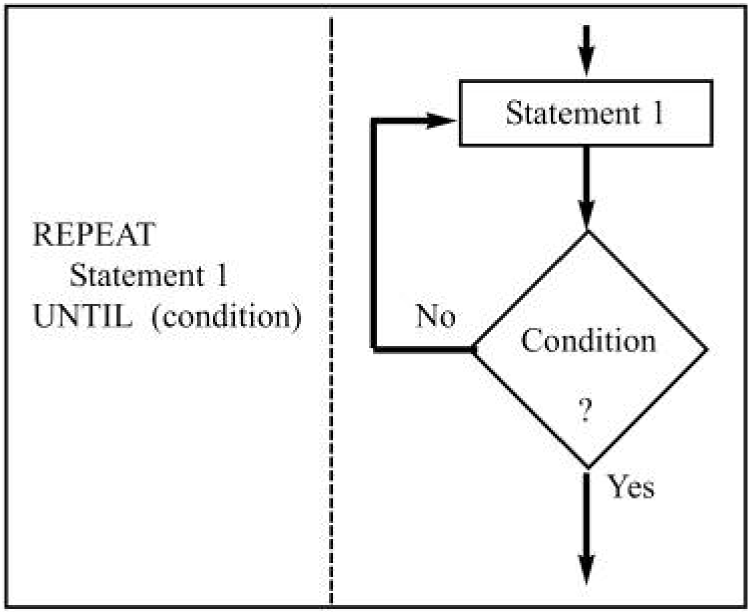


Figure 2-17
IF-THEN
Pseudocode vs. Flowchart

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.

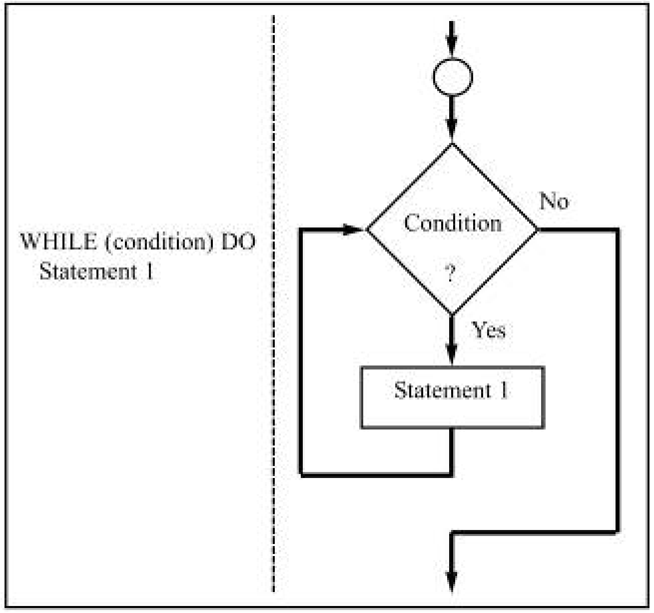


Figure 2-19
WHILE-DO
Pseudocode vs. Flowchart

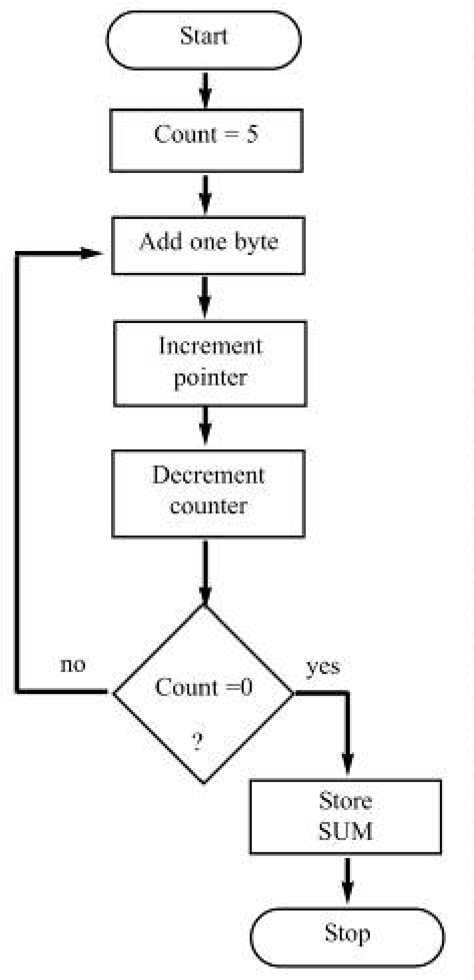
WHILE-DO may not execute the statement(s) at all, as the condition is checked at the beginning of each iteration.

