15CS – 44

MICROPROCESSORS AND MICROCONTROLLERS

MODULE 1

THE x86 MICROPROCESSOR

Mahesh Prasanna K.

Dept. of CSE, VCET.

BRIEF HISTORY OF THE x86 FAMILY

- A study of history is not essential to understand the microprocessor, but it provides a historical perspective of the fast-paced evolution of the computer

Evolution from 8080/8085 to 8086:

- In 1978 – Intel Corporation – a 16-bit microprocessor – 8086
- In 1979 – Intel Corporation – a 16-bit microprocessor – 8088

The Intel 8085

- 8-bit non pipelined microprocessor
- Addressed 64K bytes of memory
- Can execute 769,230 instructions per second
- Its instruction set contained 246 instructions

The Intel 8086/8088

- 16-bit pipelined microprocessors
- Addressed 1M bytes (1M byte = 1024K bytes = 1024 * 1024 bytes = 1,048,576 bytes) of memory
- Executed 2.5 MIPs (millions of instructions per second)
- Its instruction set contained over 20,000 instructions
- A small 6- or 4-byte instruction cache or queue that pre-fetched a few instructions before they were executed

Evolution of Intel's Microprocessors (from 8008 to 8088)

Product	**8008**	**8080**	**8085**	**8086**	**8088**
Year introduced	1972	1974	1976	1978	1979
Technology	PMOS	NMOS	NMOS	NMOS	NMOS
Number of pins	18	40	40	40	40
Number of transistors	3000	4500	6500	29,000	29,000
Number of instructions	66	111	113	133	133
Physical memory	16KB	64KB	64KB	1MB	1MB
Virtual memory	None	None	None	None	None
Internal data bus	8	8	8	16	16
External data bus	8	8	8	16	8
Address bus	8	16	16	20	20
Data types	8	8	8	8/16	8/16

The Intel 80286

- 16-bit internal and external data buses
- 24 address lines; which give 16M bytes of memory (2^{24} = 16M bytes)
- The clock speed of 80286 was increased; hence, it executed 4 MIPs
- Virtual memory –swapping data between disk storage and RAM
- The 80286 can operate in one of two modes: *real mode* and *protected mode*

- **The Intel 80386**
  - Internally and externally a 32-bit microprocessor
  - 32-bit address bus; capable of hand ling physical memory of up to 4 gigabytes (2^{32} = 4G bytes)
  - Virtual memory was increased to 64 terabytes (2^{46} = 64T bytes)
- **The Intel 80386SX**
  - Internally identical to 80386 ,icroprocessor
  - 24-bit address bus, which gives a capacity of 16M bytes (2^{24} = 16M bytes) of memory
  - 16-bit external data bus – This makes the 386SX system much cheaper

» The Intel 80486

- The 80486 is available as an 80486DX (contains the numeric coprocessor), or an 80486SX (does not contain numeric coprocessor)
- Executes many of 80386 instructions in one clock period
- 80486 microprocessor improved 80386 numeric coprocessor
- 80486 microprocessor also contains an 8K byte cache memory
- The 80486DX contains a 16K byte cache memory
- When the 80486 is operated at the same clock frequency as an 80386, it performs with about a 50% speed improvement
- The 80486 is available as a 25 MHz, 33 MHz, 50 MHz, 66 MHz, or 100 MHz device
- Note that, all programs written for the 8088/86 will run on 286, 386, and 486 computers

The Intel Pentium

- Submicron fabrication technology – more than 3 million transistors
- The Pentium had speeds of 60 and 66 MHz (twice that of 80486 and over 300 times faster than that of the original 8088)
- Separate 8K cache memory for code and data
- 64-bit external data bus with 32-bit register and 32-bit address bus capable of addressing 4G bytes of memory
- Improved floating-point processor
- Pentium is packaged in a 273-pin PGA chip
- A dual-integer processor, can execute 2 instructions at a time
- It uses BICMOS technology, which combines the speed of bipolar transistors with the power efficiency of CMOS technology.

- **The Intel Pentium Pro**
  - Sixth generation of the x86 family
  - Pentium Pro is an enhanced version of Pentium that uses 5.5 million transistors
  - It was designed to be used for 32-bit servers and workstations

Evolution of Intel's Microprocessors (from 8086 to the Pentium Pro)

Product	**8086**	**80286**	**80386**	**80486**	**Pentium**	**Pentium Pro**
Year introduced	1978	1982	1985	1989	1993	1995
Technology	NMOS	NMOS	CMOS	CMOS	BICMOS	BICMOS
Clock rate (MHz)	3 – 10	10 – 16	16 – 33	25 – 33	60, 66	150
Number of pins	40	68	132	168	273	387
Number of transistors	29,000	134,000	275,000	1.2 million	3.1 million	5.5 million
Physical memory	1MB	16MB	4GB	4GB	4GB	64GB
Virtual memory	None	1GB	64TB	64TB	64TB	64TB
Internal data bus	16	16	32	32	32	32
External data bus	16	16	32	32	64	64
Address bus	20	24	32	32	32	36
Data types	8/16	8/16	8/16/32	8/16/32	8/16/32	8/16/32

Pentium II	Pentium III	Pentium 4	Intel 64 Architecture
✓7.5 million transistor ✓MMX (multi-media extension) ✓Used for servers and workstations	✓9.5 million transistor ✓Instructions to handle video and audio ✓Used for servers and workstations	✓Designed for heavy multimedia processing ✓Operates at 400MHz ✓Used as high end multi-media processing microprocessor	✓64-bit family of processors, formerly called as Merced. ✓Can execute many instructions simultaneously ✓Designed to meet needs of powerful workstations

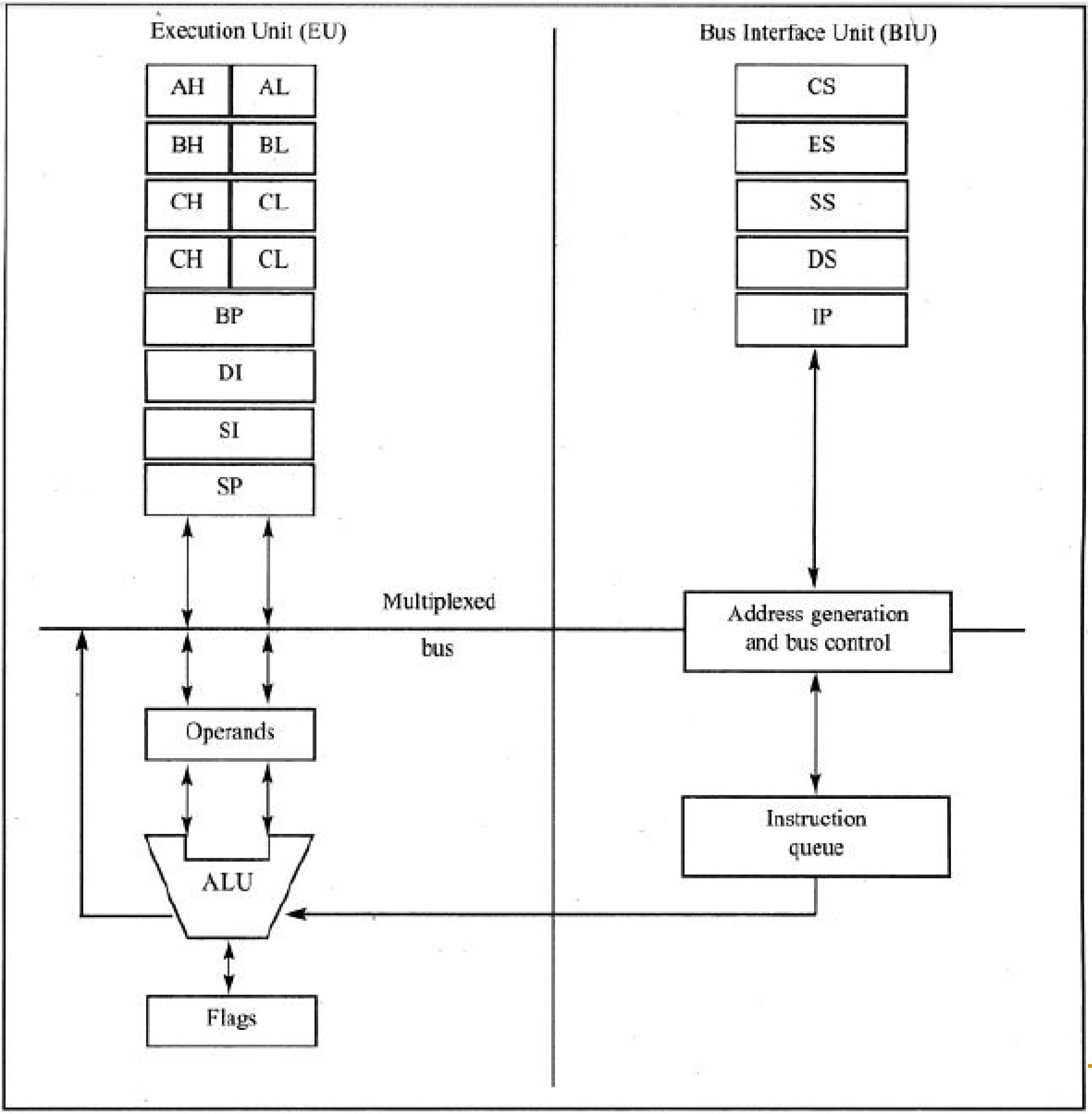
Evolution of Intel’s Microprocessors (from Pentium II to Itanium)

Product	Pentium II	Pentium III	Pentium 4	Itanium II
Year introduced	1997	1999	2000	2002
Technology	BICMOS	BICMOS	BICMOS	BICMOS
Number of transistors	7.5 million	9.5 million	42 million	220 million
Cache size	512K	512K	512K	3MB
Physical memory	64GB	64GB	64GB	64GB
Virtual memory	64TB	64TB	64TB	64TB
Internal data bus	32	32	32	64
External data bus	64	64	64	64
Address bus	36	36	36	64
Data types	8/16/32	8/16/32	8/16/32	8/16/32/64

REVIEW

1. Features of the 8086 that were improvements over the 8080/8085
2. Differences between the 8086 and 8088 microprocessors
3. Differences between the 80386 and the 80386SX
4. Additional features introduced with the 80286 that were not present in the 8086
5. Additional features introduced with the 80486 that were not present in the 80386
6. Additional features introduced with the Pentium that were not present in the 80486

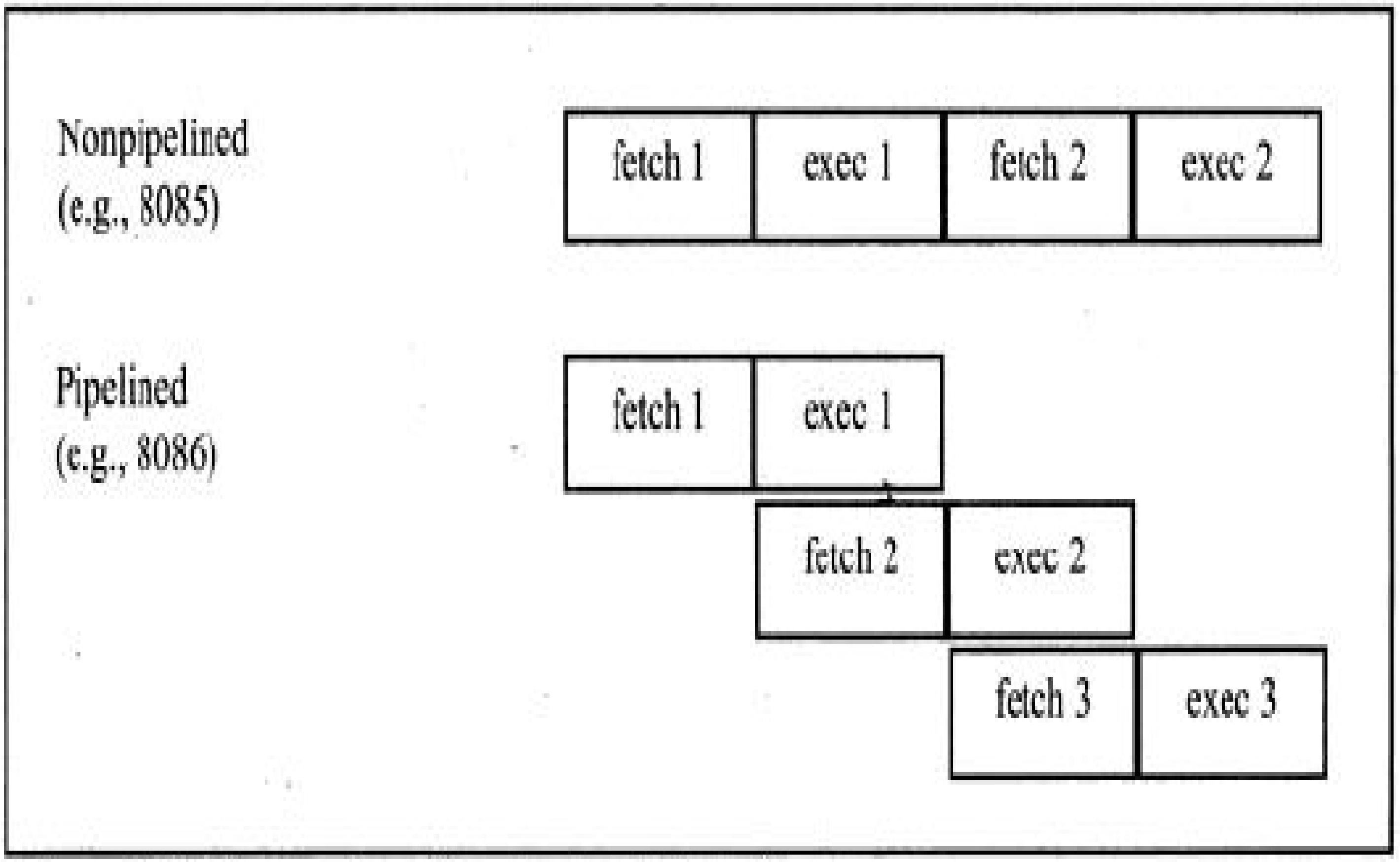
INSIDE THE 8088/86



Pipelining

To process the information faster, the CPU can:

- » Increase the working frequency –
  - » But, it is technology dependent
- » Change the internal architecture of the CPU –
  - » Eg: In 8085, the CPU had to fetch an instruction from memory, then execute it and then fetch again, execute it, and so on; i.e., 8085 CPU could either fetch or execute at a given time

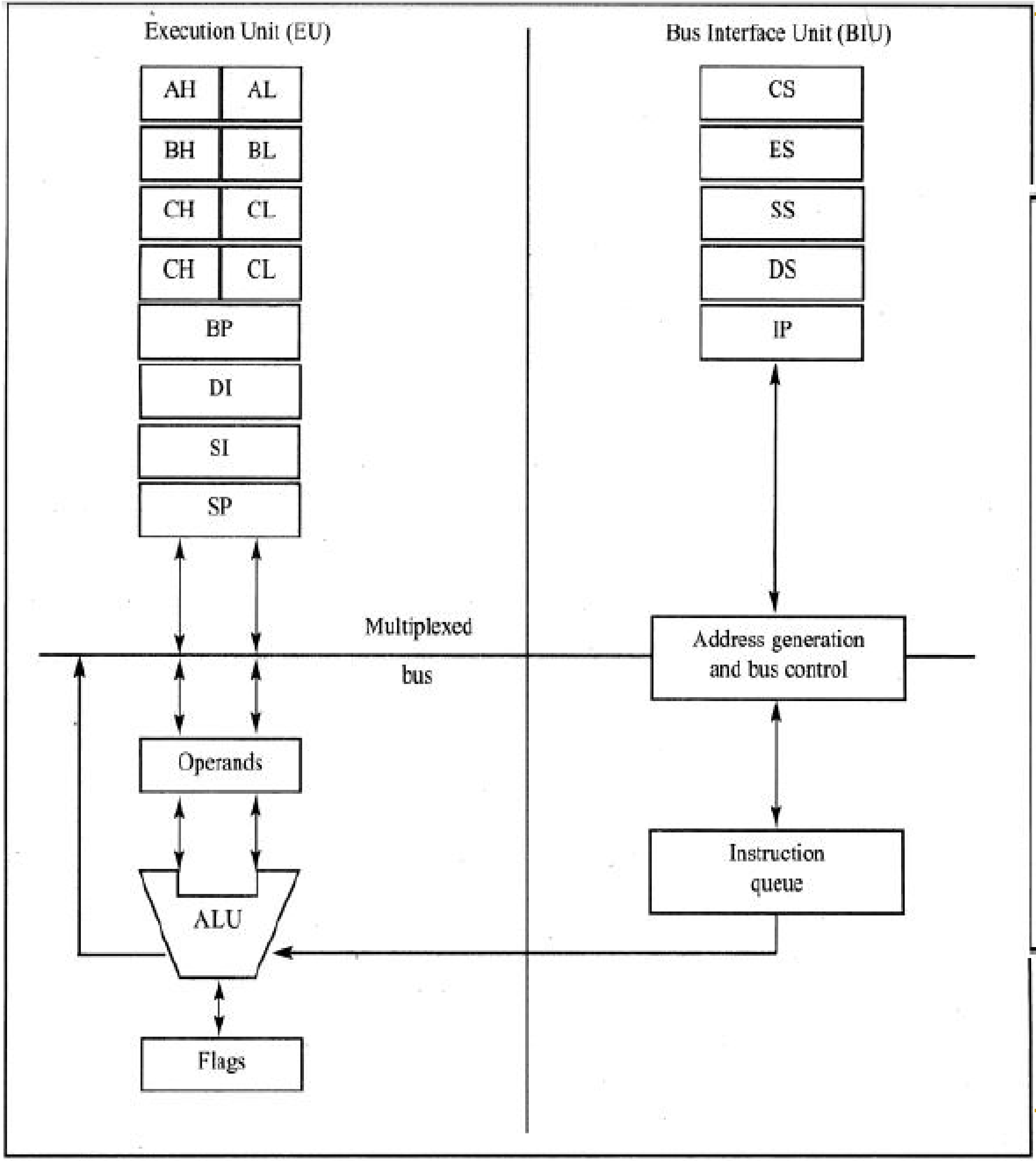


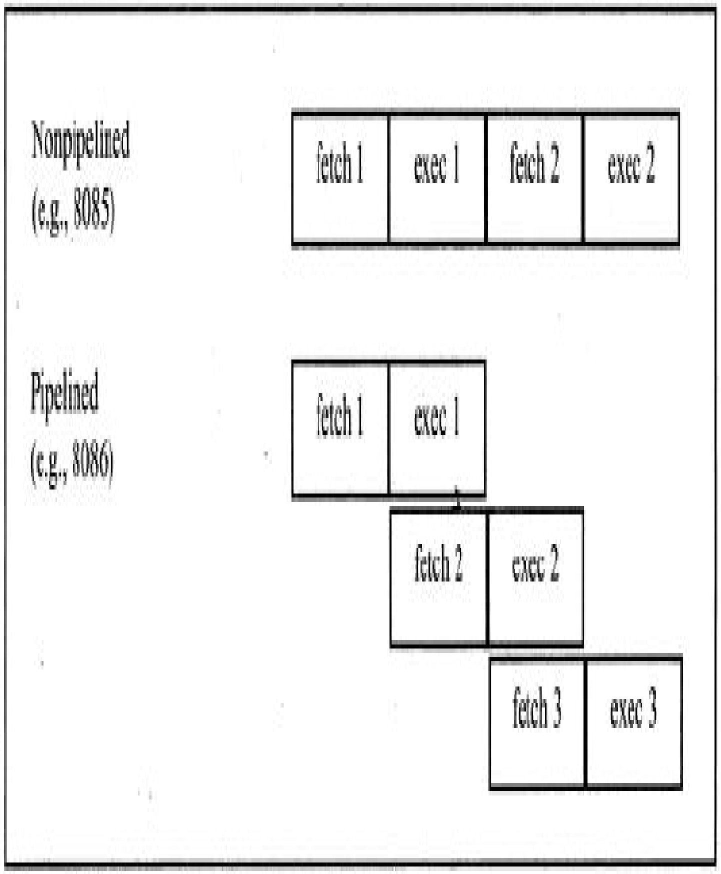
Nonpipelined
(e.g., 8085)

fetch 1 | exec 1 | fetch 2 | exec 2

Pipelined
(e.g., 8086)

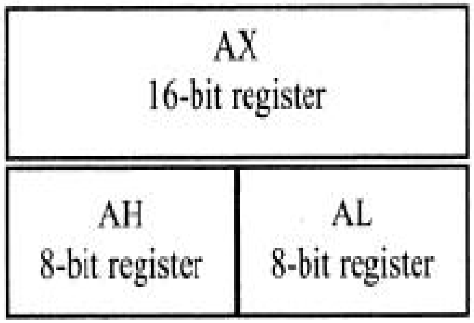
fetch 1 | exec 1
fetch 2 | exec 2
fetch 3 | exec 3


Execution Unit (EU)
Bus Interface Unit (BIU)
AH
AL
BH
BL
CH
CL
CH
CL
BP
DI
SI
SP
CS
ES
SS
DS
IP
Multiplexed
bus
Address generation and bus control
Operands
Instruction queue
ALU
Flags


Nonpipelined (e.g., 8085)
fetch 1
exec 1
fetch 2
exec 2
Pipelined (e.g., 8086)
fetch 1
exec 1
fetch 2
exec 2
fetch 3
exec 3

Registers

Category	**Bits**	**Register Names**
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, VH, CL, DH, DL
Pointer	16	SP (Stack Pointer) BP (Base Pointer)
Index	16	SI (Source Index) DI (Destination Index)
Segment	16	CS (Code Segment) DS (Data Segment) SS (Stack Segment) ES (Extra Segment)
Instruction	16	IP (Instruction Pointer)
Flag	16	FR (Flag Register)

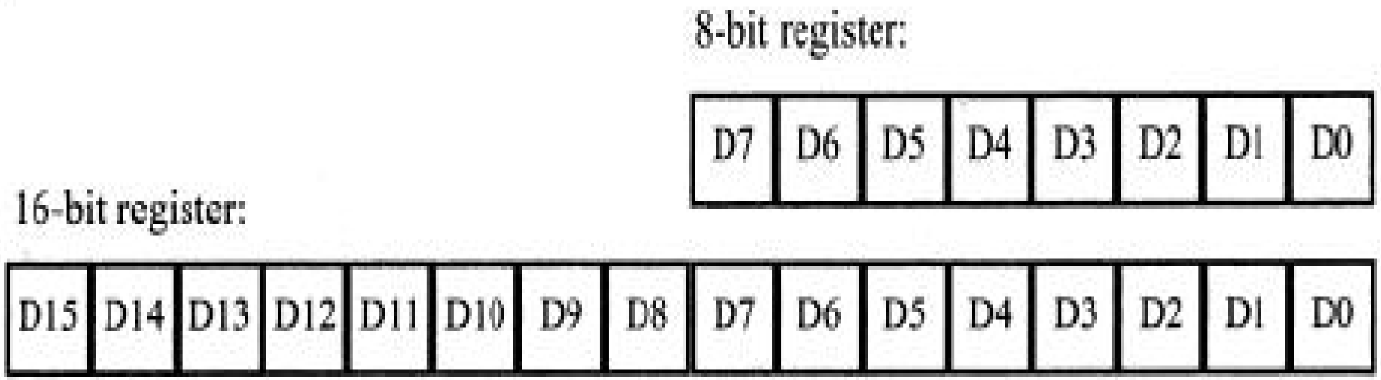


AX is used for the accumulator

BX as a base addressing register

CX as a counter in loop operations

DX to point to data in I/O operations



REVIEW

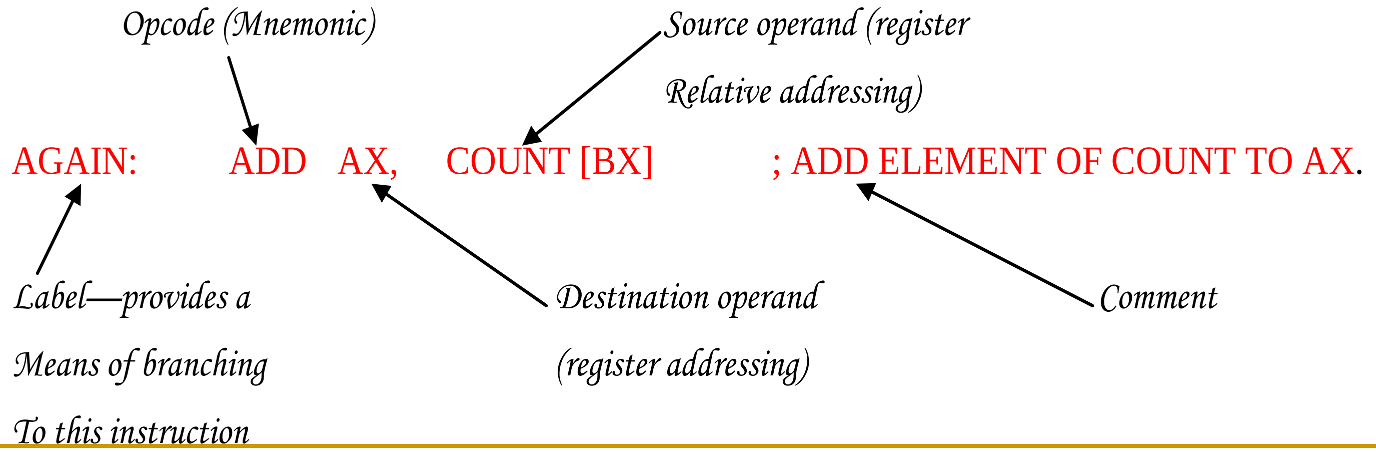
1. Explain the functions of the EU and BIU
2. What is pipelining? How does it make the CPU execute faster?

INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

- Machine Language – quite tedious and slow for humans to deal with 0s and 1s
- Assembly Language – mnemonic for the machine code instruction – programming is faster and less prone to errors
  - ALP must be translated into machine code (also called as object code) by a program called an ***assembler***
  - Assembly language is referred to as a ***low-level language*** – deals directly with the internal structure of the CPU
- High-level Language – programmer does not have to be concerned with the internal details of the CPU – ***compiler***

Assembly Language Programming

- An *Assembly language program (ALP)* consists of –
  - A series of lines of Assembly language instructions; which consists of –
    - a mnemonic – the commands to the CPU
    - (Optionally) operands – the data items being manipulated



Opcode (Mnemonic)

Source operand (register Relative addressing)

AGAIN: ADD AX, COUNT [BX] ; ADD ELEMENT OF COUNT TO AX.

Label—provides a Means of branching To this instruction

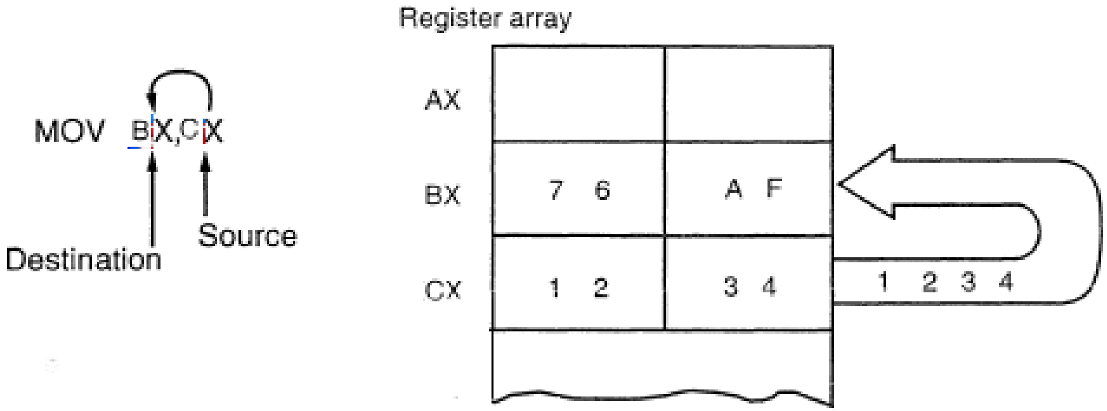
Destination operand (register addressing)

Comment

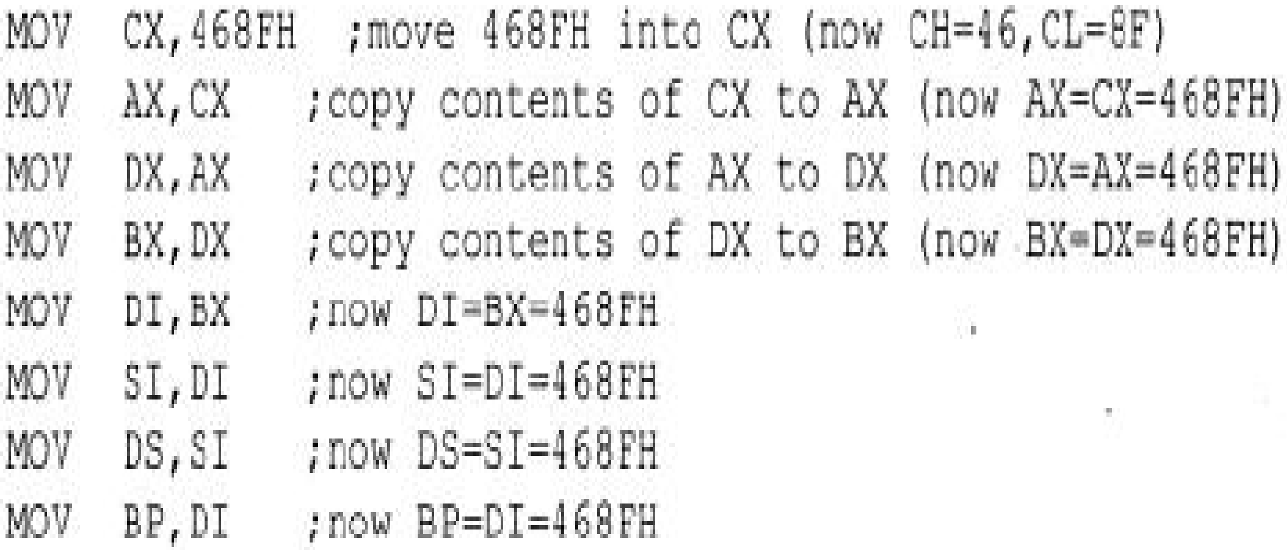
MOV Instruction:

» Copies data from one location to another

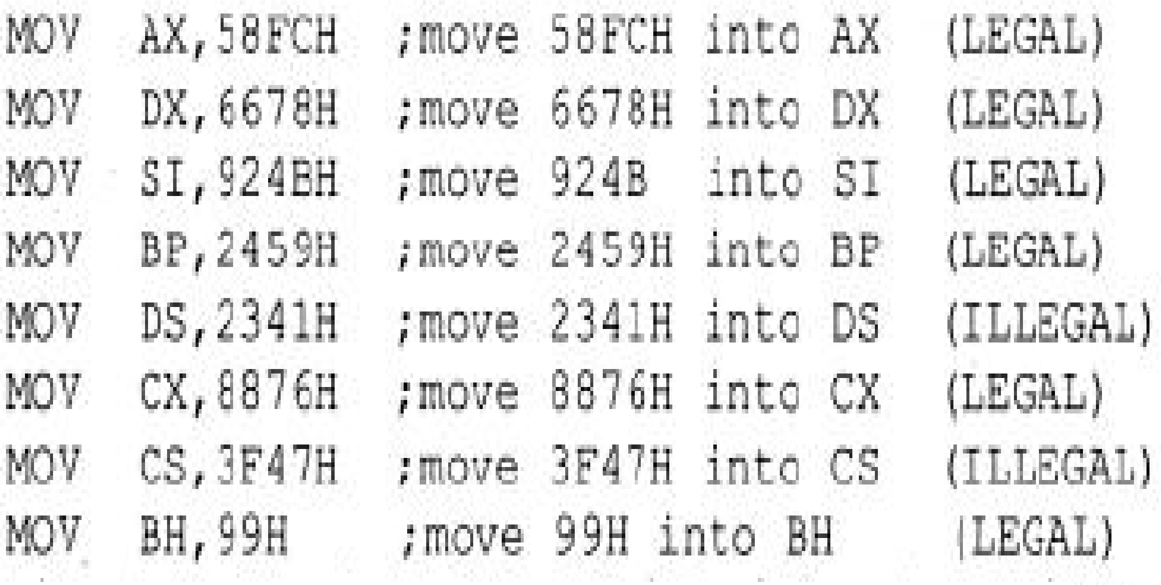
```
MOV  destination,source ;copy source operand to destination
```



```
MOV  CL,55H ;move 55H into register CL
MOV  DL,CL ;copy the contents of CL into DL (now DL=CL=55H)
MOV  AH,DL ;copy the contents of DL into AH (now AH=DL=55H)
MOV  AL,AH ;copy the contents of AH into AL (now AL=AH=55H)
MOV  BH,CL ;copy the contents of CL into BH (now BH=CL=55H)
MOV  CH,BH ;copy the contents of BH into CH (now CH=BH=55H)
```



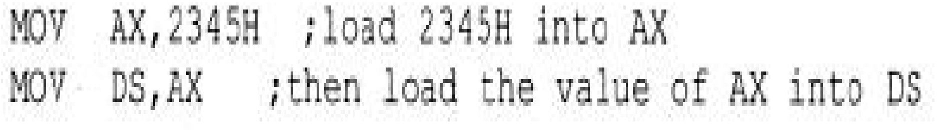
```
MOV   CX,468FH   ;move 468FH into CX (now CH=46,CL=8F)
MOV   AX,CX     ;copy contents of CX to AX (now AX=CX=468FH)
MOV   DX,AX     ;copy contents of AX to DX (now DX=AX=468FH)
MOV   BX,DX     ;copy contents of DX to BX (now BX=DX=468FH)
MOV   DI,BX     ;now DI=BX=468FH
MOV   SI,DI     ;now SI=DI=468FH
MOV   DS,SI     ;now DS=SI=468FH
MOV   BP,DI     ;now BP=DI=468FH
```



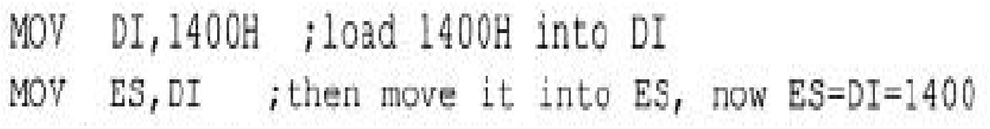
```
MOV   AX,58FCH   ;move 58FCH into AX   (LEGAL)
MOV   DX,6678H   ;move 6678H into DX   (LEGAL)
MOV   SI,924BH   ;move 924B  into SI   (LEGAL)
MOV   BP,2459H   ;move 2459H into BP   (LEGAL)
MOV   DS,2341H   ;move 2341H into DS   (ILLEGAL)
MOV   CX,8876H   ;move 8876H into CX   (LEGAL)
MOV   CS,3F47H   ;move 3F47H into CS   (ILLEGAL)
MOV   BH,99H     ;move 99H into BH     (LEGAL)
```

Note:

- Values cannot be loaded directly into any segment register (CS, DS, SS, and ES)



```
MOV  AX,2345H  ;load 2345H into AX
MOV  DS,AX     ;then load the value of AX into DS
```

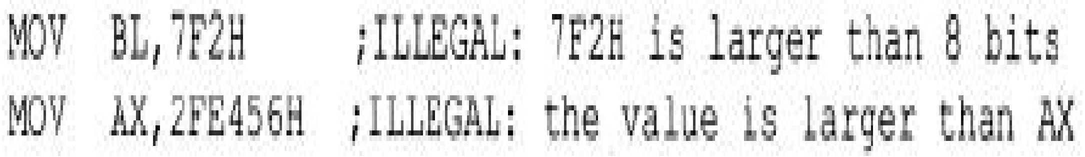


```
MOV  DI,1400H  ;load 1400H into DI
MOV  ES,DI     ;then move it into ES, now ES=DI=1400
```

- If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros

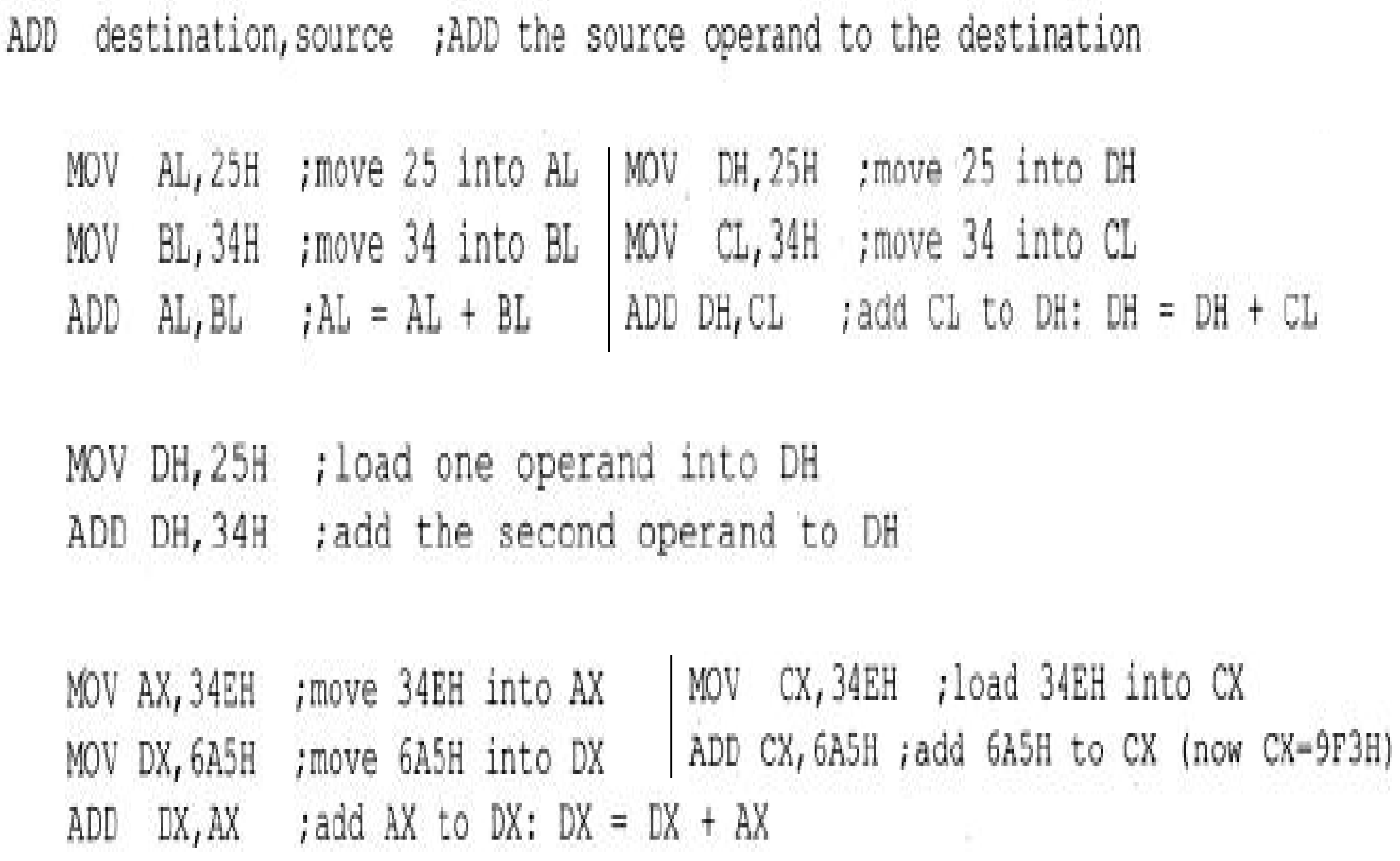
E.g.: MOV BX, 5 ; result will be BX = 0005, i.e., BH = 00 and BL = 05

- Moving a value that is too large into a register will cause an error



```
MOV  BL,7F2H     ;ILLEGAL: 7F2H is larger than 8 bits
MOV  AX,2FE456H  ;ILLEGAL: the value is larger than AX
```

ADD Instruction



```
ADD  destination,source  ;ADD the source operand to the destination

   MOV  AL,25H  ;move 25 into AL | MOV  DH,25H  ;move 25 into DH
   MOV  BL,34H  ;move 34 into BL | MOV  CL,34H  ;move 34 into CL
   ADD  AL,BL   ;AL = AL + BL    | ADD DH,CL    ;add CL to DH: DH = DH + CL

   MOV DH,25H  ;load one operand into DH
   ADD DH,34H  ;add the second operand to DH

   MOV AX,34EH  ;move 34EH into AX  | MOV  CX,34EH  ;load 34EH into CX
   MOV DX,6A5H  ;move 6A5H into DX  | ADD CX,6A5H ;add 6A5H to CX (now CX=9F3H)
   ADD  DX,AX   ;add AX to DX: DX = DX + AX
```

REVIEW

1. Which of the following instructions can not be coded in 8086 Assembly language? Give reason

(a) MOV AX, 27H (b) MOV AL, 97FH (c) MOV DS, 9BF2H

(d) MOV CX, 397H (e) MOV Si, 9516H (f) MOV CS, 3490

(g) MOV DS, BX (h) MOV BX, CS (i) MOV CH, AX

(j) MOV CS, BH (k) MOV AX, DL (l) MOV AX, 23FB9H

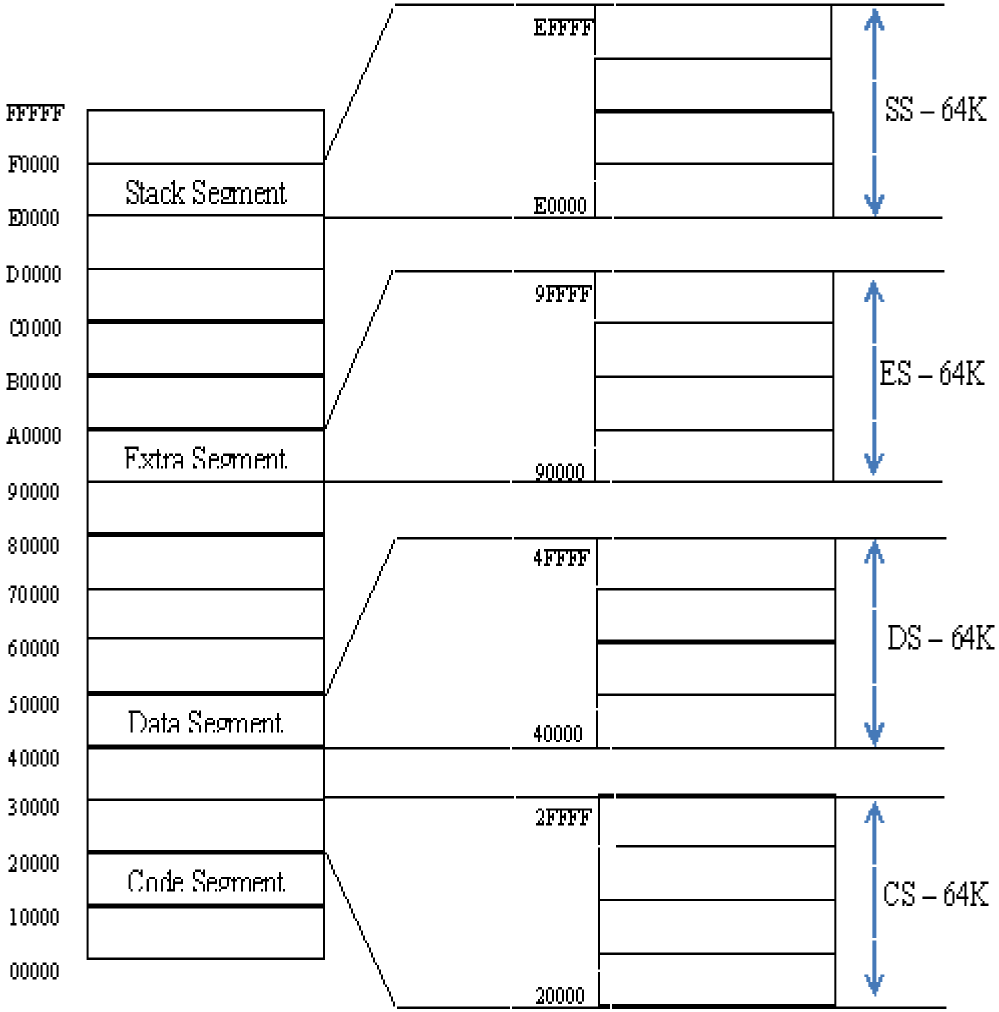
INTRODUCTION TO PROGRAM SEGMENTS

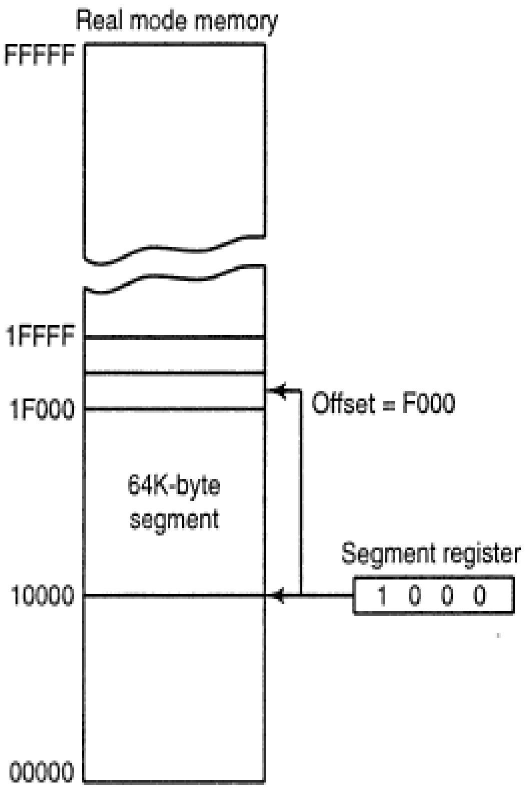
- A ***segment*** is –
  - an area of memory
  - includes up to 64K bytes
  - begins on an address evenly divisible by 16 (such an address ends in 0H)
- In 8085, there was only 64K byte (2^{16} = 64K bytes) of memory for all code, data, and stack information
- In the 8088/86 (addressable range of 1M bytes (2^{20} = 1MB) of memory) there can be up to 64K bytes of memory assigned to each category

Logical Address & Physical Address

- 3 types of addresses in 8086:

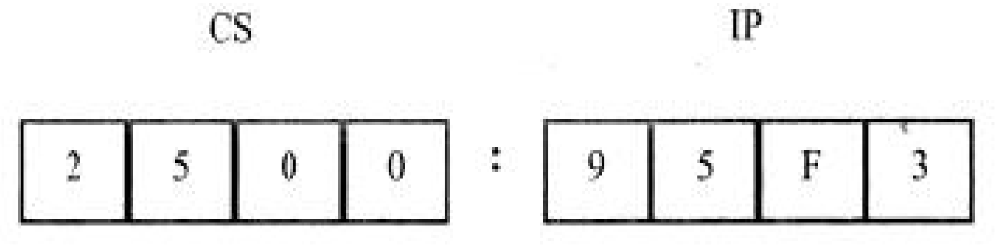
1. The physical address – is the 20-bit address that is actually put on the address pins of the 8086 microprocessor and decoded by memory interfacing circuitry (00000H – FFFFFH)
2. The offset address – is a location within a 64K byte segment range (0000H – FFFFH)
3. The logical address – consists of a segment value and an offset address.


FFFFF
F0000
E0000
D0000
C0000
B0000
A0000
90000
80000
70000
60000
50000
40000
30000
20000
10000
00000
Stack Segment
Extra Segment
Data Segment
Code Segment
EFFFF
E0000
SS – 64K
9FFFF
90000
ES – 64K
4FFFF
40000
DS – 64K
2FFFF
20000
CS – 64K


Real mode memory
FFFFF
1FFFF
1F000
Offset = F000
64K-byte segment
Segment register
1 0 0 0
10000
00000

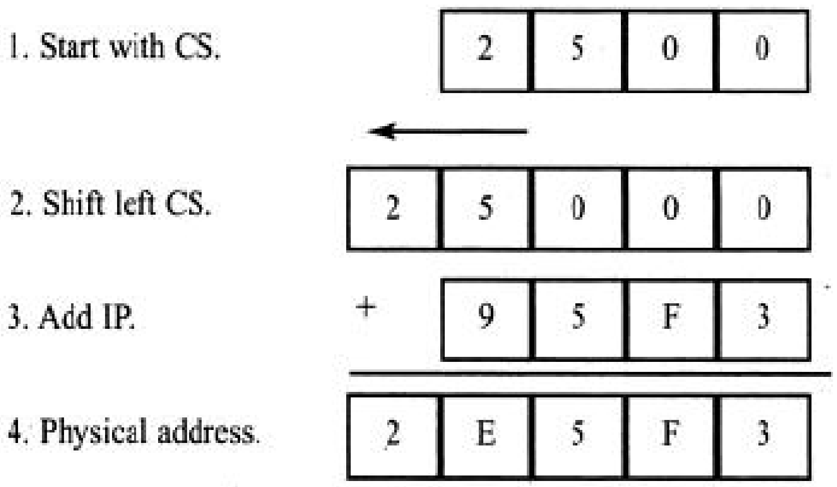
Code Segment

- 8086 fetches the instruction from the code segment
- The logical address of an instruction always consists of a CS and an IP



- The physical address of the instruction is generated by –
- Shifting the CS left by one hex digit and then adding it to the IP (offset)

- The offset address is contained in IP; let it be 95F3H.
- The logical address is CS: IP, or 2500: 95F3H.
- Then the physical address will be 25000 + 95F3 = 2E5F3H.



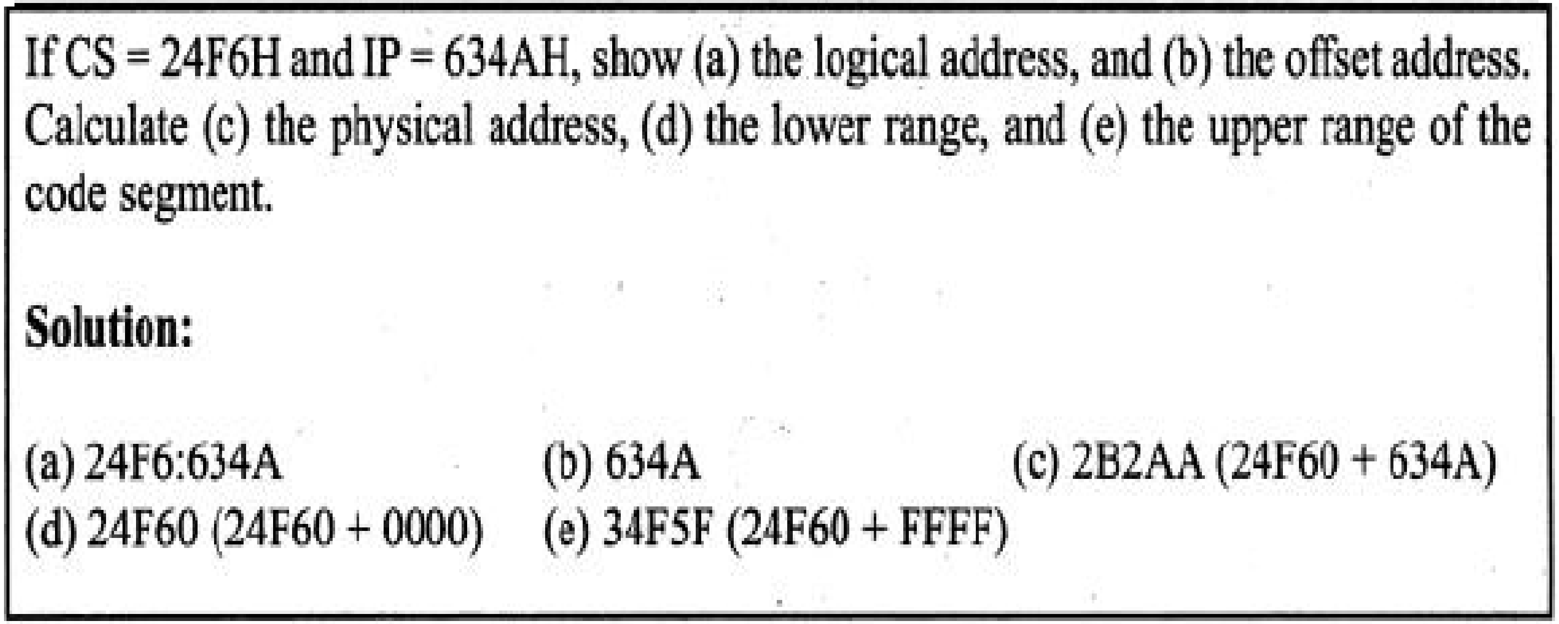
1. Start with CS. 2 5 0 0

2. Shift left CS. 2 5 0 0 0

3. Add IP. + 9 5 F 3

4. Physical address. 2 E 5 F 3

- The lowest memory location of the code segment will be 25000H (25000+0000)
- The highest memory location will be 34FFFH (25000+FFFF)

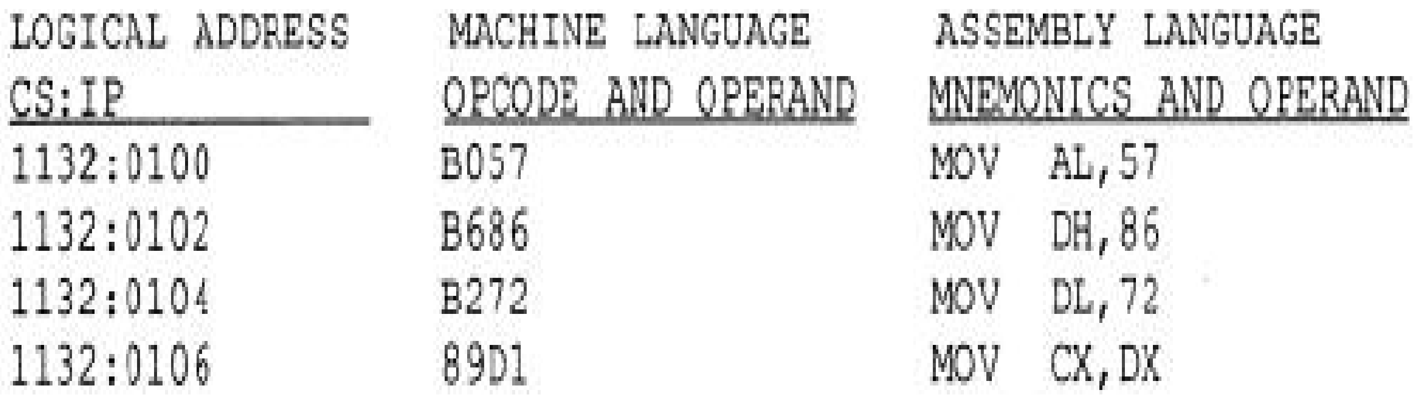


If CS = 24F6H and IP = 634AH, show (a) the logical address, and (b) the offset address. Calculate (c) the physical address, (d) the lower range, and (e) the upper range of the code segment.

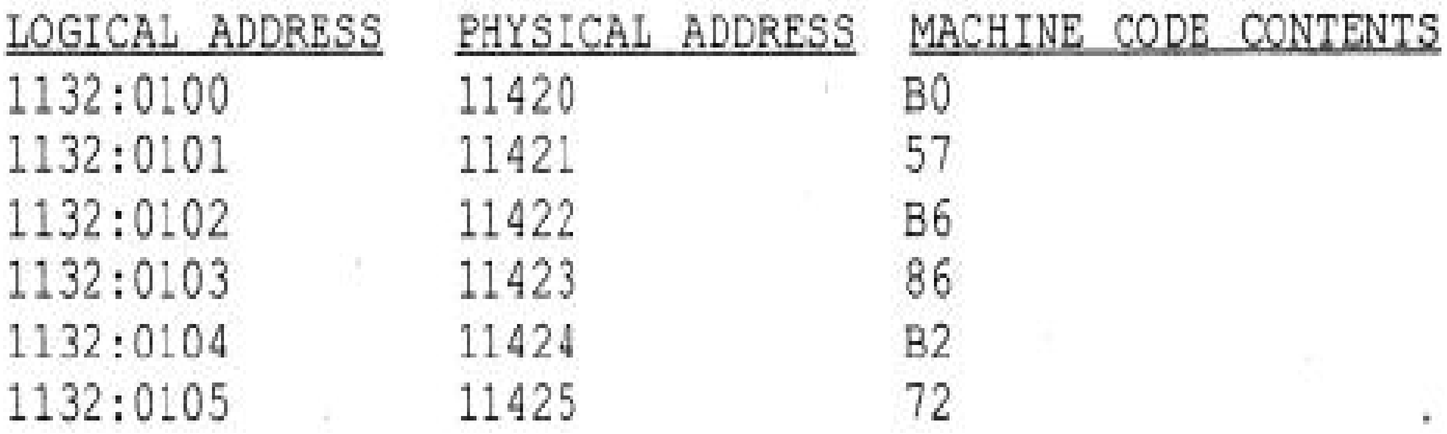
Solution:

(a) 24F6:634A (b) 634A (c) 2B2AA (24F60 + 634A)
(d) 24F60 (24F60 + 0000) (e) 34F5F (24F60 + FFFF)

Logical Address vs Physical Address



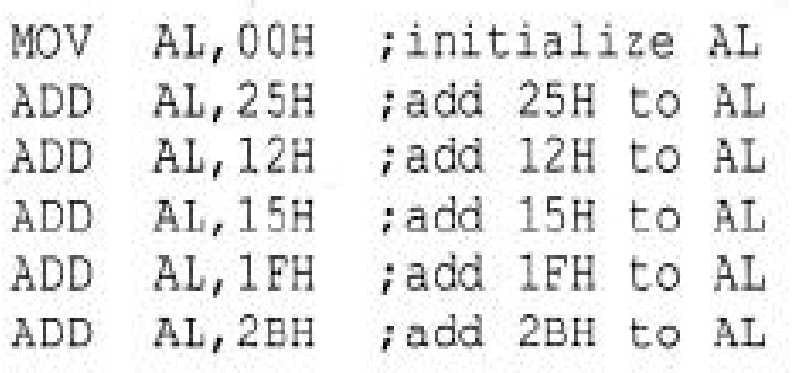
LOGICAL ADDRESS	MACHINE LANGUAGE	ASSEMBLY LANGUAGE
CS:IP	OPCODE AND OPERAND	MNEMONICS AND OPERAND
1132:0100	B057	MOV AL,57
1132:0102	B686	MOV DH,86
1132:0104	B272	MOV DL,72
1132:0106	89D1	MOV CX,DX



LOGICAL ADDRESS	PHYSICAL ADDRESS	MACHINE CODE CONTENTS
1132:0100	11420	B0
1132:0101	11421	57
1132:0102	11422	B6
1132:0103	11423	86
1132:0104	11424	B2
1132:0105	11425	72

Data Segment

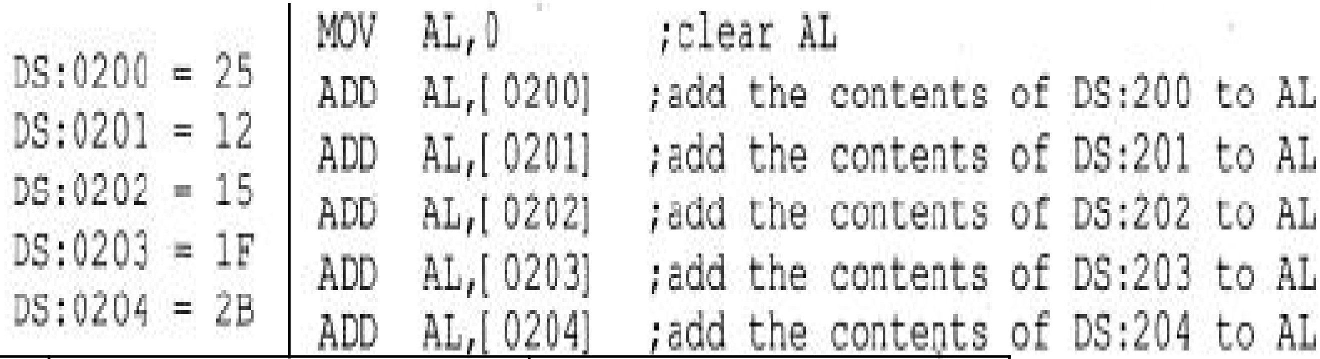
- One way to add 25H, 12H, 15H, IFH, and 2BH is –



```
MOV   AL,00H   ;initialize AL
ADD   AL,25H   ;add 25H to AL
ADD   AL,12H   ;add 12H to AL
ADD   AL,15H   ;add 15H to AL
ADD   AL,1FH   ;add 1FH to AL
ADD   AL,2BH   ;add 2BH to AL
```

- But, here, data and code are mixed together.
- Hence, if the data changes, the code must be searched for every place the data is included, and the data retyped.

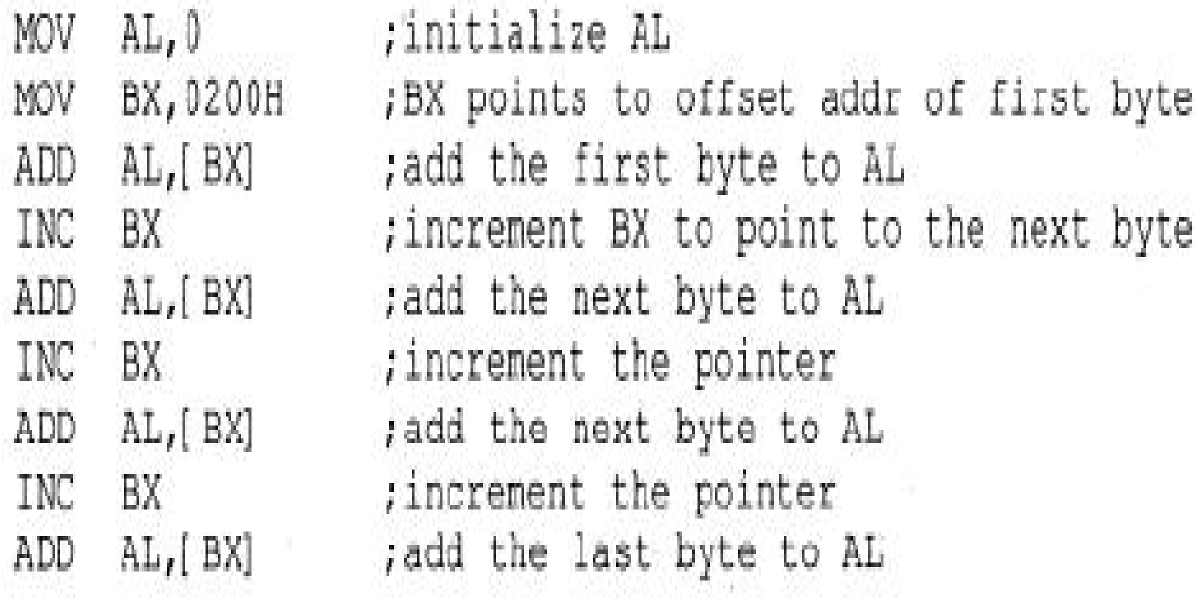
- To overcome the problem; set aside an area of memory, strictly for data – data segment



```
               MOV  AL,0         ;clear AL
DS:0200 = 25   ADD  AL,[0200]    ;add the contents of DS:200 to AL
DS:0201 = 12   ADD  AL,[0201]    ;add the contents of DS:201 to AL
DS:0202 = 15   ADD  AL,[0202]    ;add the contents of DS:202 to AL
DS:0203 = 1F   ADD  AL,[0203]    ;add the contents of DS:203 to AL
DS:0204 = 2B   ADD  AL,[0204]    ;add the contents of DS:204 to AL
```

Segment	**Offset**	**Special Purpose**
CS	IP	Instruction address
DS	SI, DI, BX, an 8- or 16-bit number	Data address
SS	SP or BP	Stack address
ES	SI, DI, BX for string instructions	String destination address

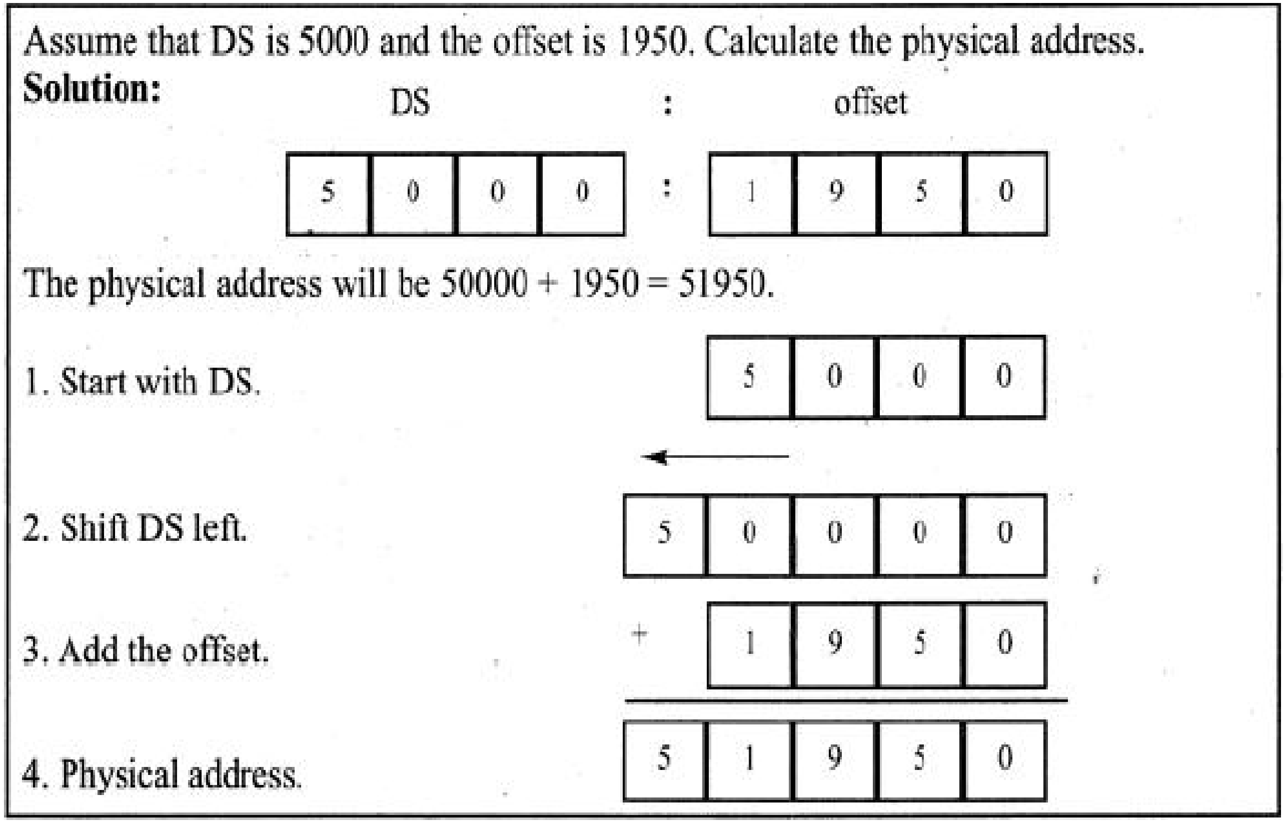
- The term *pointer* is often used for a register holding an offset address. In the following example, BX is used as a pointer



```
MOV  AL,0       ;initialize AL
MOV  BX,0200H   ;BX points to offset addr of first byte
ADD  AL,[BX]    ;add the first byte to AL
INC  BX         ;increment BX to point to the next byte
ADD  AL,[BX]    ;add the next byte to AL
INC  BX         ;increment the pointer
ADD  AL,[BX]    ;add the next byte to AL
INC  BX         ;increment the pointer
ADD  AL,[BX]    ;add the last byte to AL
```

- The INC instruction adds 1 to (increments) its operand.
- "INC BX" achieves the same result as "ADD BX, 1"

Logical Address & Physical Address in DS



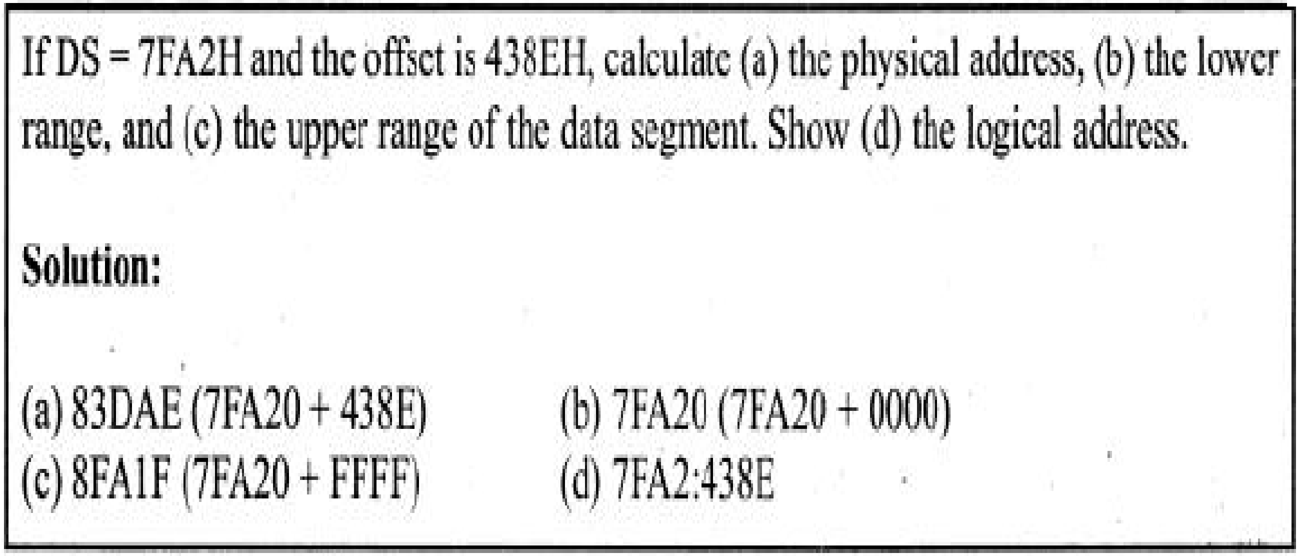
Assume that DS is 5000 and the offset is 1950. Calculate the physical address.

Solution: DS : offset

5 0 0 0 : 1 9 5 0

The physical address will be 50000 + 1950 = 51950.

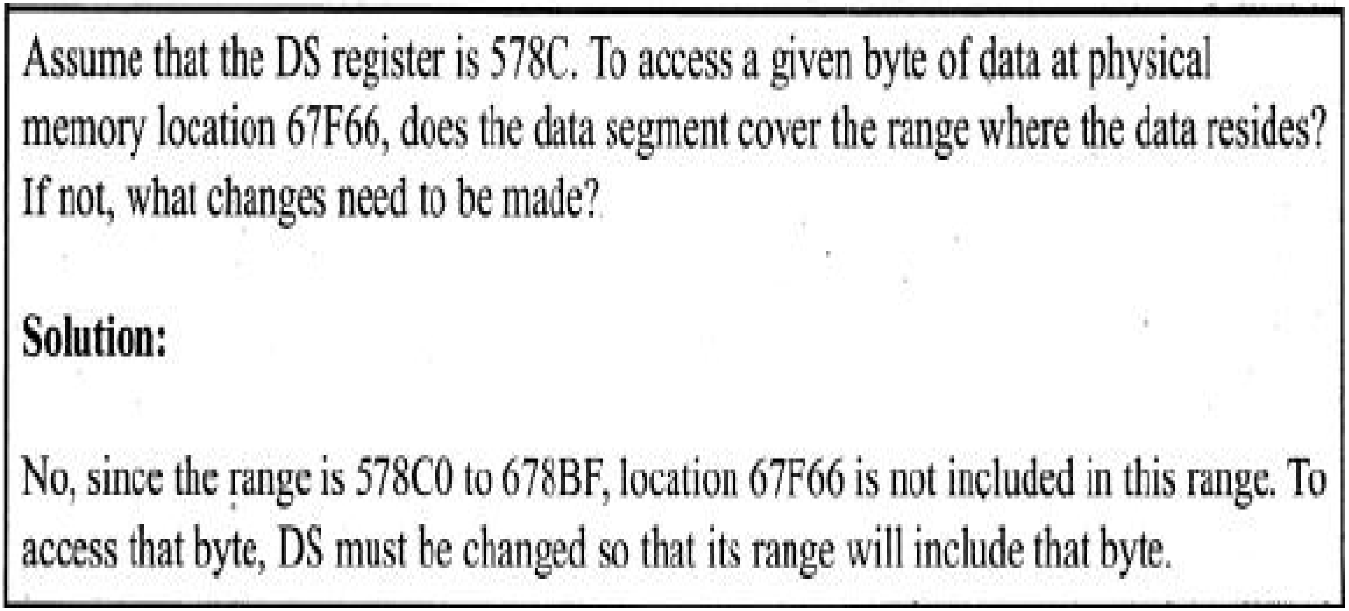
1. Start with DS. 5 0 0 0
2. Shift DS left. 5 0 0 0 0
3. Add the offset. + 1 9 5 0
4. Physical address. 5 1 9 5 0



If DS = 7FA2H and the offset is 438EH, calculate (a) the physical address, (b) the lower range, and (c) the upper range of the data segment. Show (d) the logical address.

Solution:

(a) 83DAE (7FA20 + 438E) (b) 7FA20 (7FA20 + 0000)
(c) 8FA1F (7FA20 + FFFF) (d) 7FA2:438E



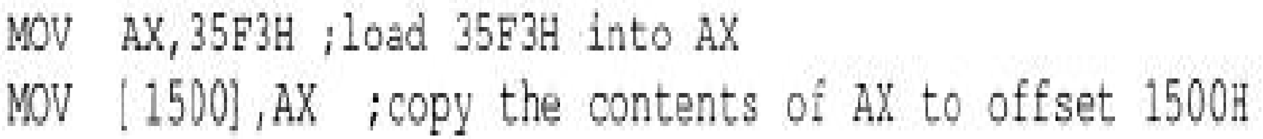
Assume that the DS register is 578C. To access a given byte of data at physical memory location 67F66, does the data segment cover the range where the data resides? If not, what changes need to be made?

Solution:

No, since the range is 578C0 to 678BF, location 67F66 is not included in this range. To access that byte, DS must be changed so that its range will include that byte.

Little Endian Conversion

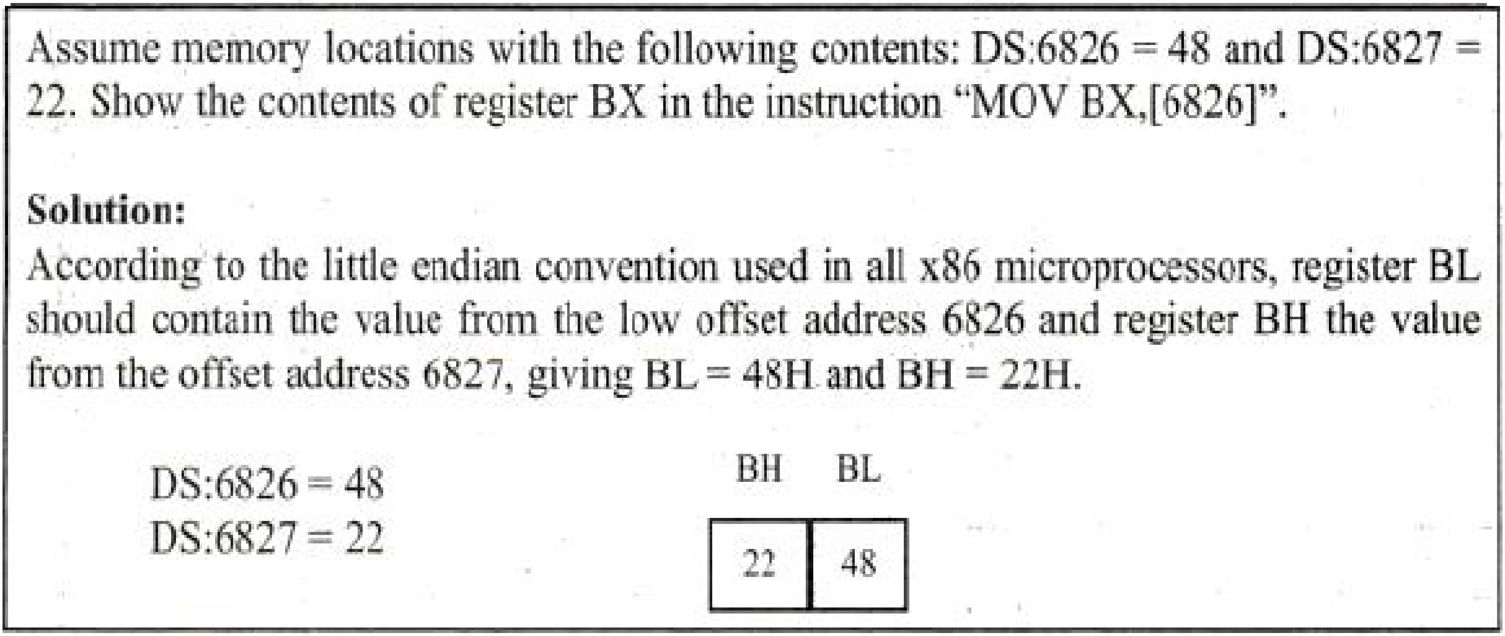
- In x86, the 16-bit data can be used as follows –



```
MOV  AX,35F3H ;load 35F3H into AX
MOV  [1500],AX  ;copy the contents of AX to offset 1500H
```

- The low byte goes to the low memory location and the high byte goes to the high memory location
- Hence, memory location DS: 1500 contains F3H and memory location DS: 1501 contains 35H
- (DS: 1500 = F3 and DS: 1501 = 35). This is called little endian conversion

- In the ***big endian method***,
  - the high byte goes to the low address
- In the ***little endian method***,
  - the high byte goes to the high address and the low byte goes to the low address.
- All Intel microprocessors use the little endian conversion



Assume memory locations with the following contents: DS:6826 = 48 and DS:6827 = 22. Show the contents of register BX in the instruction "MOV BX,[6826]".