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
            ADD    AL,[BX]
```

Flowchart vs. pseudocode for Program 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.



Count = 5

Repeat
    Add next byte
    Increment pointer
    Decrement count
Until Count = 0

Store SUM

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.

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

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

- There are 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 or EMU8086 etc.
    - 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.

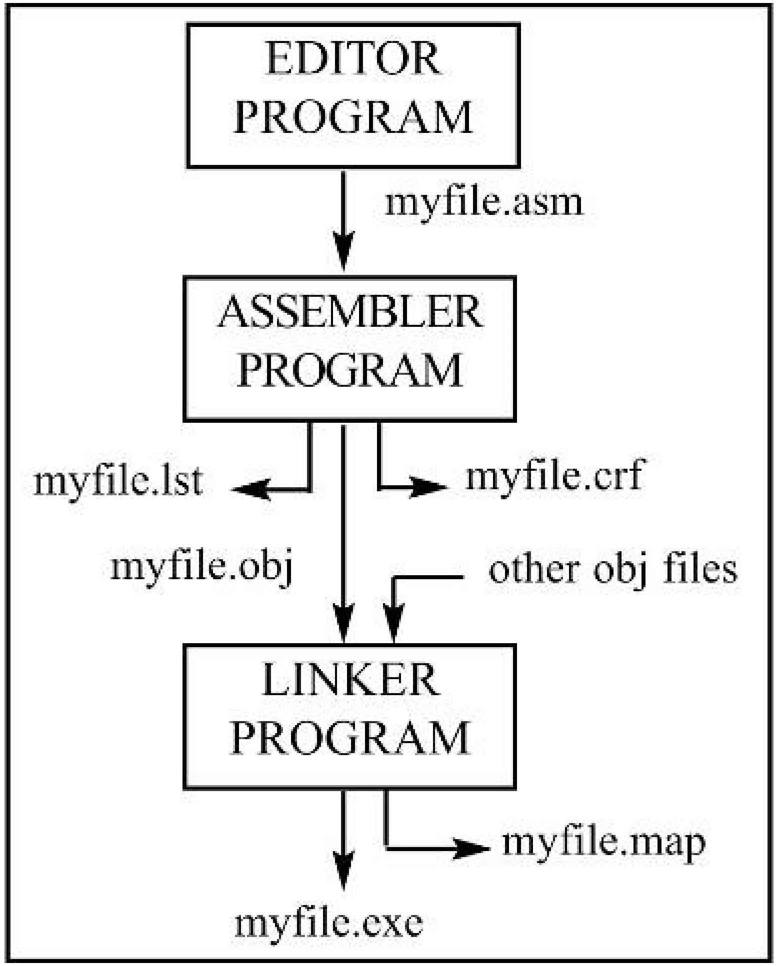
ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It tells compiler that the executable file will be loaded at the **offset** of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their **offsets**. Directives are never converted to any real **machine code**.

Why executable file is loaded at **offset** of **100h**? Operating system keeps some data about the program in the first 256 bytes of the

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
LINKing the program

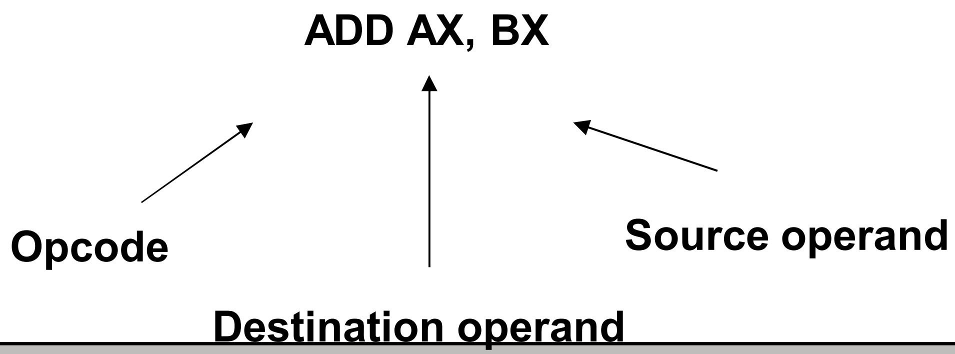
- The assembler creates the opcodes, operands & offset addresses under the ".obj" file.
- The LINK program produces the ready-to-run program with the ".exe" (EXEcutable) extension.
  - The LINK program sets up the file so it can be loaded by the OS and executed.
- The program can be run at the OS level, 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
TITLE directives

- 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.



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

- 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
```