Solution:

According to the little endian convention used in all x86 microprocessors, register BL should contain the value from the low offset address 6826 and register BH the value from the offset address 6827, giving BL = 48H and BH = 22H.

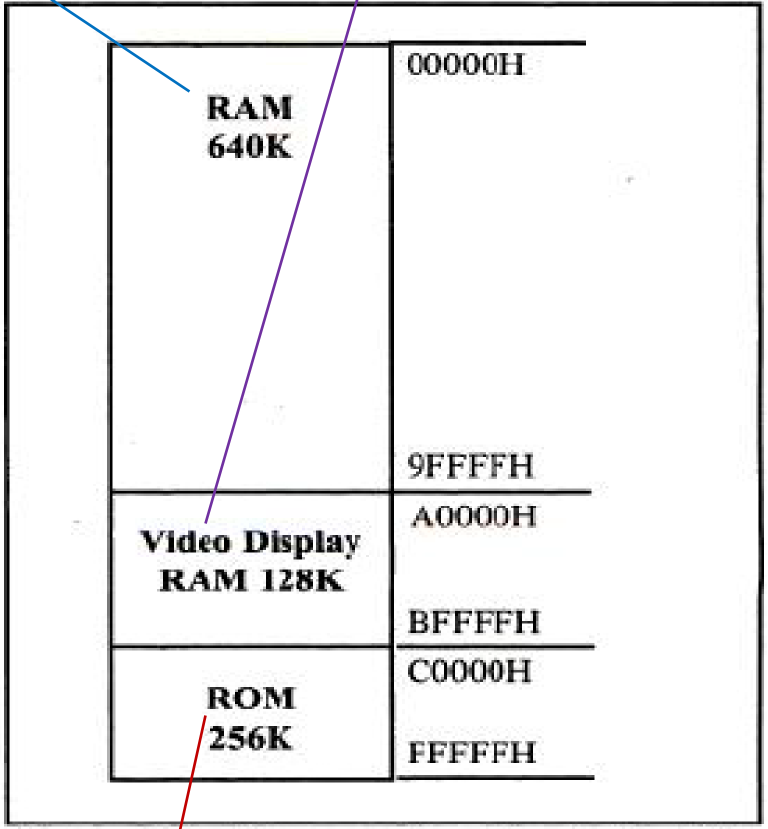
DS:6826 = 48
DS:6827 = 22

BH	BL
22	48

Memory Map of IBM PC

- The 20-bit address of 8088/86 allows a total of 1M bytes (00000H – FFFFFH)
- *Memory map* is the process of allocating the 1M bytes of memory space to various sections of the PC
- Out of 1M byte –
  - 640K bytes from the address 00000H – 9FFFFH, for RAM
  - The 128KB from A0000H – BFFFFH, for video memory
  - The remaining 256KB from C0000H – FFFFFH for ROM

- Memory management is the function of OS
- OS allocates the RAM (first) for its own use and for applications
  - The amount of memory used by Windows varies among its various versions
  - The memory needs of the application packages are different
  - The program would not be portable to another PC
- The amount used and location vary depending on the video board installed on the PC



- 64 KB are used by the BIOS
- Some space is used by various adapter cards
- Rest is free

Functions of BIOS ROM

- CPU can only execute programs that are stored in memory
- When the power is turned on, there must be some permanent (nonvolatile) memory to hold the programs, which tell the CPU what to do
- This collection of programs held by ROM is referred to as BIOS in the PC literature

- BIOS, *basic input-output system*, contains –
  - Programs to test RAM and other components connected to the CPU
  - Programs that allow Windows to communicate with peripheral devices such as the keyboard, video, printer, and disk.
- The functions of BIOS is to –
  - Test all the devices connected to the PC when the computer is turned on
  - Report any errors
- After testing and setting up the peripherals; BIOS will
  - Load Windows from disk into RAM and hand over control of the PC to Windows
- Windows always controls the PC once it is loaded

REVIEW

1. How large is a segment in 8086? Can the physical address 346E0H be the starting address for a segment? Justify
2. Name segment registers and their functions in 8086
3. If CS = 3499H and IP = 2500H, find: (a) the logical address (b) the physical address (c) the lower and the upper ranges of the code segment
4. If CS = 1296H and IP = 100H, find: (a) the logical address (b) the physical address (c) the lower and the upper ranges of the code segment

REVIEW

5. If DS = 3499H and IP = 3FB9H, find: (a) the logical address (b) the physical address (c) the lower and the upper ranges of the code segment

6. If CS = 1296H and IP = 7CC8H, find: (a) the logical address (b) the physical address (c) the lower and the upper ranges of the code segment

7. If SS = 2000H and SP = 4578H, find: (a) the logical address (b) the physical address (c) the lower and the upper ranges of the code segment

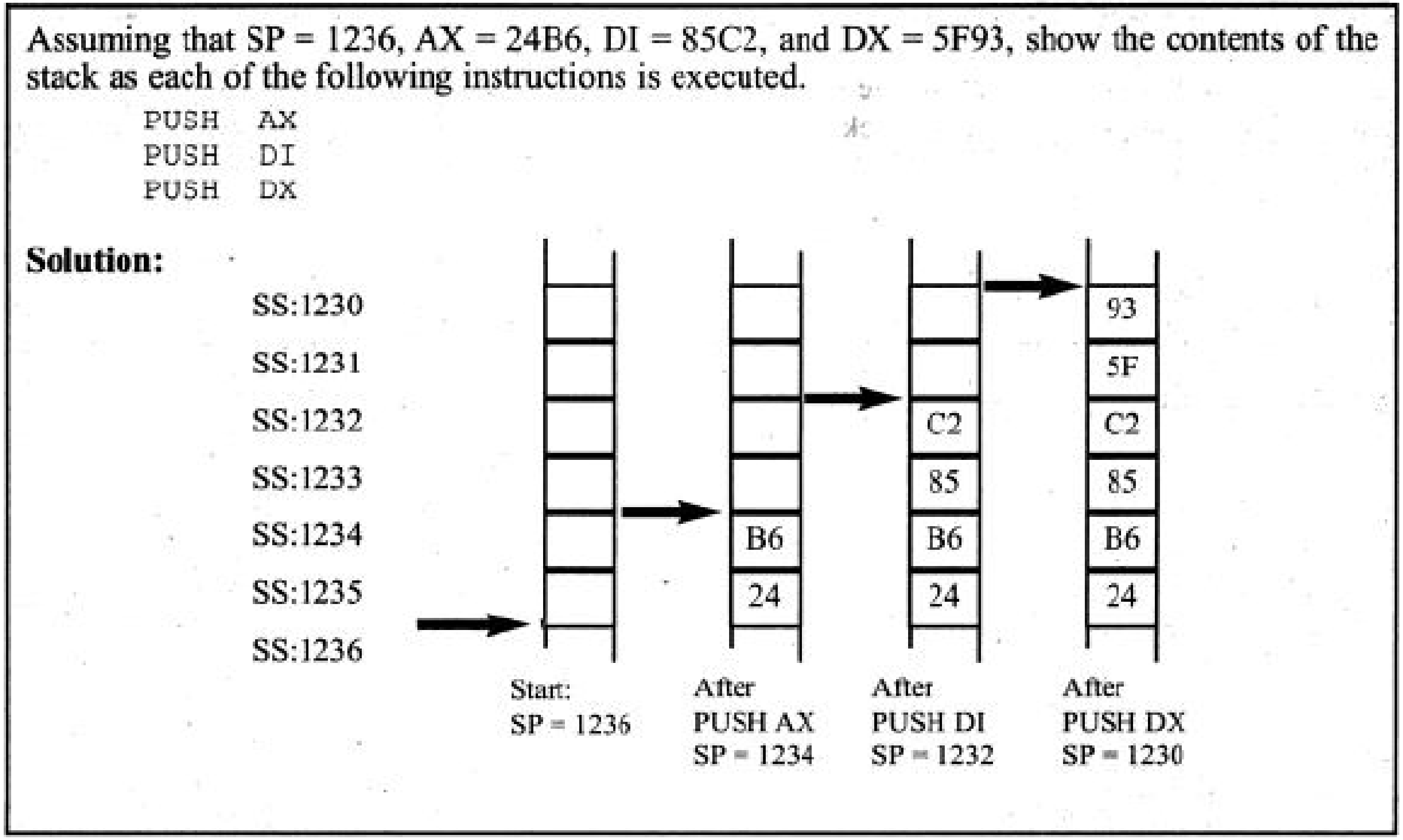
THE STACK

- The *stack* is a section of read/write memory (RAM) used by the CPU to store information temporarily
- The CPU needs this storage area since there are only a limited number of registers
- The disadvantage is its access time – since the stack is in RAM, it takes much longer to access compared to the access time of registers.
- Note that, the registers are inside the CPU and RAM is outside.

How the Stack are Accessed?

- The stack must be loaded, before accessing it
- SS & SP are registers used to access the stack
- Storing of a CPU register in the stack is a push
- Loading the contents of the stack into the CPU register is a pop
  - Push/Pop is associated with entire 16-bit register
- SP points at the current memory location used for the top of the stack
  - When data is pushed onto the stack SP is decremented
  - When data is popped off the stack, SP is incremented
- Stack is growing downward from upper addresses to lower addresses

Pushing onto the Stack



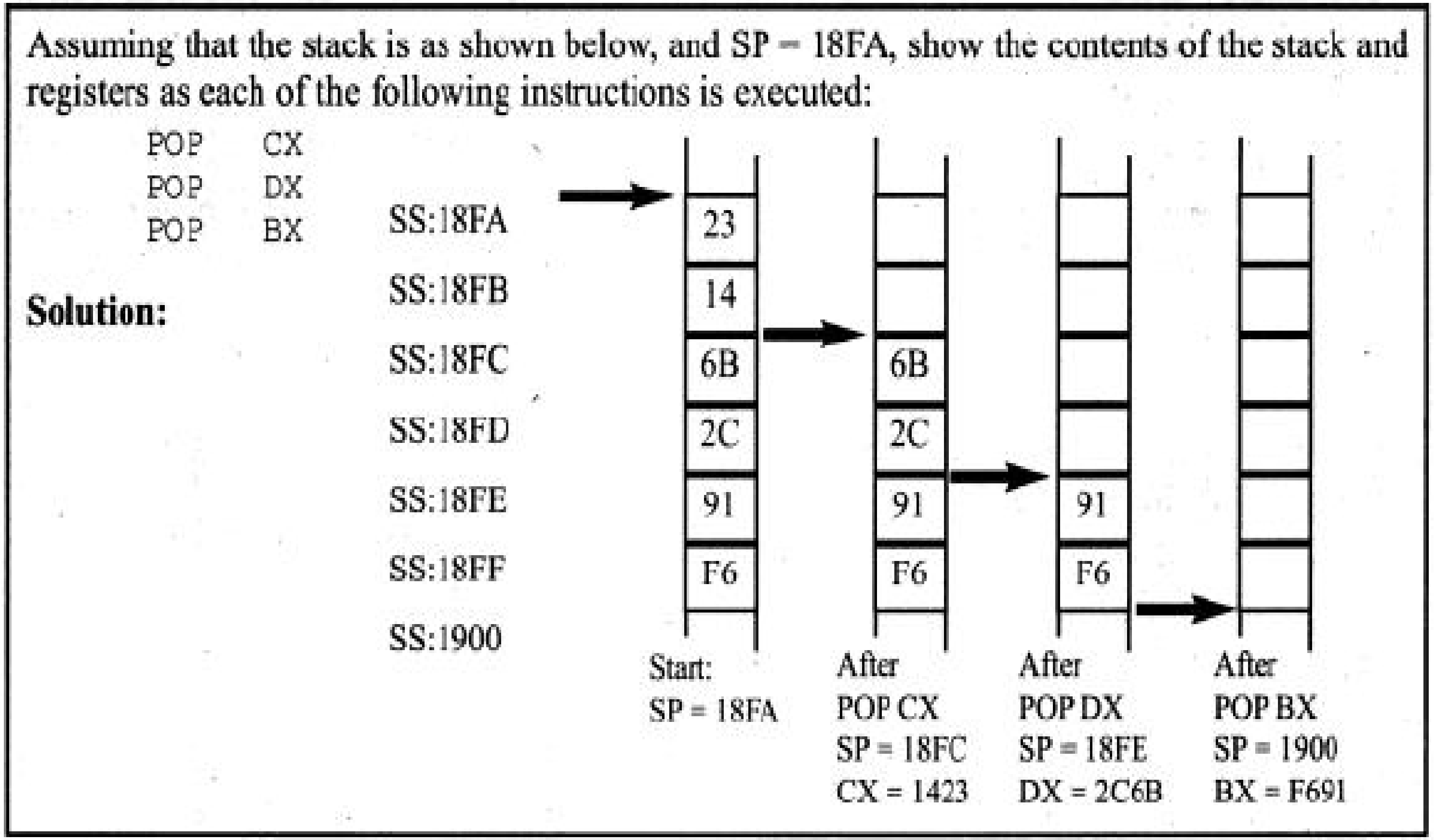
Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed.

```
PUSH  AX
PUSH  DI
PUSH  DX
```

Solution:

	Start: SP = 1236	After PUSH AX SP = 1234	After PUSH DI SP = 1232	After PUSH DX SP = 1230
SS:1230				93
SS:1231				5F
SS:1232			C2	C2
SS:1233			85	85
SS:1234		B6	B6	B6
SS:1235		24	24	24
SS:1236				

Popping the Stack



Assuming that the stack is as shown below, and SP = 18FA, show the contents of the stack and registers as each of the following instructions is executed:

```
POP    CX
POP    DX
POP    BX
```

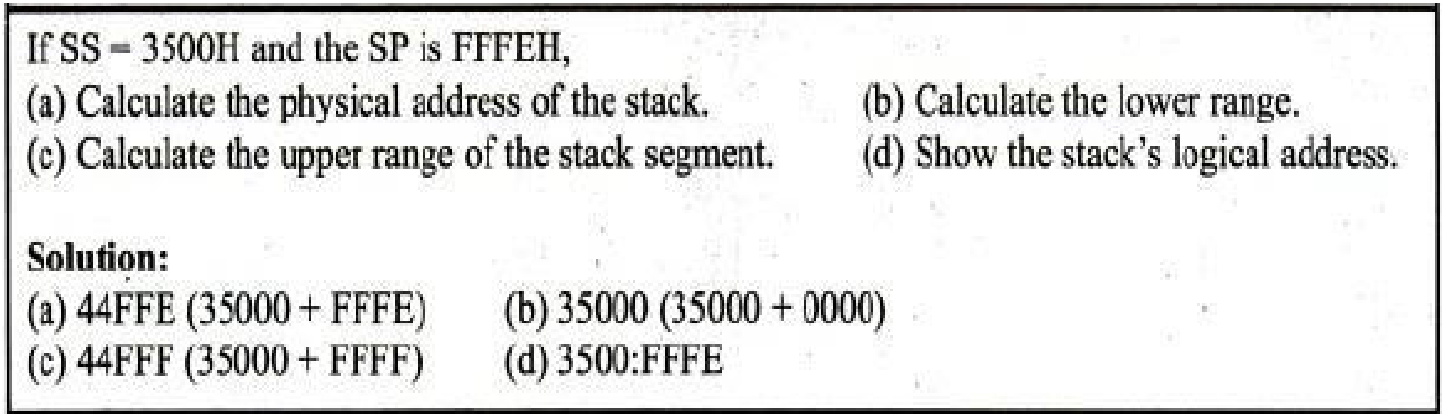
Solution:

	Start:
	SP = 18FA
		After
		POP CX
		SP = 18FC
		CX = 1423
			After
			POP DX
			SP = 18FE
			DX = 2C6B
				After
				POP BX
				SP = 1900
				BX = F691

SS:18FA	23			
SS:18FB	14			
SS:18FC	6B	6B		
SS:18FD	2C	2C		
SS:18FE	91	91	91	
SS:18FF	F6	F6	F6	
SS:1900				

Logical Address vs Physical Address for the Stack

- The physical location of the stack depends on
  - The value of the SS (stack segment) register
  - The SP (stack pointer).
- To compute the physical address for stack, shift left SS and then add offset SP register



If SS = 3500H and the SP is FFFEH,

(a) Calculate the physical address of the stack.

(b) Calculate the lower range.

(c) Calculate the upper range of the stack segment.

(d) Show the stack's logical address.

Solution:

(a) 44FFE (35000 + FFFE)

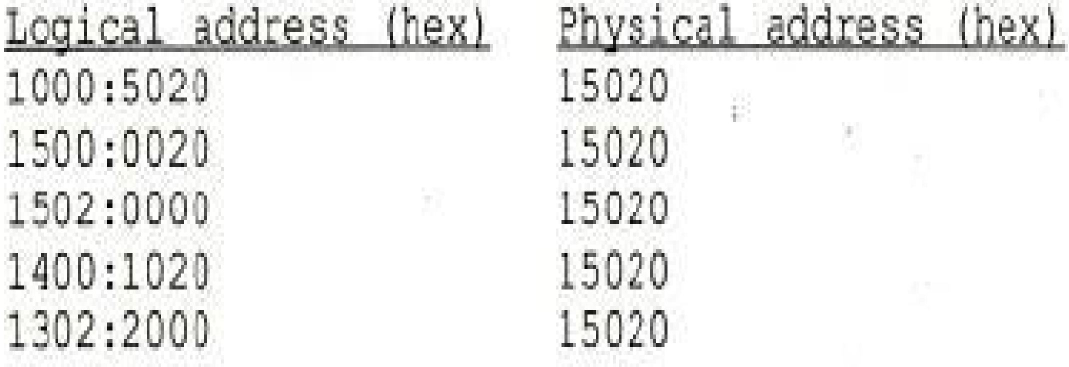
(b) 35000 (35000 + 0000)

(c) 44FFF (35000 + FFFF)

(d) 3500:FFFE

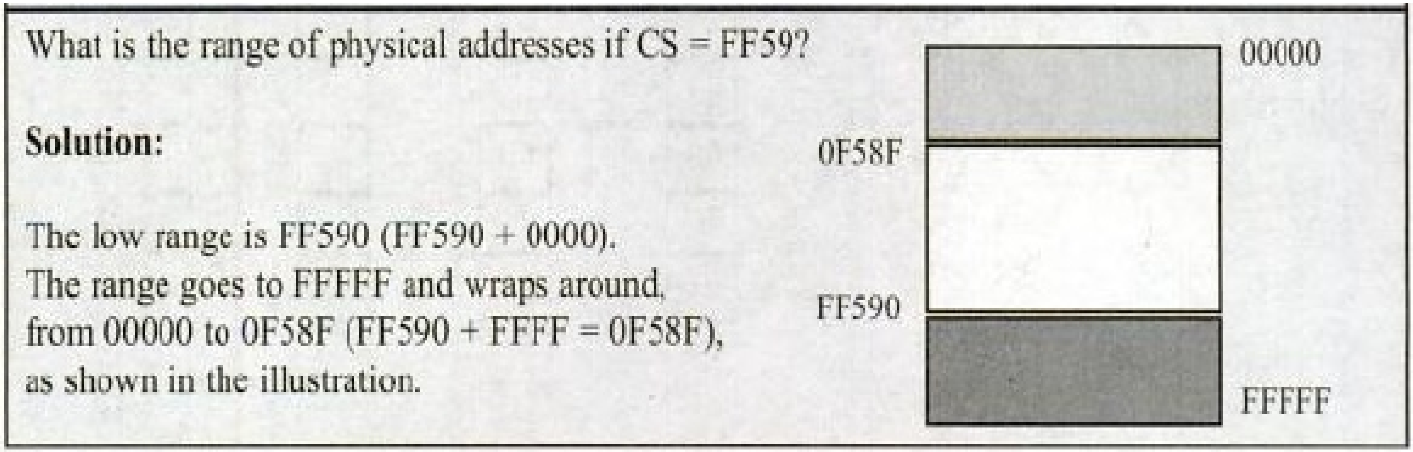
NOTE

1. Dynamic behavior of the segment and offset concept in the 8086 CPU – A single physical address may belong to many different logical addresses



Logical address (hex)	Physical address (hex)
1000:5020	15020
1500:0020	15020
1502:0000	15020
1400:1020	15020
1302:2000	15020

2. When adding the offset to the shifted segment register; if an address beyond the maximum allowed range (FFFFFH) is resulted, then wrap-around will occur



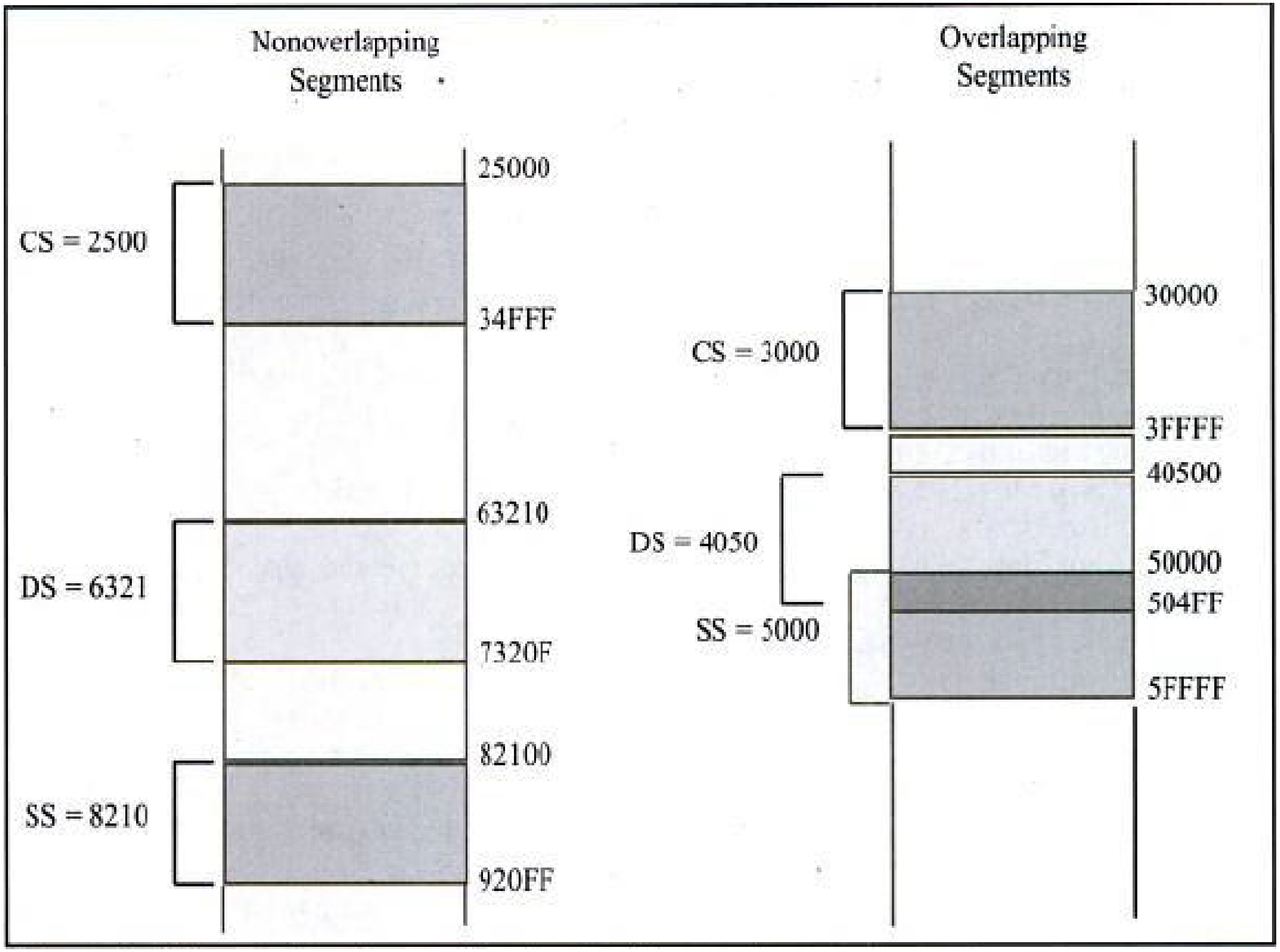
What is the range of physical addresses if CS = FF59?

Solution:

The low range is FF590 (FF590 + 0000).
The range goes to FFFFF and wraps around,
from 00000 to 0F58F (FF590 + FFFF = 0F58F),
as shown in the illustration.

00000
0F58F
FF590
FFFFF

3. Non-overlapping vs Overlapping Segments



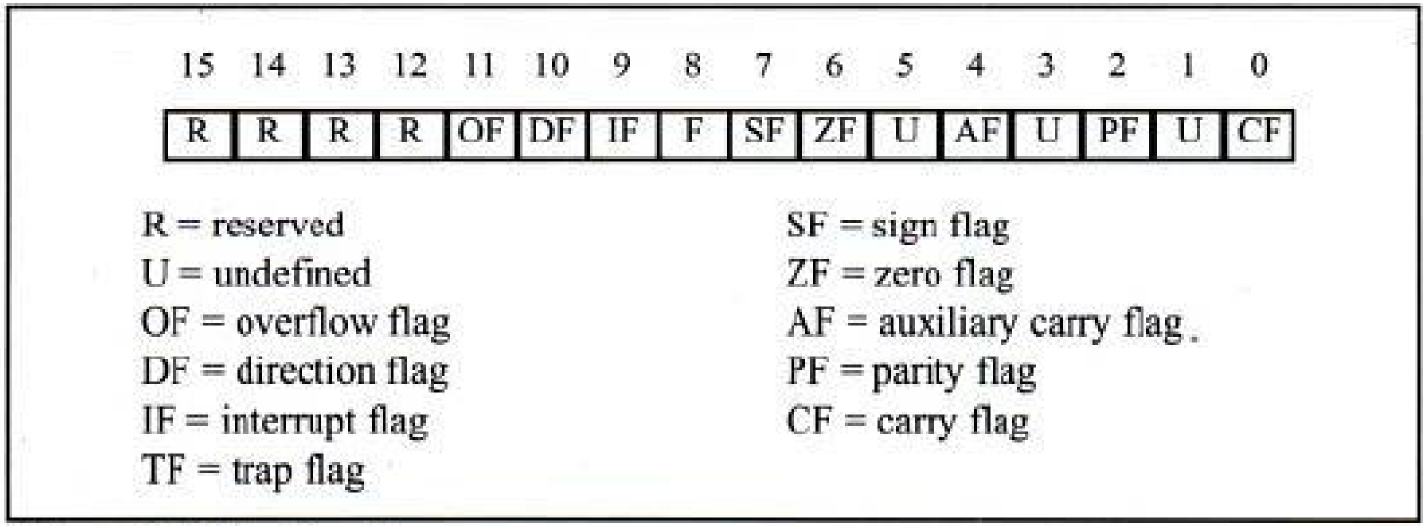
REVIEW

1. If SS = 2000H and SP = 4578H, find: (a) the logical address (b) the physical address (c) the lower and the upper ranges of the code segment

2. Assume that SP = FF2EH, AX = 3291H, BX = F43CH, and CX = 09. Find the contents of the stack and SP after the execution of each of following instructions: PUSH AX PUSH BX PUSH CX

3. Show the segment register(s) used in the following cases: (a) MOV SS: [BX], AX (b) MOV SS: [DI], BX (c) MOV DX, DS:[BP+6]

FLAG REGISTER



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

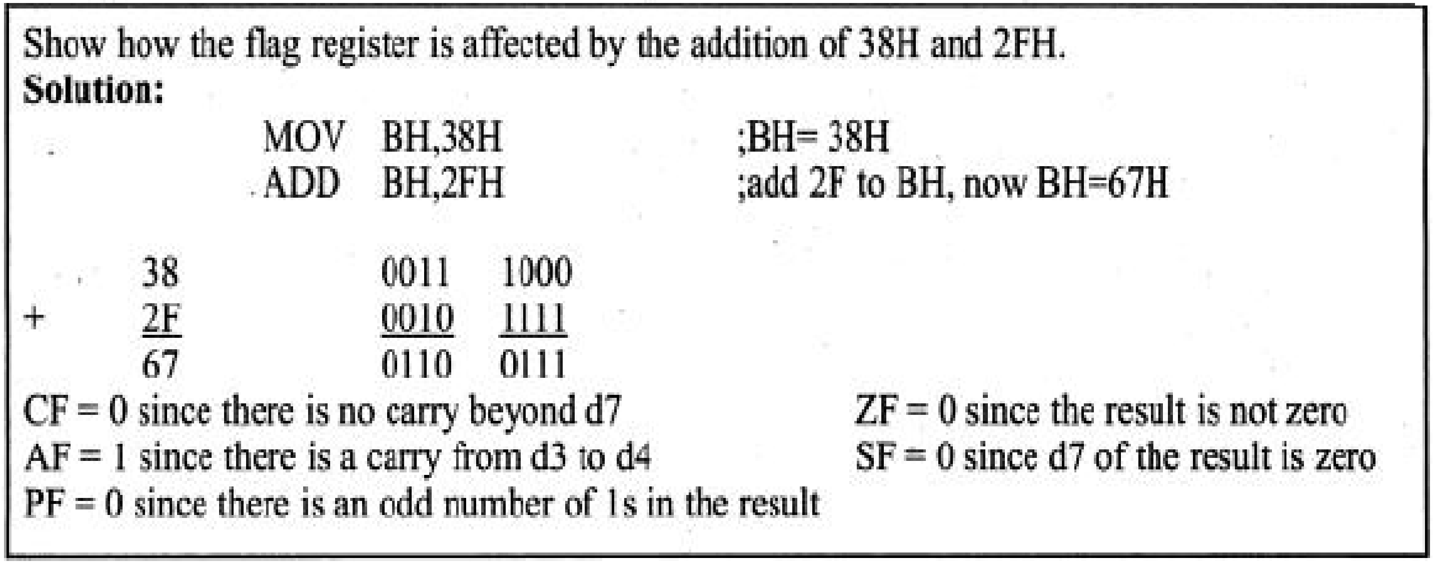
R = reserved
U = undefined
OF = overflow flag
DF = direction flag
IF = interrupt flag
TF = trap flag

SF = sign flag
ZF = zero flag
AF = auxiliary carry flag.
PF = parity flag
CF = carry flag

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

» ***Conditional flags*** – indicate some condition that resulted after an instruction was executed – CF, PF, AF, ZF, SF, and OF

» ***Control flags*** – used to control the operation of instructions before they are executed – TF, IF, DF



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
```

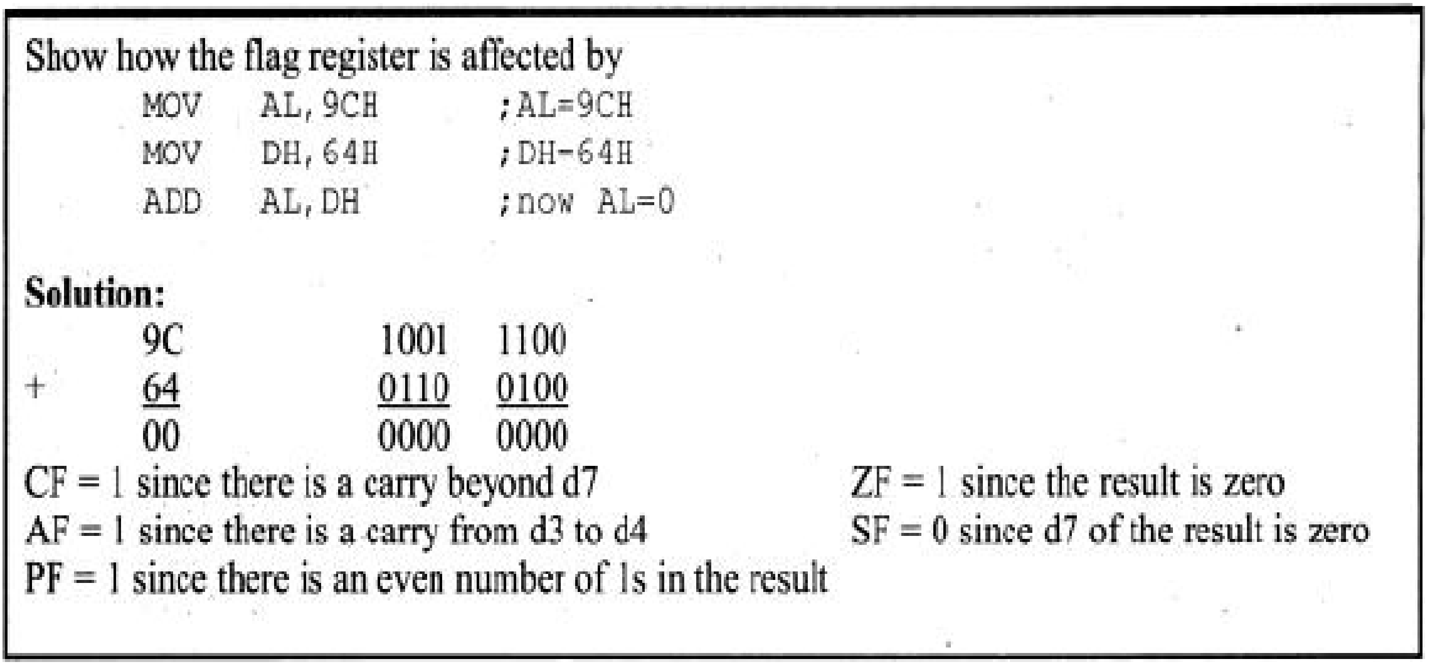
CF = 0 since there is no carry beyond d7

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



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
```

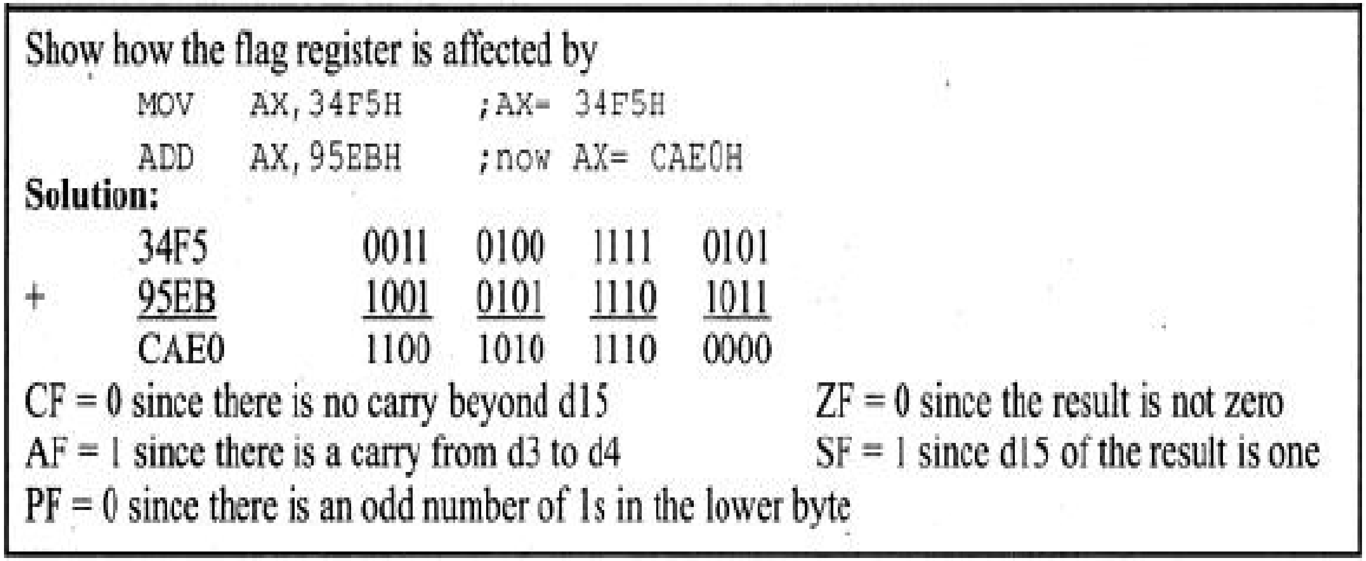
CF = 1 since there is a carry beyond d7

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



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
```

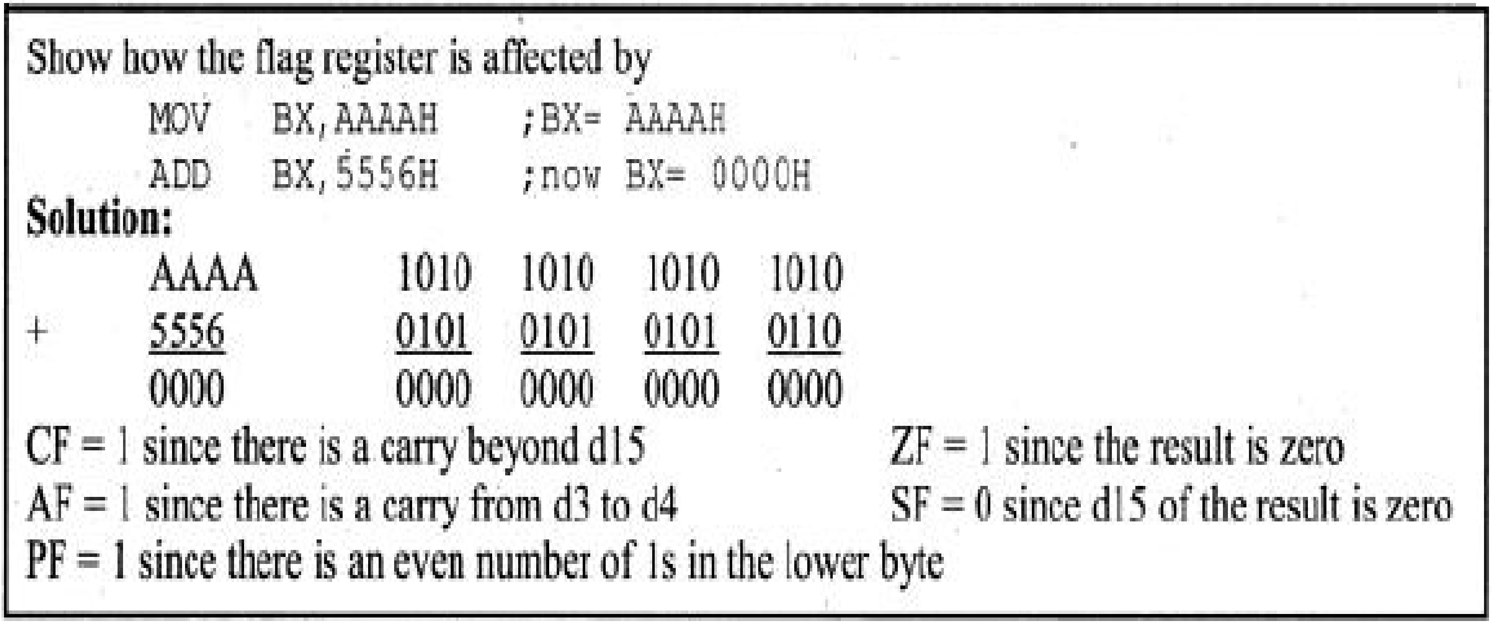
CF = 0 since there is no carry beyond d15

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



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
```

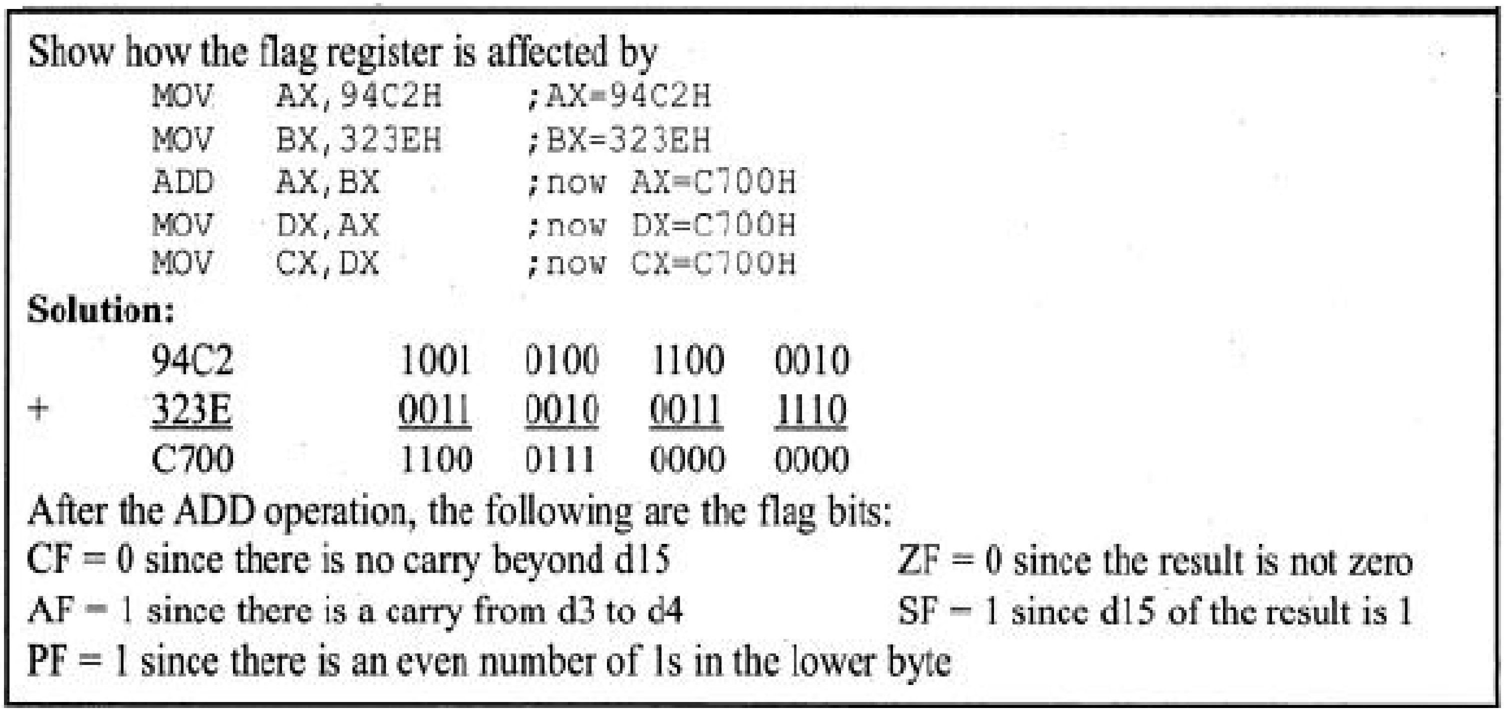
CF = 1 since there is a carry beyond d15

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



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

1. If the previous instruction performed the addition –

$$\begin{array}{r} 0010\ \ 0011\ \ 0100\ \ 0101 \\ +\underline{0011\ \ 0010\ \ 0001\ \ 1001} \\ 0101\ \ 0101\ \ 0101\ \ 1110 \end{array}$$

Then; CF = 0, PF = 0, AF = 0, ZF = 0, SF = 0, OF = 0

2. If the previous instruction performed the addition –

$$\begin{array}{r} 0101\ \ 0100\ \ 0011\ \ 1001 \\ +\underline{0100\ \ 0101\ \ 0110\ \ 1010} \\ 1001\ \ 1001\ \ 1010\ \ 0011 \end{array}$$

Then; CF = 0, PF = 1, AF = 0, ZF = 0, SF = 1, OF = 1

3. After adding two numbers 76H and 99H, what is the status of various flags?

```
  0111 0110
+ 1001 1001
-----------
  0000 1111
```

Then; CF = 1, PF = 1, AF = 0, ZF = 0, SF = 0

4. If two numbers 1234H and 95A5H are added, what is the status of various flags?

```
  0001 0010 0011 0100
+ 0001 0101 1010 0101
---------------------
  1010 0111 1101 1001
```

Then; CF = 0, AF = 0, ZF = 0, PF = 0, SF = 1, OF = 0

Here is the tip to identify the OF:

- Perform the addition in binary.
- Identify the carry out of MSB (C_n).
- Identify the carry into MSB (C_{n-1}).
- When these two are **not equal**, OF is set; i.e.,

$$\mathbf{OF = C_n \otimes C_{n-1}}$$

5. If two signed numbers 7FH and 01H are added, what is the status of various flags?

$$\begin{array}{r} 0\,1\,1\,1\ \ 1\,1\,1\,1 \\ +\,\underline{0\,0\,0\,0\ \ 0\,0\,0\,1} \\ 1\,0\,0\,0\ \ 0\,0\,0\,0 \end{array}$$

Then; CF = 0, AF = 1, ZF = 0, PF = 0, SF = 1, OF = 1

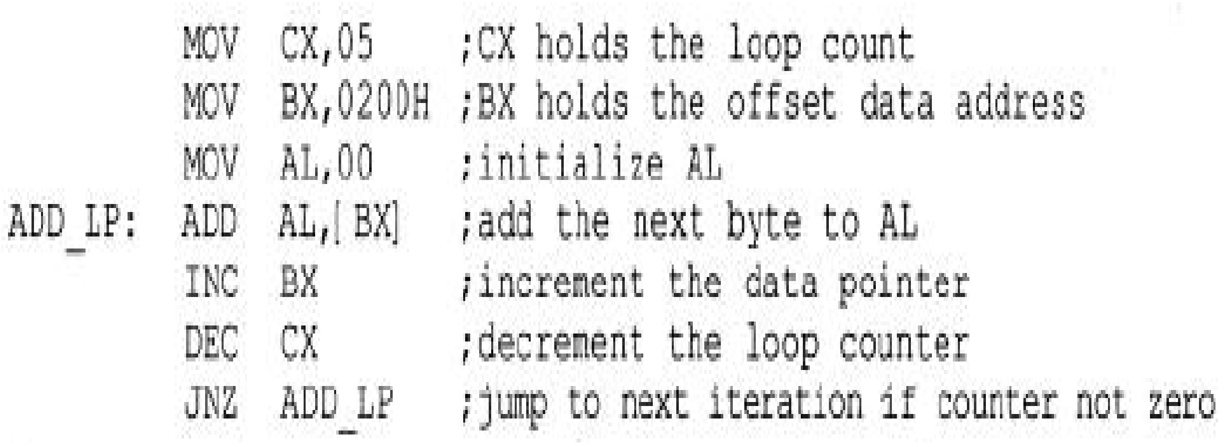
6. If two unsigned numbers 7FH and 01H are added, what is the status of various flags?

$$\begin{array}{r} 0\,1\,1\,1\ \ 1\,1\,1\,1 \\ +\,\underline{0\,0\,0\,0\ \ 0\,0\,0\,1} \\ 1\,0\,0\,0\ \ 0\,0\,0\,0 \end{array}$$

Then; CF = 0, AF = 1, ZF = 0, PF = 0

NOTE: Here, SF and OF are ignored because of unsigned numbers.