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]

```
DATA1       DB    52H
DATA2       DB    29H
SUM         DB    ?

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
```

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.

2.4: 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.
  - The period can only be used as the first character.

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.

Compiler directives

Syntax for a variable declaration:

name **DB** *value*

name **DW** *value*

DB - stays for Define Byte.
DW - stays for Define Word.

name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "**?**" symbol for variables that are not initialized.

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".

2.1: 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.

DataTypes and Data Definition

```
DATA1   DB   25
DATA2   DB   10001001b
DATA3   DB   12h
        ORG  0010h ;indicates distance  from initial DS location
DATA4   DB   "2591"
        ORG   0018h ;indicates distance from initial DS location
DATA5   DB   ?
```

This is how data is initialized in the data segment

```
0000    19
0001    89
0002    12
0010    32 35 39 31
0018    00
```

DB DW DD

```
.data
        MESSAGE2 DB '1234567'
        MESSAGE3 DW 6667H
        data1 db 1,2,3
        db 45h
        db 'a'
        db 11110000b
        data2 dw 12,13
        dw 2345h
        dd 300h
```

```
; how it looks like in memory
31 32 33 34 35 36 37
67 66
1 2 3
45
61
F0
0C 00 0D 00
45 23
00 03 00 00
```

More Examples

```
DB   6 DUP(FFh); fill 6 bytes with ffh

DW  954
DW  253Fh     ; allocates two bytes
DW  253Fh

DD 5C2A57F2h   ;allocates four bytes
DQ     12h     ;allocates eight bytes

COUNTER1  DB  COUNT
COUNTER2  DB  COUNT
```

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
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 directive assigns a symbolic name to a string or constant.
  - Maxint equ 0ffffh
  - COUNT EQU 2
- EQU for the counter constant in the immediate addressing mode:

  COUNT EQU 25
- 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 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, 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 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..........
```

More assembly – OFFESET, SEG, EQU

- OFFSET
  - The offset operator returns the distance of a label or variable from the beginning of its segment. The destination must be 16 bits
  - mov BX, offset count
- SEG
  - The segment operator returns the segment part of a label or variable's address.

```
Push DS
Mov AX, seg array
Mov DS, AX
Mov BX, offset array
.
Pop DS
```

DUP (Duplicate)

- DUP operator only appears after a storage allocation directive.
  - db 20 dup(?)

number DUP (value(s))
number - number of duplicate to make (any constant value).
value - expression that DUP will duplicate.

for example:
c DB 5 DUP(9)
is an alternative way of declaring:
c DB 9, 9, 9, 9, 9

one more example:
d DB 5 DUP(1, 2)
is an alternative way of declaring:
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2

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
                        ]
                  ]
```

The PTR Operator - Byte or word or doubleword?

- INC [20h] ; is this byte/word/dword? or
- MOV [SI],5
  - Is this byte 05?
  - Is this word 0005?
  - Or is it double word 00000005?
- To clarify we use the PTR operator
  - INC BYTE PTR [20h]
  - INC WORD PTR [20h]
  - INC DWORD PTR [20h]
- or for the MOV example:
  - MOV byte ptr [SI],5
  - MOV word ptr[SI],5

The PTR Operator

- Would we need to use the PTR operator in each of the following?

```
MOV AL,BVAL
MOV DL,[BX]
SUB [BX],2
MOV CL,WVAL
ADD AL,BVAL+1

.data
BVAL DB 10H,20H
WVAL DW 1000H
```

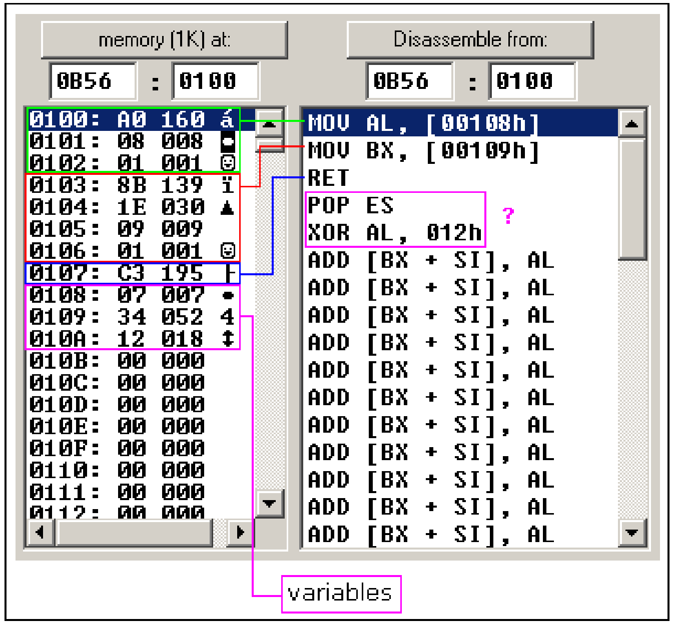
```
MOV AL,BVAL
MOV DL,[BX]
SUB [BX],byte ptr 2
MOV CL,byte ptr WVAL
ADD AL,BVAL+1
```

ORG 100h MOV AL, var1 MOV BX, var2 RET ; stops the program. VAR1 DB 7 var2 DW 1234h

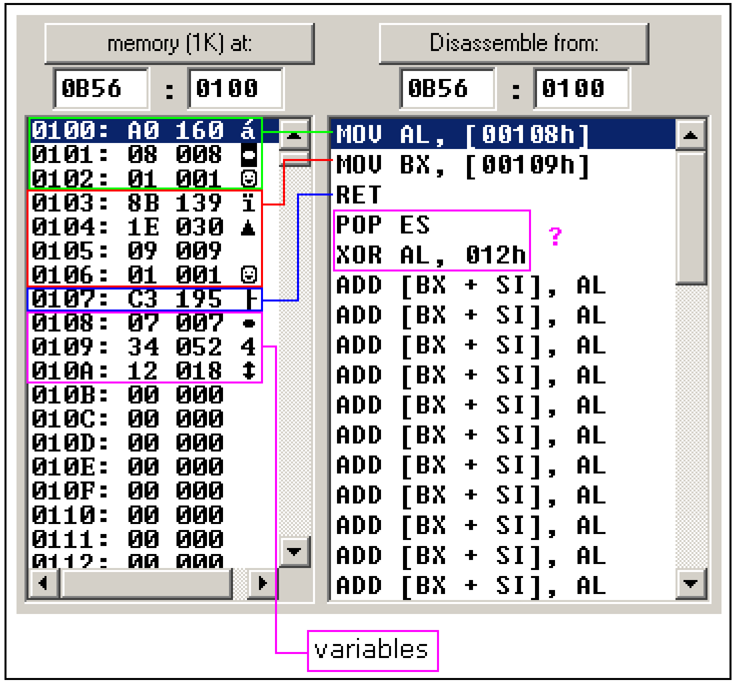
2.5: DATA TYPES AND DATA DEFINITION
ORG origin