Use of Zero Flag for Looping



```
         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
```

REVIEW

1. Find the status of the CF, PF, AF, ZF, and SF for the following operations:

   (A) MOV BL, 9FH

   ADD BL, 61H

   (A) MOV AL, 23H

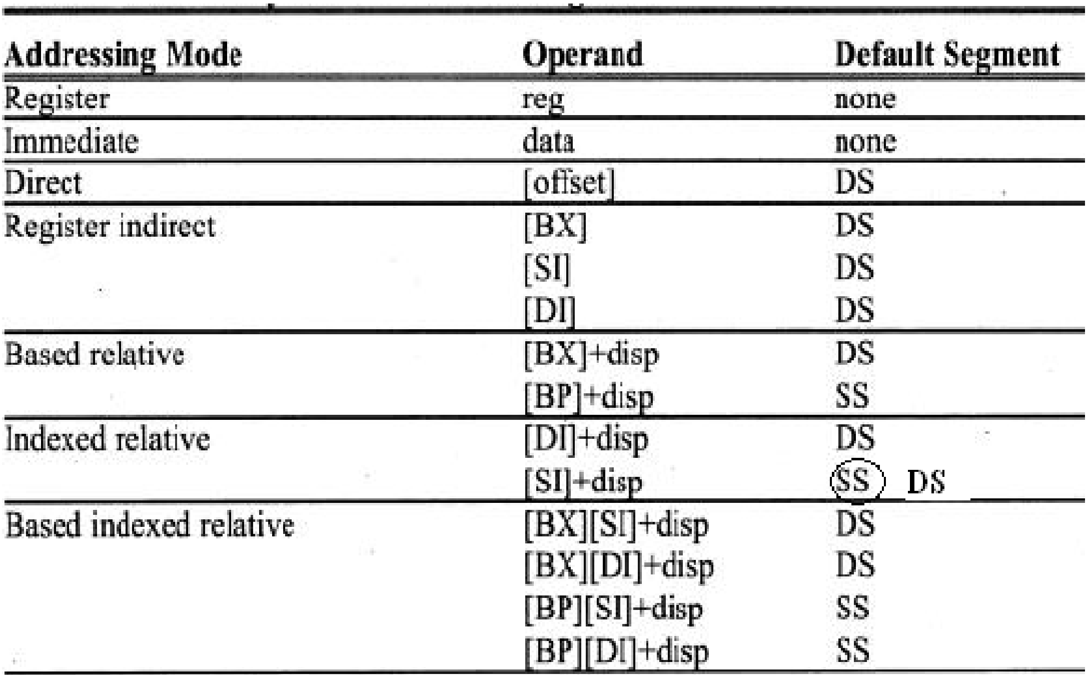
   ADD AL, 97H

   (A) MOV DX, 10FFH

   ADD DX, 1

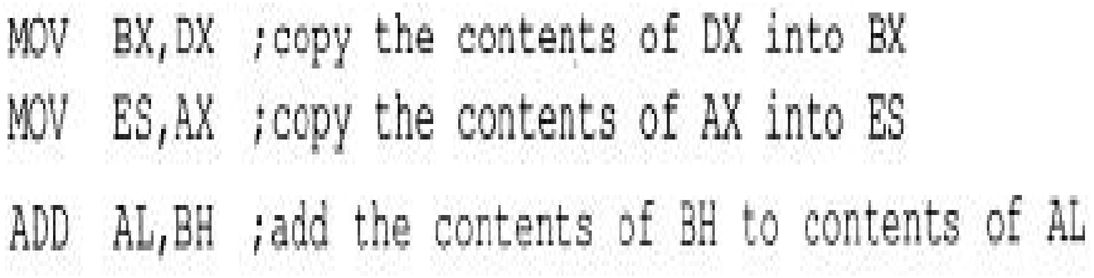
X86 ADDRESSING MODES

1. Register – MOV BX, DX
2. Immediate – MOV AX, 2550H
3. Direct – MOV DL, [2400]
4. Register Indirect – MOV AL, [BX]
5. Based Relative – MOV CX, [BX+10]
6. Indexed Relative – MOV DX, [SI]+5
7. Based Indexed Relative – MOV CL, [BX+DI+8]



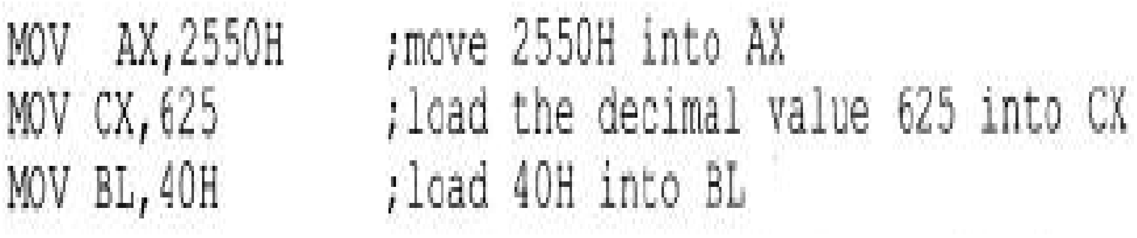
Addressing Mode	Operand	Default Segment
Register	reg	none
Immediate	data	none
Direct	[offset]	DS
Register indirect	[BX]	DS
	[SI]	DS
	[DI]	DS
Based relative	[BX]+disp	DS
	[BP]+disp	SS
Indexed relative	[DI]+disp	DS
	[SI]+disp	SS DS
Based indexed relative	[BX][SI]+disp	DS
	[BX][DI]+disp	DS
	[BP][SI]+disp	SS
	[BP][DI]+disp	SS

1. Register Addressing



```
MOV  BX,DX  ;copy the contents of DX into BX
MOV  ES,AX  ;copy the contents of AX into ES
ADD  AL,BH  ;add the contents of BH to contents of AL
```

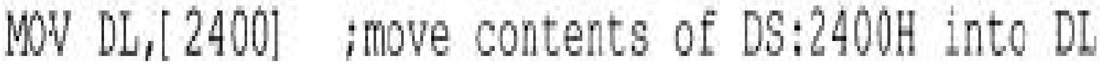
2. Immediate Addressing



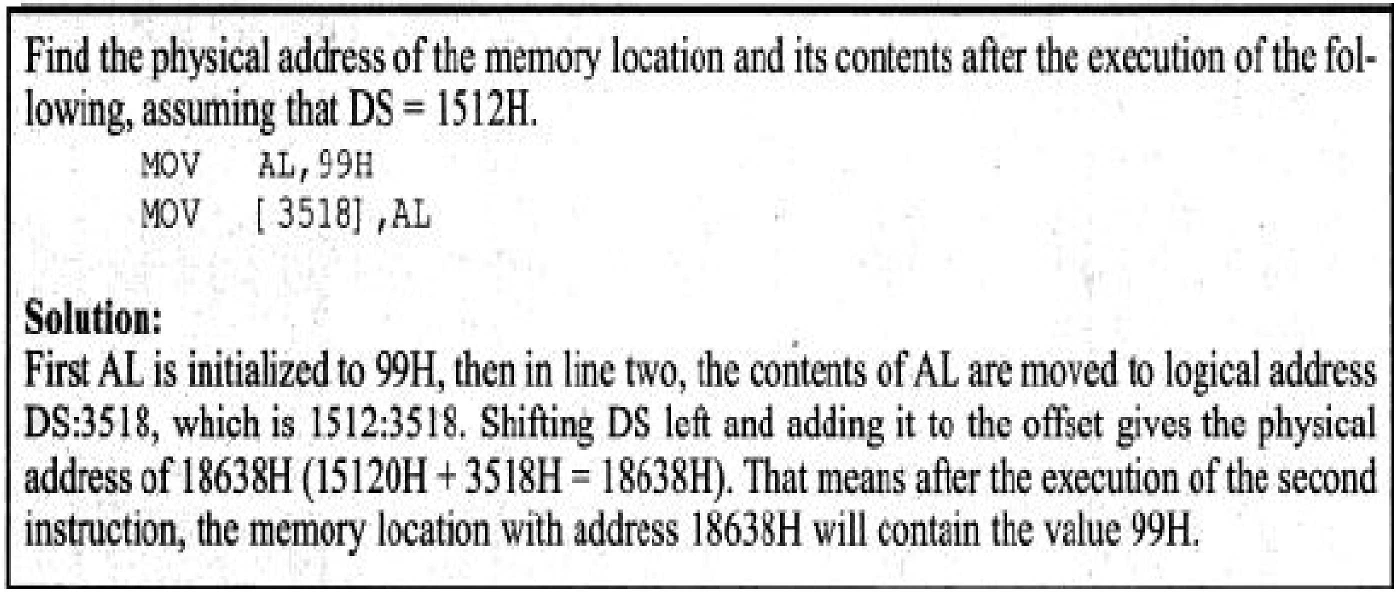
```
MOV  AX,2550H     ;move 2550H into AX
MOV CX,625        ;load the decimal value 625 into CX
MOV BL,40H        ;load 40H into BL
```

3. Direct Addressing

$PA = \left\{ DS \right\} : \left\{ \text{Direct Address} \right\}$



```
MOV DL,[2400]   ;move contents of DS:2400H into DL
```



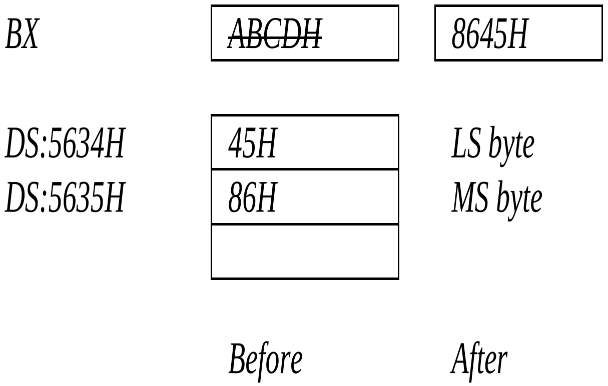
Find the physical address of the memory location and its contents after the execution of the following, assuming that DS = 1512H.

```
MOV    AL,99H
MOV    [3518],AL
```

Solution:

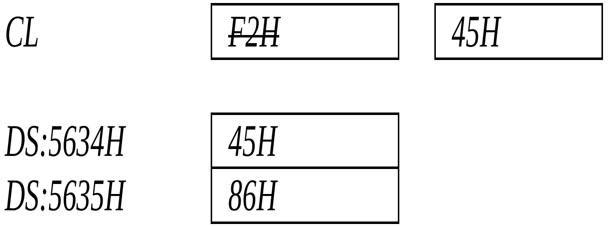
First AL is initialized to 99H, then in line two, the contents of AL are moved to logical address DS:3518, which is 1512:3518. Shifting DS left and adding it to the offset gives the physical address of 18638H (15120H + 3518H = 18638H). That means after the execution of the second instruction, the memory location with address 18638H will contain the value 99H.

Eg: *MOV BX, [5634]*



BX	~~*ABCDH*~~	*8645H*
DS:5634H	*45H*	*LS byte*
DS:5635H	*86H*	*MS byte*
	Before	*After*

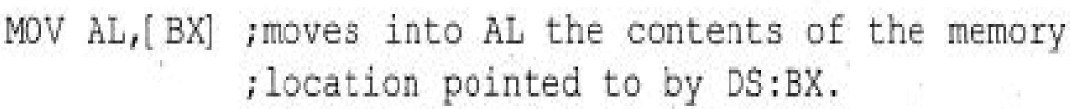
Eg: *MOV CL, [5634]*



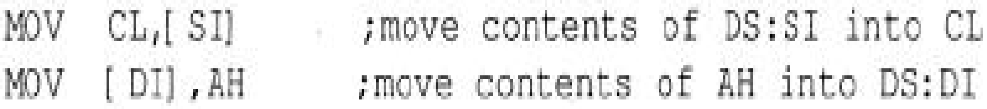
CL	~~*F2H*~~	*45H*
DS:5634H	*45H*	
DS:5635H	*86H*	

4. Register Indirect Addressing

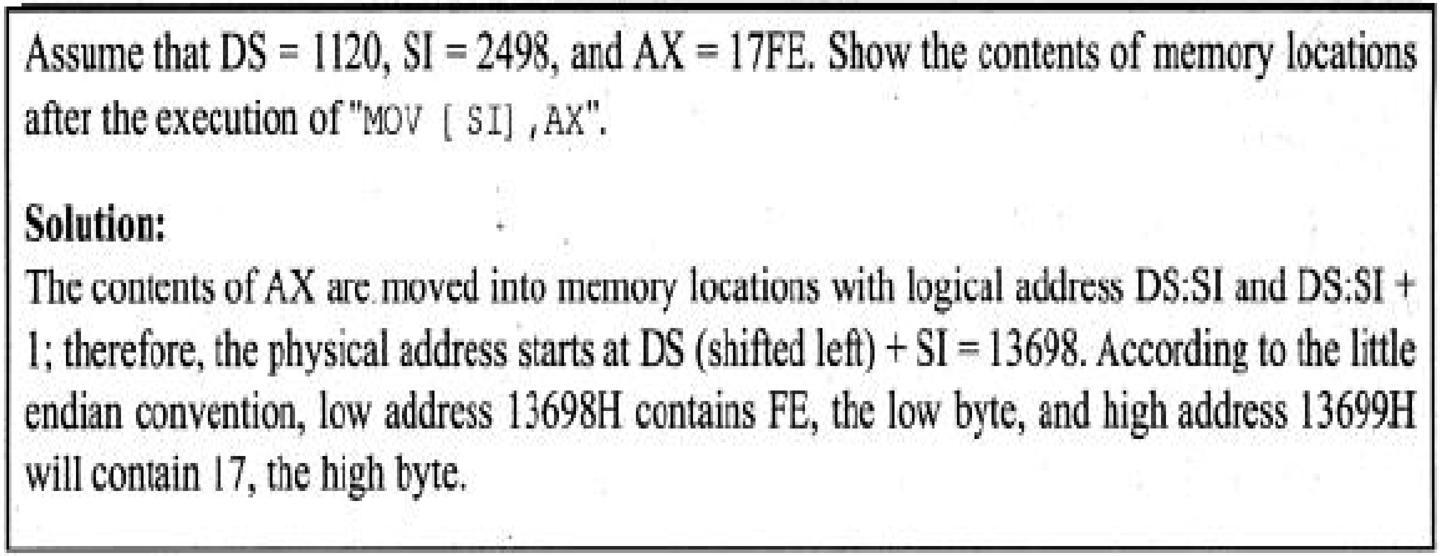
$$PA = \left\{ DS \right\} : \begin{Bmatrix} BX \\ SI \\ DI \end{Bmatrix}$$



```
MOV AL,[BX]  ;moves into AL the contents of the memory
             ;location pointed to by DS:BX.
```



```
MOV   CL,[SI]     ;move contents of DS:SI into CL
MOV   [DI],AH     ;move contents of AH into DS:DI
```



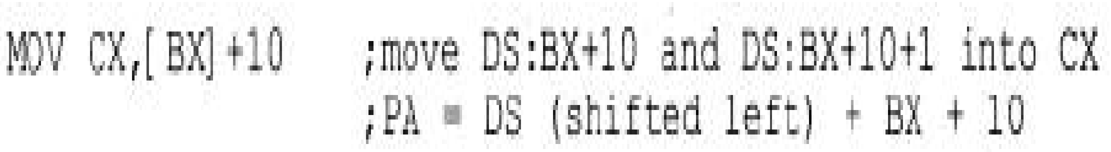
Assume that DS = 1120, SI = 2498, and AX = 17FE. Show the contents of memory locations after the execution of "`MOV [SI],AX`".

Solution:

The contents of AX are moved into memory locations with logical address DS:SI and DS:SI + 1; therefore, the physical address starts at DS (shifted left) + SI = 13698. According to the little endian convention, low address 13698H contains FE, the low byte, and high address 13699H will contain 17, the high byte.

5. Based Relative Addressing

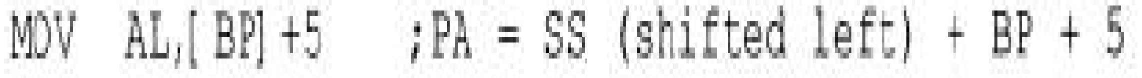
$$PA = \begin{Bmatrix} DS \\ \text{or} \\ SS \end{Bmatrix} : \begin{Bmatrix} BX \\ \text{or} \\ BP \end{Bmatrix} + 8 \text{ or } 16 \text{ bit displacement}$$



```
MOV CX,[BX]+10    ;move DS:BX+10 and DS:BX+10+1 into CX
                  ;PA = DS (shifted left) + BX + 10
```

- Alternative codings are "*MOV CX, [BX+10]*" or

  "*MOV CX, 10[BX]*"



```
MOV  AL,[BP]+5    ;PA = SS (shifted left) + BP + 5
```

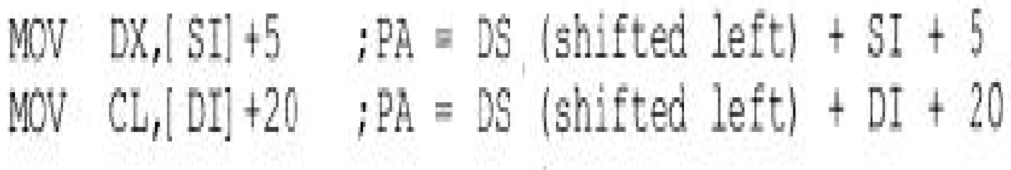
- Alternative codings are "*MOV AL, [BP+5]*" or

  "*MOV AL, 5[BP]*"

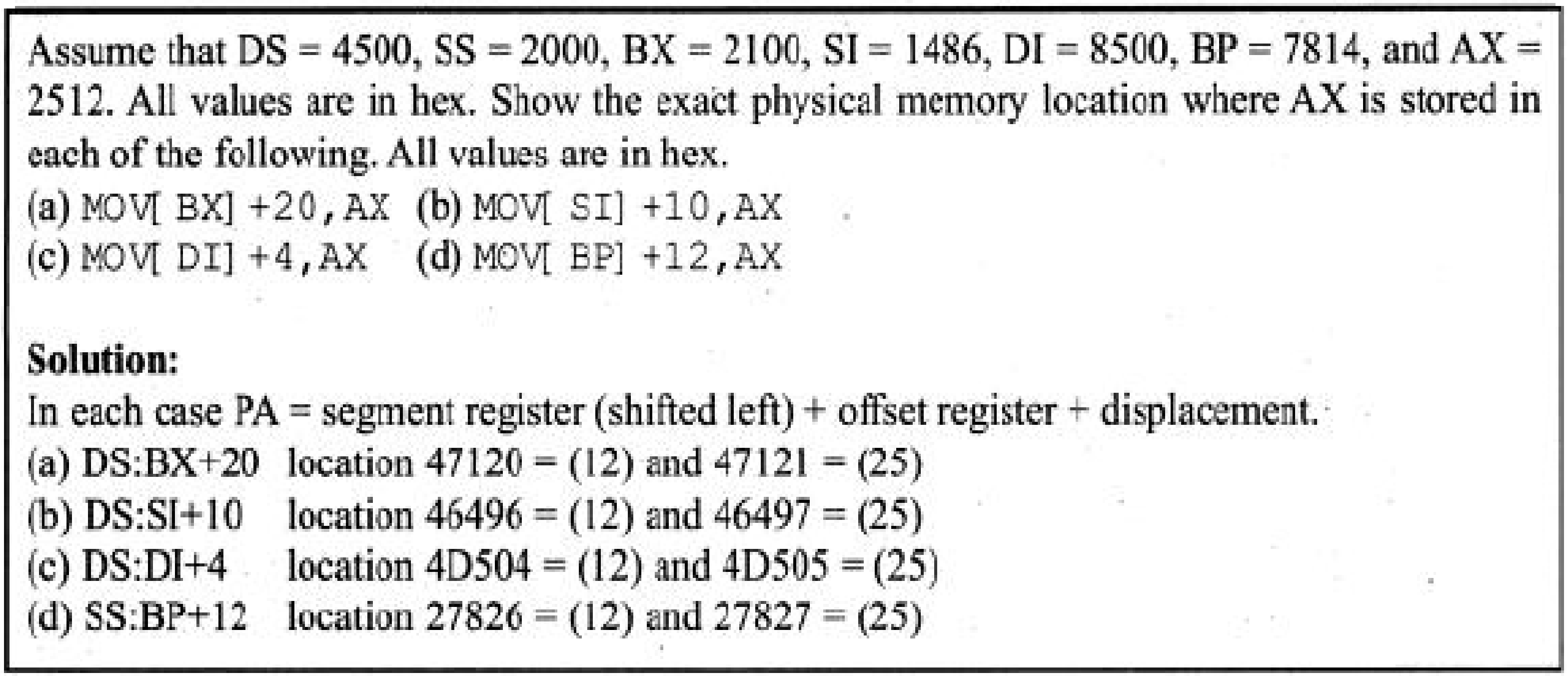
- In "*MOV AL, [BP+5]*", BP+5 is called the effective address;
- In "*MOV CX, [BX+10]*", BX+10 is called the effective address

6. Indexed Relative Addressing

$$PA = \left\{ \begin{matrix} DS \\ or \\ SS \end{matrix} \right\} : \left\{ \begin{matrix} SI \\ or \\ DI \end{matrix} \right\} + \text{8 or 16 bit displacement}$$



```
MOV  DX,[SI]+5    ;PA = DS (shifted left) + SI + 5
MOV  CL,[DI]+20   ;PA = DS (shifted left) + DI + 20
```



Assume that DS = 4500, SS = 2000, BX = 2100, SI = 1486, DI = 8500, BP = 7814, and AX = 2512. All values are in hex. Show the exact physical memory location where AX is stored in each of the following. All values are in hex.

(a) `MOV[BX]+20,AX` (b) `MOV[SI]+10,AX`

(c) `MOV[DI]+4,AX` (d) `MOV[BP]+12,AX`

Solution:

In each case PA = segment register (shifted left) + offset register + displacement.

(a) DS:BX+20 location 47120 = (12) and 47121 = (25)

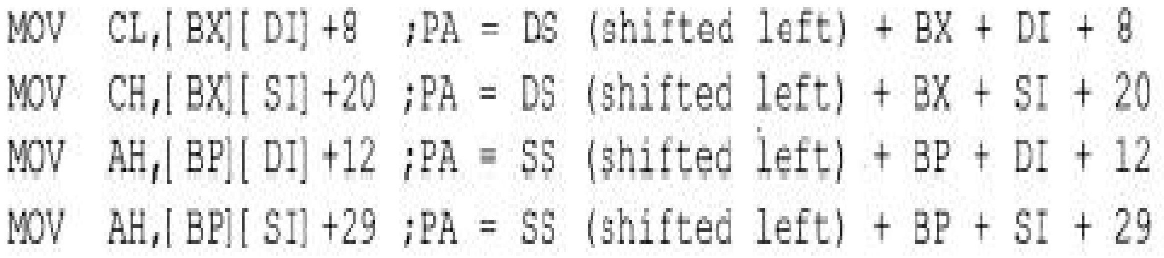
(b) DS:SI+10 location 46496 = (12) and 46497 = (25)

(c) DS:DI+4 location 4D504 = (12) and 4D505 = (25)

(d) SS:BP+12 location 27826 = (12) and 27827 = (25)

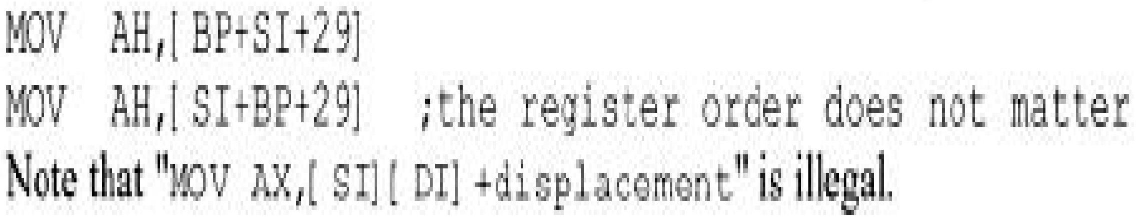
7. Based Indexed Addressing

$$PA = \left\{\begin{matrix} DS \\ or \\ SS \end{matrix}\right\} : \left\{\begin{matrix} BX \\ or \\ BP \end{matrix}\right\} + \left\{\begin{matrix} SI \\ or \\ DI \end{matrix}\right\} + \text{8 or 16bit displacement}$$



```
MOV   CL,[BX][DI]+8   ;PA = DS (shifted left) + BX + DI + 8
MOV   CH,[BX][SI]+20  ;PA = DS (shifted left) + BX + SI + 20
MOV   AH,[BP][DI]+12  ;PA = SS (shifted left) + BP + DI + 12
MOV   AH,[BP][SI]+29  ;PA = SS (shifted left) + BP + SI + 29
```

» These examples can also be written as –



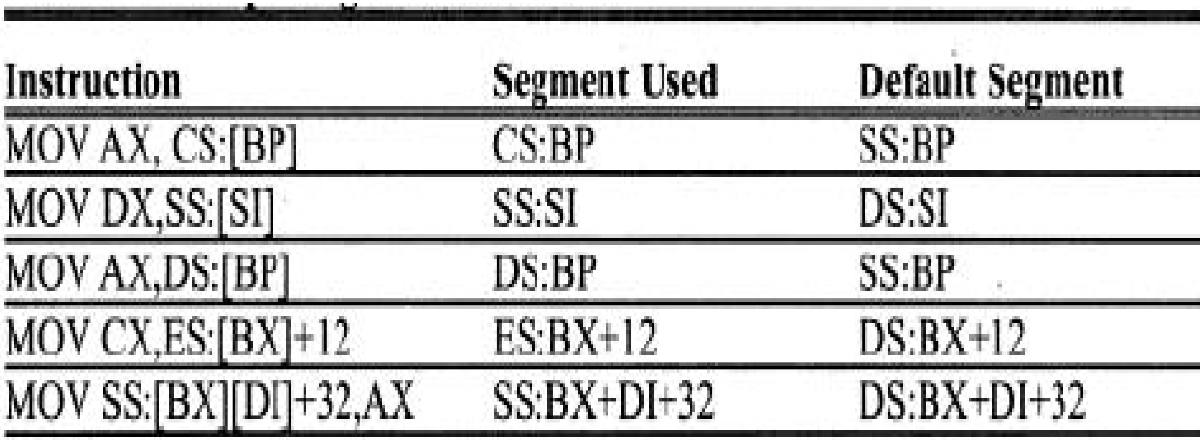
```
MOV   AH,[BP+SI+29]
MOV   AH,[SI+BP+29]   ;the register order does not matter
```

Note that "MOV AX,[SI][DI]+displacement" is illegal.

Segment Override Prefix

Segment	Offset	Special Purpose
CS	IP	Instruction address
DS	SI, DI, BX, an 8- or 16-bit number	Data address
SS	SP or BP	Stack address
ES	SI, DI, BX for string instructions	String destination address

- "MOV AL, [BX]", PA of the operand to be moved into AL is DS: BX
- "MOV AL, ES: [BX]“, PA will be ES: BX instead of DS: BX



Instruction	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

REVIEW

1. If CS = 1000H, DS = 2000H, SS = 3000H, SI = 4000H, DI = 5000H, BX = 6080H, BP = 7000H, AX = 25FFH, CX = 8791H, and DX = 1299H; calculate, the physical address of the memory accessed: (a) MOV [SI], AL (b) MOV [SI+BX+8], AH (c) MOV [BX], AX (d) MOV [DI+6], BX (e) MOV [DI][BX]+28, CX (f) MOV [BP][SI]+10, DX (g) MOV [3600], AX (h) MOV [BX]+30, DX (i) MOV [BP]+200, AX (j) MOV [BP+SI+100], BX (k) MOV [SI]+50, AH (l) MOV [DI+BP+100], AX

REVIEW

2. Identify the addressing mode for: (a) MOV AX, DS (b) MOV BX, 5678 (c) MOV CX, [3000] (d) MOV AL, CH (e) MOV [DI], BX (f) MOV AL, [BX] (g) MOV DX, [BP+DI+4] (h) MOV CX, DS (i) MOV [BP+6], AL (j) MOV AH, [BX+SI+50] (k) MOV BL, [SI]+10 (l) MOV [BP][SI]+12, AX

3. Show the content of the memory location, after the execution of:

(a)	MOV BX, 129FH	(b)	MOV DX, 8C63H
	MOV [1450], BX		MOV [2348], DX
	DS: 1450		DS: 2348
	DS: 1451		DS: 2348

15CS – 44

MICROPROCESSORS AND MICROCONTROLLERS

MODULE 1 – QUIZ 1

THE x86 MICROPROCESSOR

Mahesh Prasanna K.

Dept. of CSE, VCET.