- ORG is used to indicate the beginning of the offset address.
  - **ORG 100h** is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It tells compiler that the executable file will be loaded at the **offset** of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their **offsets**. Directives are never converted to any real **machine code**.

```
ORG 100h

MOV AL, var1
MOV BX, var2

RET   ; stops the program.

VAR1 DB 7
var2 DW 1234h
```



Equivalent code using only DB



ORG 100h

DB 0A0h
DB 08h
DB 01h

DB 8Bh
DB 1Eh
DB 09h
DB 01h

DB 0C3h

DB 7

DB 34h
DB 12h

Procedures

- *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.

Procedures

- The syntax for procedure declaration:

```
name PROC
    ; here goes the code
    ; of the procedure ...
RET
name ENDP
```

Example Proc

```
ORG    100h
main proc ; this is optional but very strongly recommended
MOV    AL, 1
MOV    BL, 2

CALL   m2
CALL   m2
CALL   m2
CALL   m2

RET                ; return to operating system.
main endp ; this is optional but very  strongly recommended

m2     PROC
MUL    BL          ; AX = AL * BL.
RET                ; return to caller.
m2     ENDP

END ;main program should end with END
```

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.

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

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

```
TITLE ADD_5_BYTES
org 100h
DATA_IN DB 25H,12H,15H,1FH,2BH
SUM     DB ?
MAIN PROC FAR
           MOV AX,@DATA
           MOV DS, AX
           MOV CX,5
           MOV BX, OFFSET DATA_IN
           MOV AL,0
           CALL  ADDC
           MOV SUM, AL
           MOV AH, 4CH
           INT 21H
           RET
MAIN ENDP
ADDC PROC ; A PROCEDURE USED!!!!!!
           AGAIN: ADD AL, [BX]
                     INC BX
                     DEC CX
           JNZ AGAIN
           RET
ADDC ENDP
END
```

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
analysis of Program 2-2

Write a program that adds four words of data and saves the result. The values will be 234DH,1DE6H, 3BC7H and 566AH. Verify the result is: D364H

```
TITLE ADDS_4_words_data
ORG 100H
DATA_IN DW 234DH, 1DE6H, 3BC7H,566AH
ORG 10H
SUM DW ? ; The 16-bit data (a word) is stored with the low-order byte first, referred to as "little endian.“
MAIN PROC FAR
        MOV AX,@DATA
        MOV DS, AX
        MOV CX,4
        MOV DI, OFFSET DATA_IN
        MOV BX,00
        CALL ADD_16
        MOV SI, OFFSET SUM
        MOV [SI], BX
        MOV AH, 4CH
        INT 21H
MAIN ENDP
END
```

```
ADD_16 PROC
        ADD_LP: ADD BX,[DI]
        INC DI
        INC DI
        DEC CX
        JNZ ADD_LP
        RET
ENDP ADD_16
```

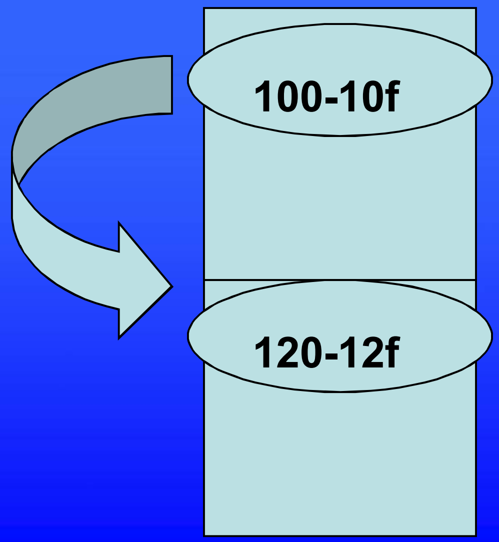
2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-2

- 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.