1. The 80286 is a ______-bit microprocessor, where as the 80386 is a ______-bit microprocessor
2. Itanium has a ______-bit architecture
3. Which of the following registers cannot be split into high and low bytes? [CS, AX, DS, SS, BX, DX, CX, SI, DI]
4. Write the Assembly language instructions to add the vales 16H and ABH; place the result in AX register
5. Values cannot be moved directly into _____ registers

6. The largest 8-bit hex value is ______, and its decimal equivalent is ______

7. The largest 16-bit hex value is ______, and its decimal equivalent is ______

8. A segment is an area of memory that includes up to ______ bytes

9. A physical address is a ______-bit address; and offset address is a ______-bit address

10. For CS, ______ is used as the offset register

11. If BX = 1234H and the instruction “MOV [2400], BX” were executed; then, the contents of memory location at offset 2400 is ______ and the contents of memory location at offset 2401 ______

12. The stack is a section of RAM used for temporary storage [TRUE/FALSE]

13. The Carry Flag will be set to 1 in an 8-bit addition, if there is a carry out from bit ______

14. The Auxiliary Carry Flag will be set to 1 in an 8-bit addition, if there is a carry out from bit ______

1. The 80286 is a ______-bit microprocessor, where as the 80386 is a ______-bit microprocessor (16, 32)
2. Itanium has a ______-bit architecture (64)
3. Which of the following registers cannot be split into high and low bytes? [CS, AX, DS, SS, BX, DX, CX, SI, DI] (CS, DS, SS, SI, and DI)
4. Write the Assembly language instructions to add the vales 16H and ABH; place the result in AX register (MOV AX, 16H ADD AX, ABH)
5. Values cannot be moved directly into _____ registers (CS, DS, ES, and SS)

6. The largest 8-bit hex value is ______, and its decimal equivalent is ______ (FFFFH, 65535)

7. The largest 16-bit hex value is ______, and its decimal equivalent is ______ (FFH, 255)

8. A segment is an area of memory that includes up to ______ bytes (64K)

9. A physical address is a ______-bit address; and offset address is a ______-bit address (20, 16)

10. For CS, ______ is used as the offset register (IP)

11. If BX = 1234H and the instruction “MOV [2400], BX” were executed; then, the contents of memory location at offset 2400 is _____ and the contents of memory location at offset 2401 _____ (34, 12)

12. The stack is a section of RAM used for temporary storage [TRUE/FALSE]

13. The Carry Flag will be set to 1 in an 8-bit addition, if there is a carry out from bit _____ (7)

14. The Auxiliary Carry Flag will be set to 1 in an 8-bit addition, if there is a carry out from bit _____ (3)

15CS – 44

MICROPROCESSORS AND MICROCONTROLLERS

MODULE 1

ASSEMBLY LANGUAGE PROGRAMMING

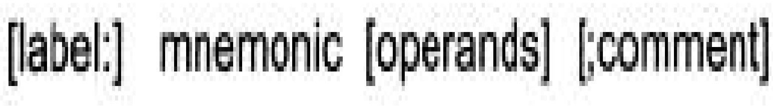
Mahesh Prasanna K.

Dept. of CSE, VCET.

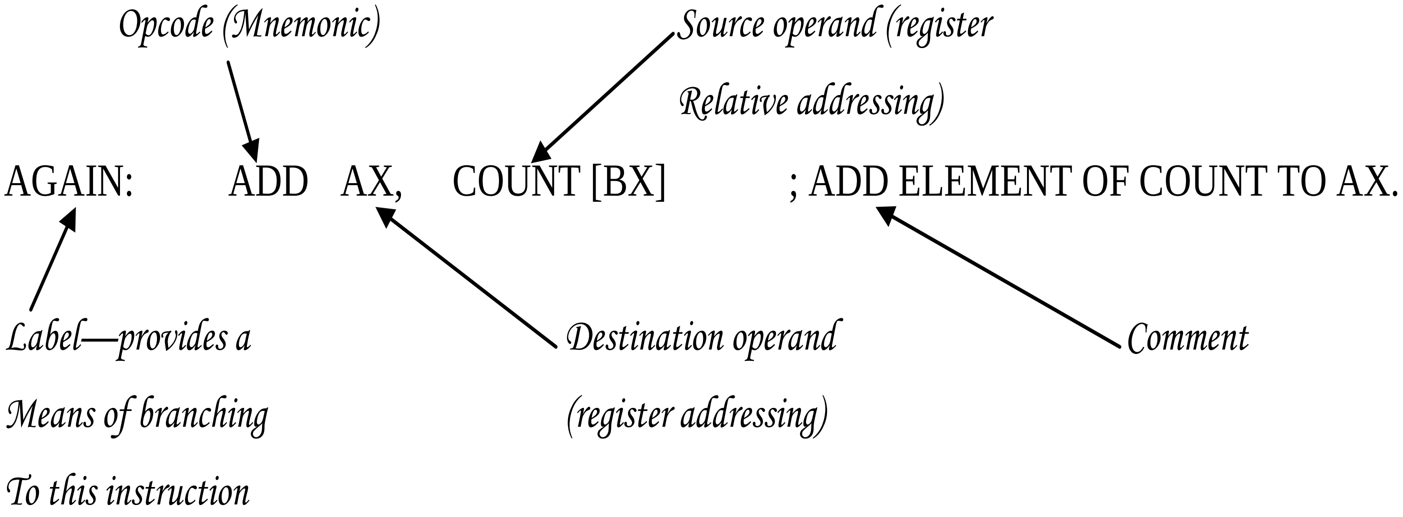
DIRECTIVES AND A SAMPLE PROGRAM

- A given Assembly language program (ALP) is a series of statements. There are two types of statements:

1. *Assembly language instructions* – instructions to the microprocessor to do the specific task. (*E.g.: MOV, ADD, etc.*)
2. *Pseudo instructions/Directives* – give directions to the assembler about how it should translate. (*E.g.: DB, DW, ASSUME, etc.*)
   - These instructions are not translated into machine code
   - Used by the assembler to organize the program as well as other output files



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



Opcode (Mnemonic)

Source operand (register Relative addressing)

AGAIN: ADD AX, COUNT [BX] ; ADD ELEMENT OF COUNT TO AX.

Label—provides a Means of branching To this instruction

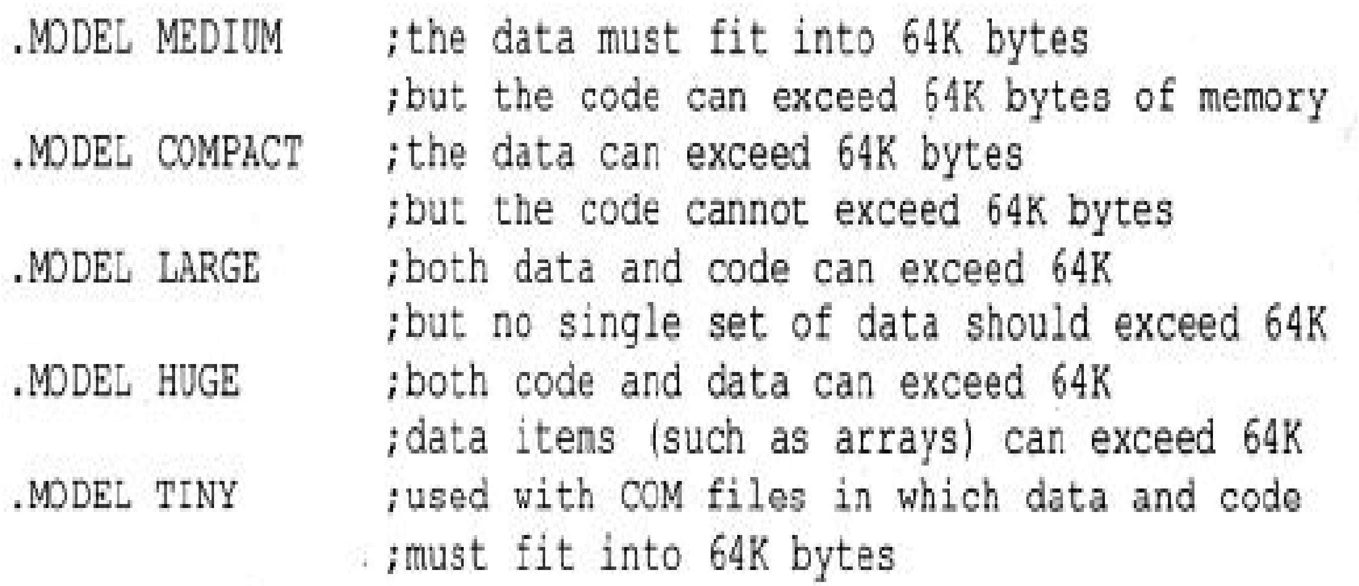
Destination operand (register addressing)

Comment

Model Definition

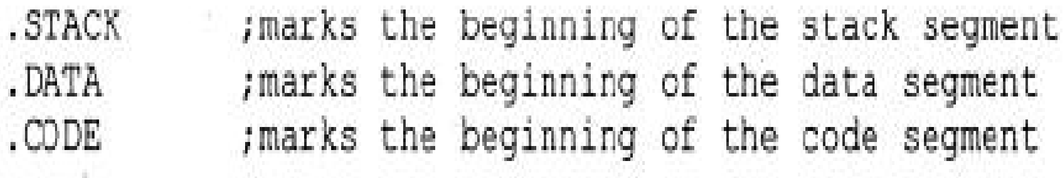
» •*MODEL* – directive selects the size of the memory model

•*MODEL SMALL ; this directive defines the model as small*

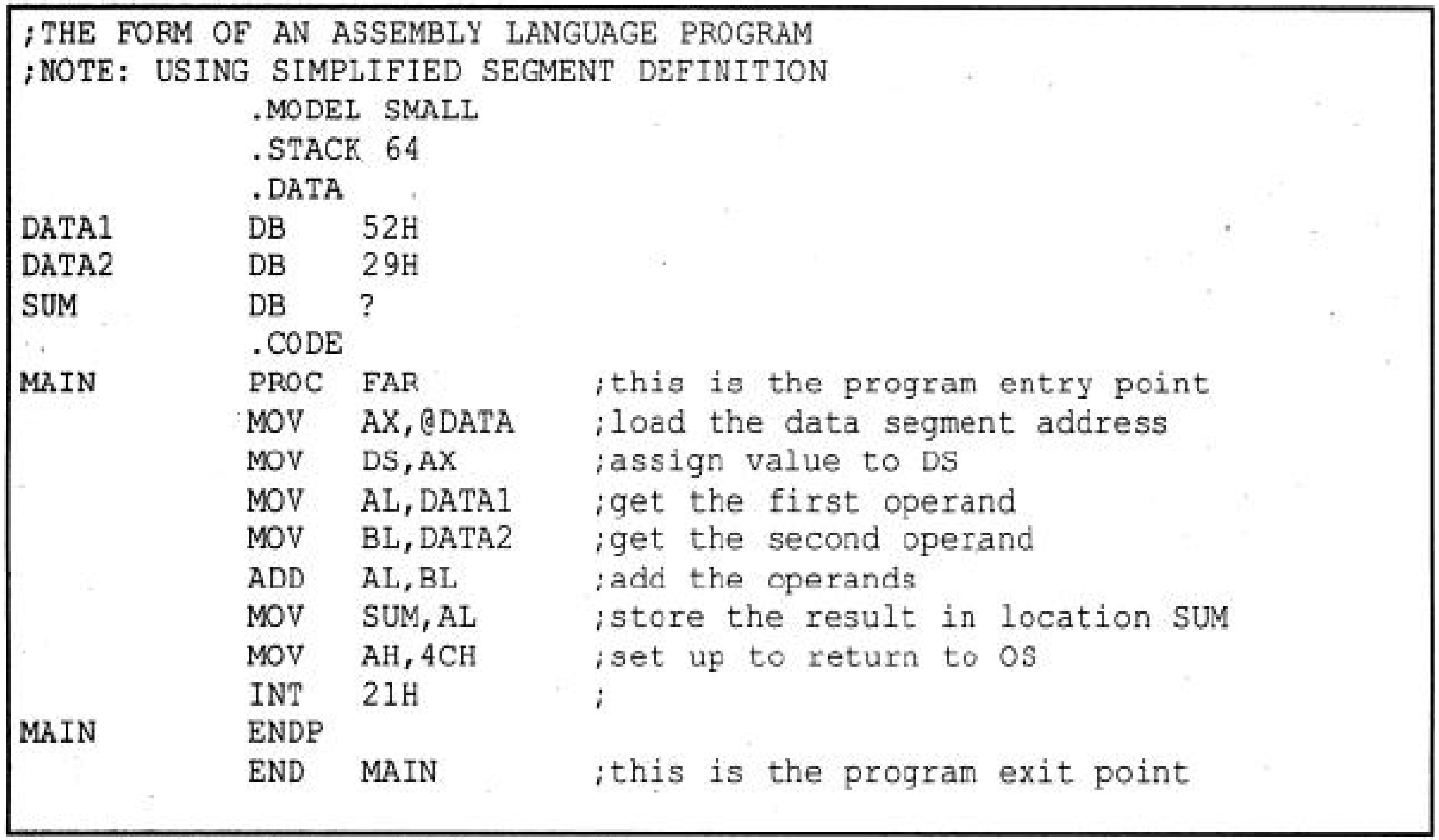


```
.MODEL MEDIUM     ;the data must fit into 64K bytes
                  ;but the code can exceed 64K bytes of memory
.MODEL COMPACT    ;the data can exceed 64K bytes
                  ;but the code cannot exceed 64K bytes
.MODEL LARGE      ;both data and code can exceed 64K
                  ;but no 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
```

Segment Definition



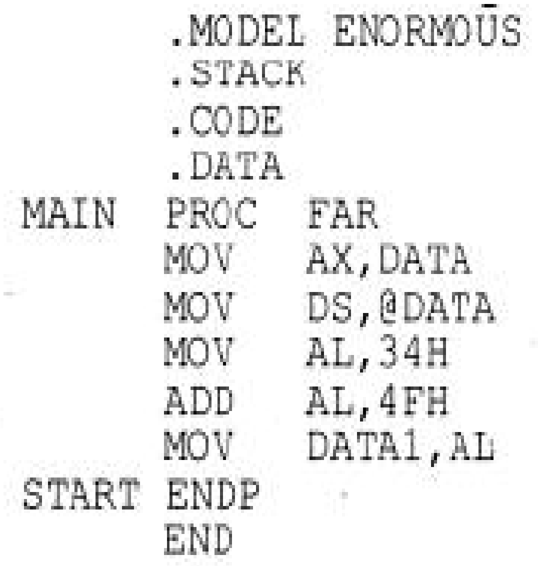
```
.STACK      ;marks the beginning of the stack segment
.DATA       ;marks the beginning of the data segment
.CODE       ;marks the beginning of the code segment
```



```
;THE 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
```

REVIEW

1. Find the errors in the following:



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

ASSEMBLE, LINK, AND RUN A PROGRAM

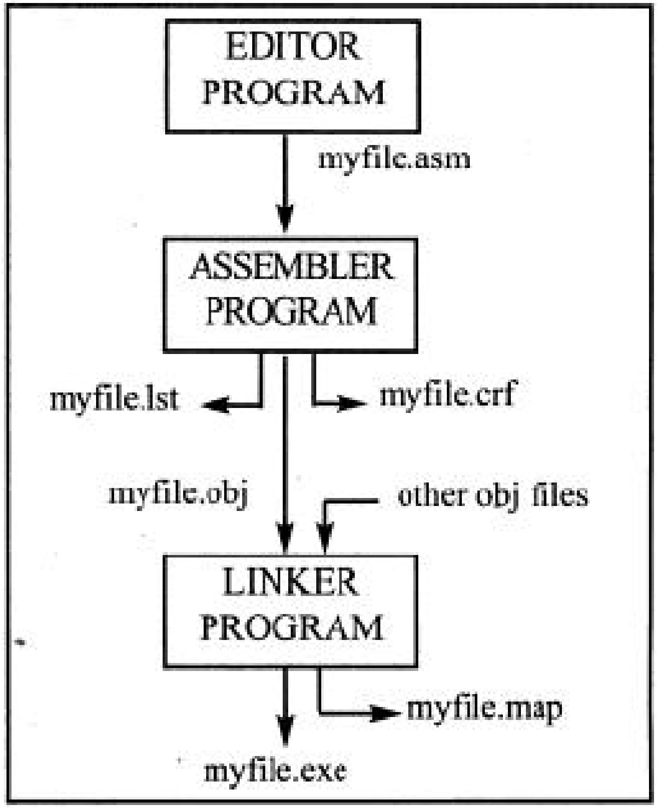
Step	Input	Program	Output
1. Edit the program	Keyboard	Editor	myfile.asm
2. Assemble the program	myfile.asm	MASM or TASM	myfile.obj
3. Link the program	myfile.obj	LINK or TLINK	myfile.exe

(.lst) – all the opcodes and the offset addresses, as well as errors

C>type myfile.lst | more

(.obj) – produces the executable program (.exe)

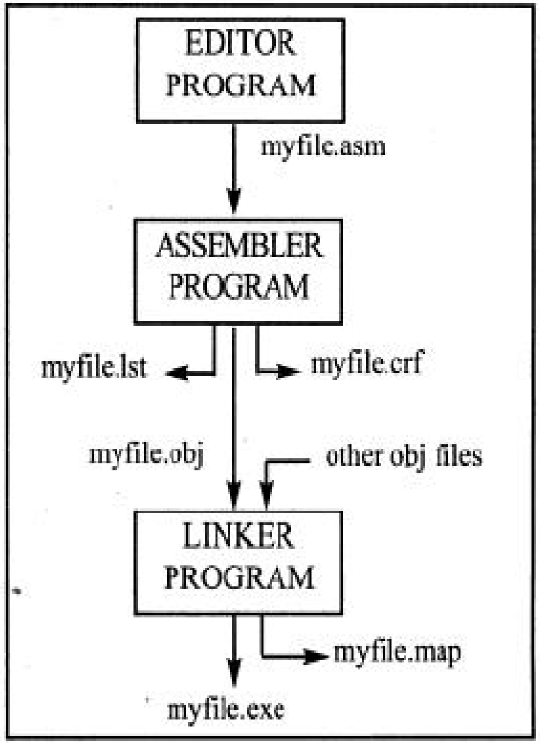
use DEBUG to execute the program and analyze the results

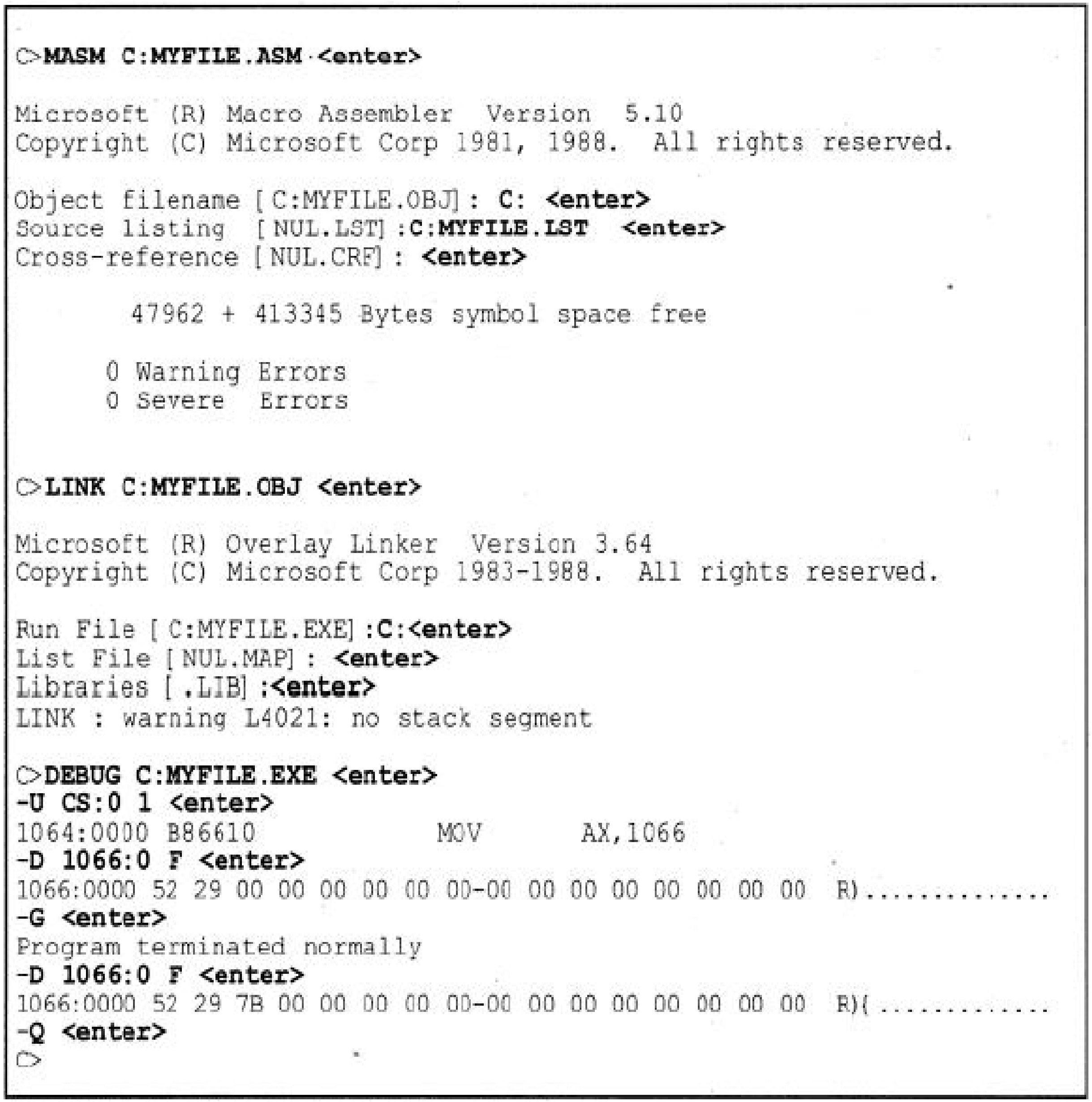


(.crf) – an alphabetical list of all symbols and tables used in the program as well as program line numbers

LINK program sets up the file, so that, it can be loaded by the OS and executed

Run program in OS level, type C:>myfile – OS loads the program – mapping (program is mapped into physical memory