Example program

Copy the contents of a block of memory (X bytes) starting at location SI to another block of memory starting at DIh

```
        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
```



2.3: MORE SAMPLE PROGRAMS
analysis of Program 2-3

- ACTUAL EXAMPLE TO RUN

```
TITLE TRANSFER_6_BYTES
ORG 100H
DATA_IN DB 25H,4FH,85H,1FH,2BH,0C4H
ORG 10H
COPY DB 6 DUP (?)
MAIN PROC FAR
          MOV AX,@DATA
          MOV DS, AX
          MOV SI,OFFSET DATA_IN
          MOV DI,OFFSET COPY
          MOV CX, 06H
          MOV_LOOP: MOV AL,[SI]
            MOV [DI],AL
            INC SI
            INC DI
          DEC CX
          JNZ MOV_LOOP
          MOV AH,4CH
          INT 21H
MAIN ENDP
END
```

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.

2.6: 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.

```
LABEL SEGMENT DATA
        DATA_IN DB 25H,4FH,85H,1FH,2BH,0C4H
        ORG 10H
        COPY DB 6 DUP (?)
END SEGMENT DATA
```

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

- 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

- rewritten using full segment definition.

```
TITLE TRANSFER
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
MOV DI, OFFSET COPY
MOV CX,06H
MOV_LOOP: MOV AL,[SI]
MOV [DI],AL
INC SI
INC DI
DEC CX
JNZ MOV_LOOP
MOV AH,4CH
INT 21H
MAIN ENDP
CDSEG ENDS
END MAIN
```

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.

```
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
```

```
data segment
    DATA_IN DW 234DH, 1DE6H, 3BC7H,566AH
    SUM DW ? ;referred to as "little endi
ends

stack segment
    dw   128  dup(0)
ends
```

```
code segment
main proc
start:
        mov ax,data
        mov ds,ax
        MOV CX,4
        MOV DI, OFFSET DATA_IN
        MOV BX,00
        ADD_LP: ADD BX,[DI]
        INC DI
        INC DI
        DEC CX
        JNZ ADD_LP
        MOV SI, OFFSET SUM
        MOV [SI], BX
        MOV AH, 4CH
        INT 21H
        ret
end main
ends
```

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.

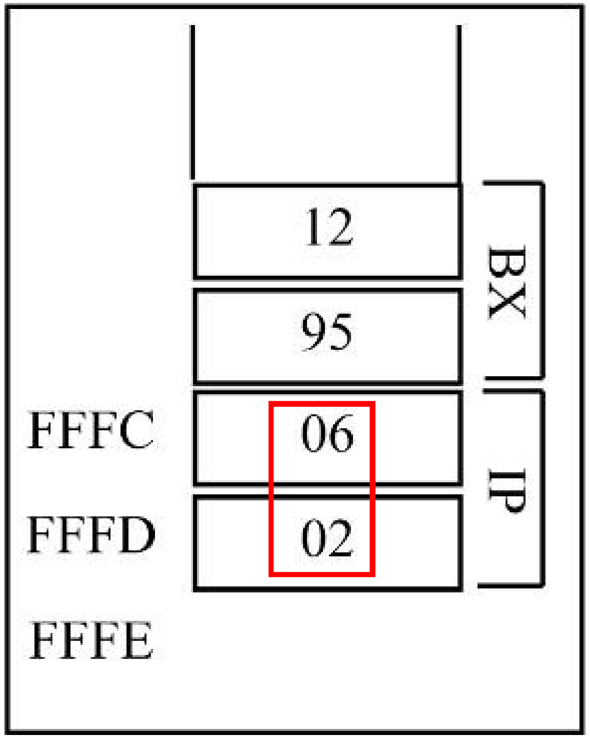
2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- 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
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..
- 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.

PEARSON
The x86 PC
Assembly Language, Design, and Interfacing
By Muhammad Ali Mazidi, Janice Gillespie Mazidi and Danny Causey

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
```

```
Dec  Hex  Bin
2    2    00000010
```

ENDS ; THREE

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