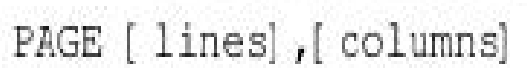




```
C>MASM C:MYFILE.ASM <enter>

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.

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

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

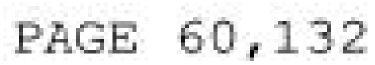
PAGE and TITLE Directives

- Used make the “.lst” file more readable
- The PAGE directive tells the printer how the list should be printed.



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

  - In the default mode, the output will have 66 lines per page and with a maximum of 80 characters per line



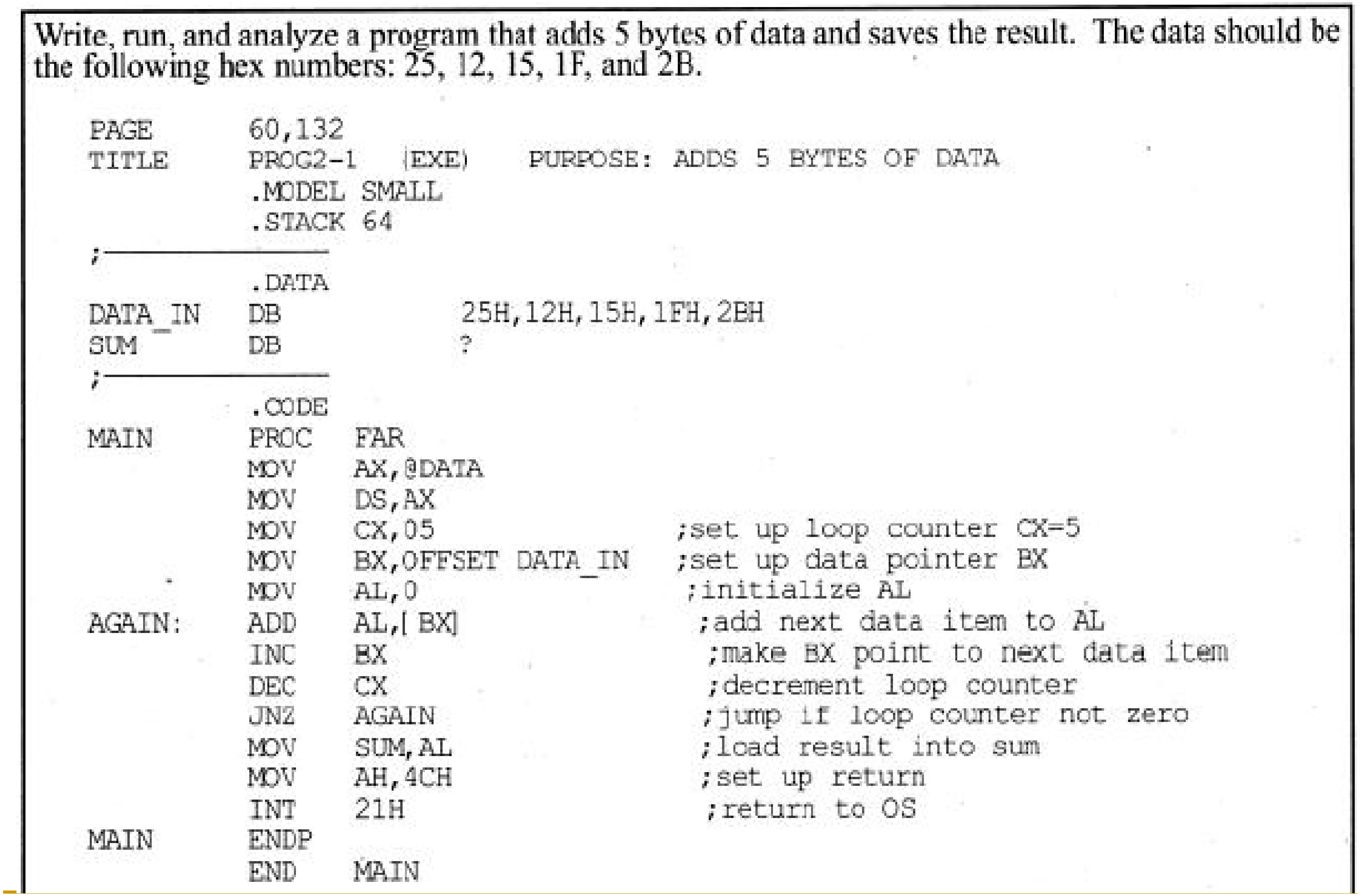
```
PAGE 60,132
```

- TITLE directive can be used to instruct the assembler to print the title of the program on the top of each page

REVIEW

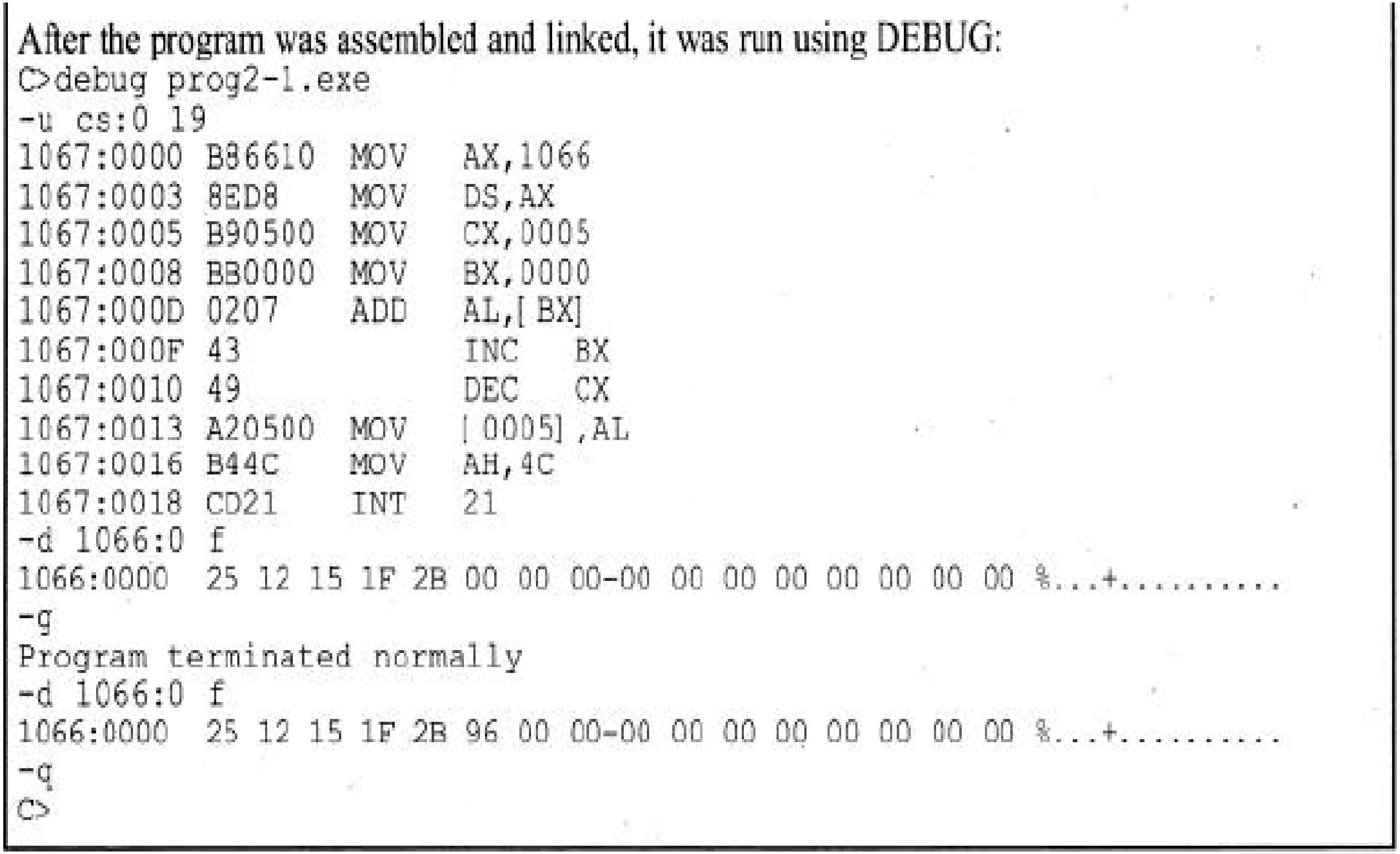
1. List the steps in getting a ready to run Assembly language program

MORE SAMPLE PROGRAMS



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

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



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

```
C>debug prog2-1.exe
-u cs:0 19
1067:0000 B86610  MOV   AX,1066
1067:0003 8ED8    MOV   DS,AX
1067:0005 B90500  MOV   CX,0005
1067:0008 BB0000  MOV   BX,0000
1067:000D 0207    ADD   AL,[ BX]
1067:000F 43            INC   BX
1067:0010 49            DEC   CX
1067:0013 A20500  MOV   [ 0005] ,AL
1067:0016 B44C    MOV   AH,4C
1067:0018 CD21    INT   21
-d 1066:0 f
1066:0000  25 12 15 1F 2B 00 00 00-00 00 00 00 00 00 00 00 %...+..........
-g
Program terminated normally
-d 1066:0 f
1066:0000  25 12 15 1F 2B 96 00 00-00 00 00 00 00 00 00 00 %...+..........
-q
C>
```

- **INC destination** – adds 1 to the specified destination
  - Flags affected: AF, OF, PF, SF, and ZF. The CF is not affected

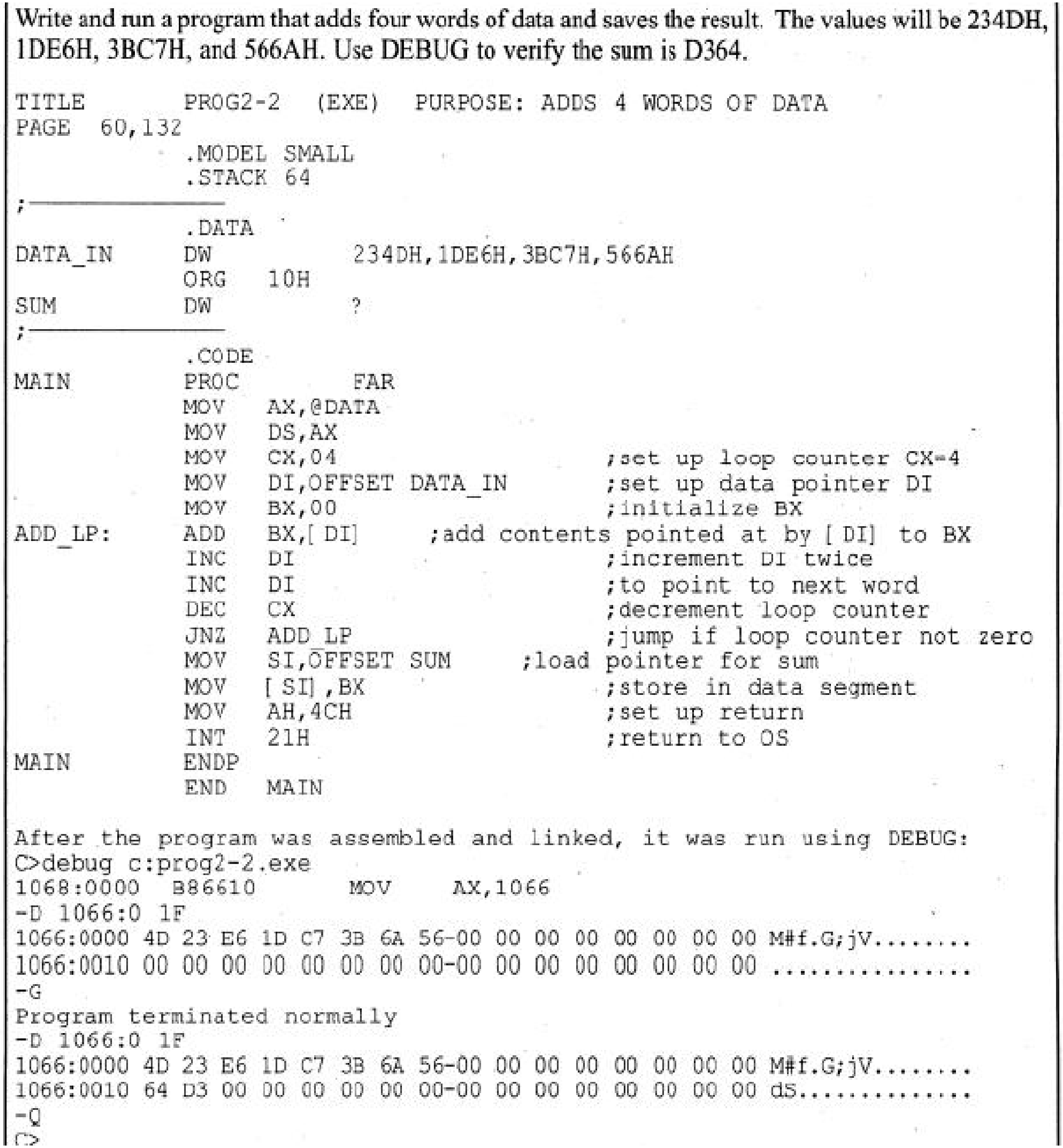
Eg1: INC AL ; Add one to the contents of AL.

Eg2: INC BX ; Add one to the contents of BX.

- **DEC destination** – subtract 1 from the specified destination
  - Flags affected: AF, OF, PF, SF, and ZF. The CF is not affected

Eg: DEC AL ; Subtract 1 from the contents of AL.

- **JNZ label** – jump if not zero; if ZF = 0, jumps to the label specified. Checks for zero flag
  - **See MASM List for Program 2-1**



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

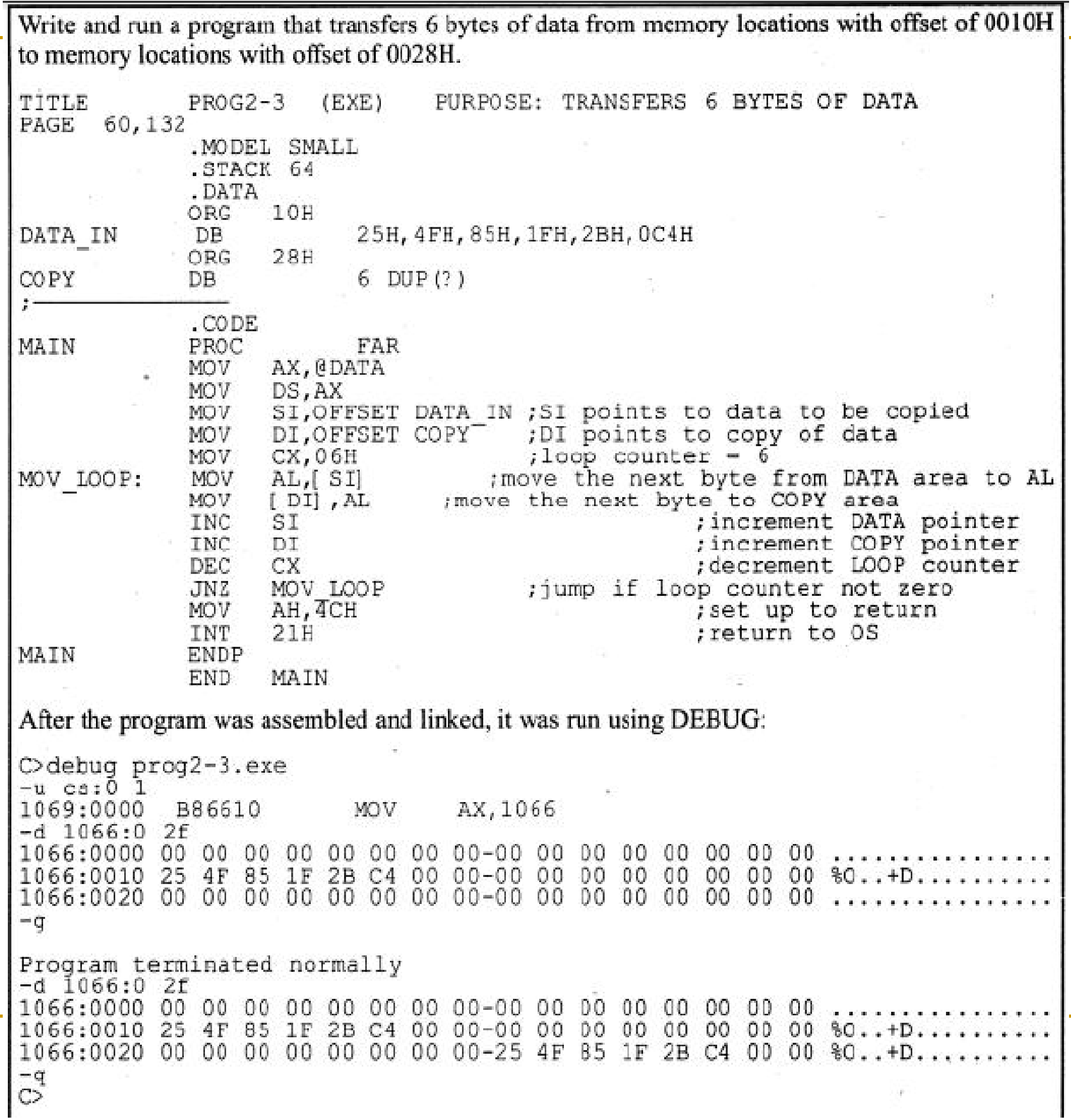
```
TITLE        PROG2-2   (EXE)   PURPOSE: ADDS 4 WORDS OF DATA
PAGE   60,132
             .MODEL SMALL
             .STACK 64
;--------------
             .DATA
DATA_IN      DW           234DH,1DE6H,3BC7H,566AH
             ORG    10H
SUM          DW           ?
;--------------
             .CODE
MAIN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

```
After the program was assembled and linked, it was run using DEBUG:
C>debug c:prog2-2.exe
1068:0000   B86610       MOV    AX,1066
-D 1066:0 1F
1066:0000 4D 23 E6 1D C7 3B 6A 56-00 00 00 00 00 00 00 00 M#f.G;jV........
1066:0010 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-G
Program terminated normally
-D 1066:0 1F
1066:0000 4D 23 E6 1D C7 3B 6A 56-00 00 00 00 00 00 00 00 M#f.G;jV........
1066:0010 64 D3 00 00 00 00 00 00-00 00 00 00 00 00 00 00 dS..............
-Q
C>
```

- **OFFSET** – tells the assembler to determine the offset or displacement of a named data item (variable) from the start of the segment

  Eg: MOV AX, OFFSET MES1 ; Loads the offset of variable MES1 in AX register.

- **ORG directive** – Used to set the offset addresses for data items.
  - In the above program, the ORG directive causes SUM to be stored at DS: 0010



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

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

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

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

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

REVIEW

1. Explain INC instruction and DEC instruction with example
2. State the difference between the following two instructions:

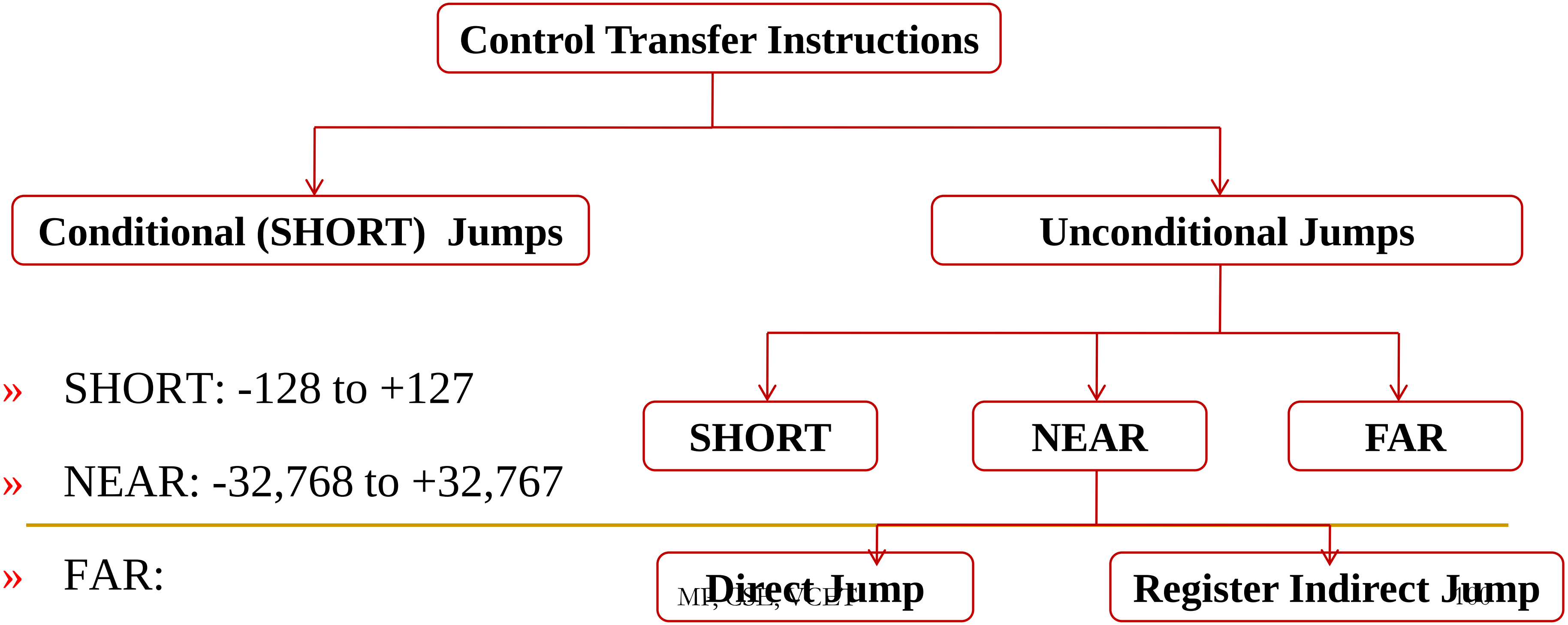
   MOV BX, DATA1 MOV BX, OFFSET DATA1

3. State the difference between the following two instructions:

   ADD AX, BX ADD AX, [BX]

CONTROL TRANSFER INSTRUCTIONS

- In an ALP, instructions are executed sequentially
- It is often necessary to transfer program control to a different location
- Since the CS: IP registers always point to the address of the next instruction to be executed
- Hence, they must be updated when a control transfer instruction is executed



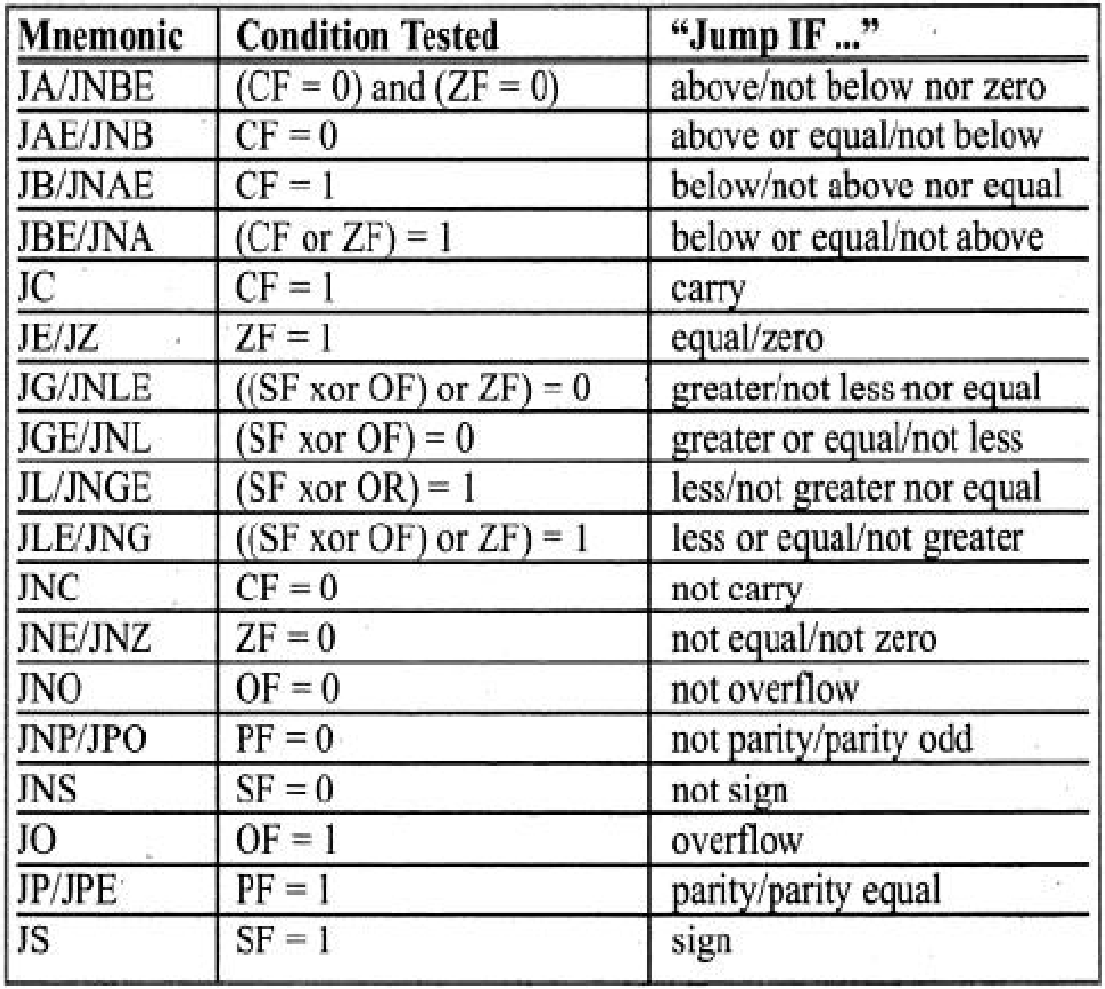
- SHORT: -128 to +127
- NEAR: -32,768 to +32,767
- FAR:

FAR and NEAR

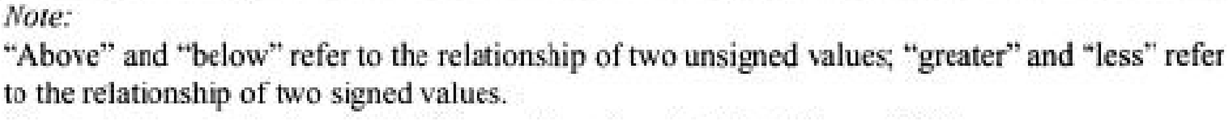
- If control is transferred to a memory location within the current code segment, it is ***NEAR*** [***intra-segment*** (within segment) jump]
  - In a NEAR jump, the IP is updated and CS remains the same
- If control is transferred to a memory location outside the current code segment, it is a ***FAR*** [***intersegment*** (between segments) jump]
  - In a FAR jump, both CS and IP have to be updated to the new values.

Conditional Jumps

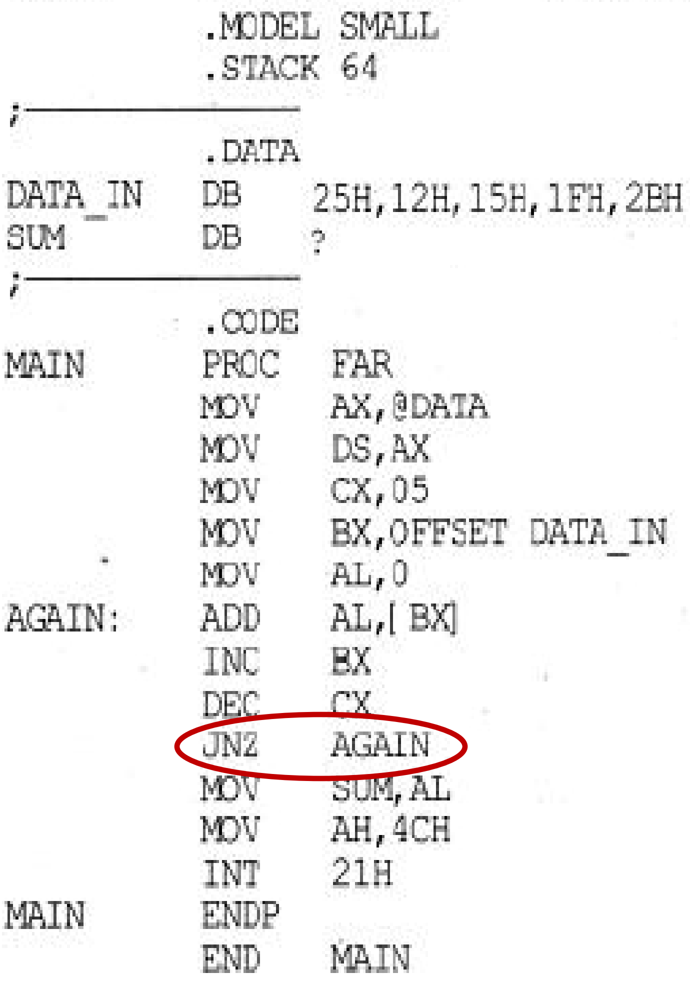
» If the condition is met, the control will be transferred to a new location



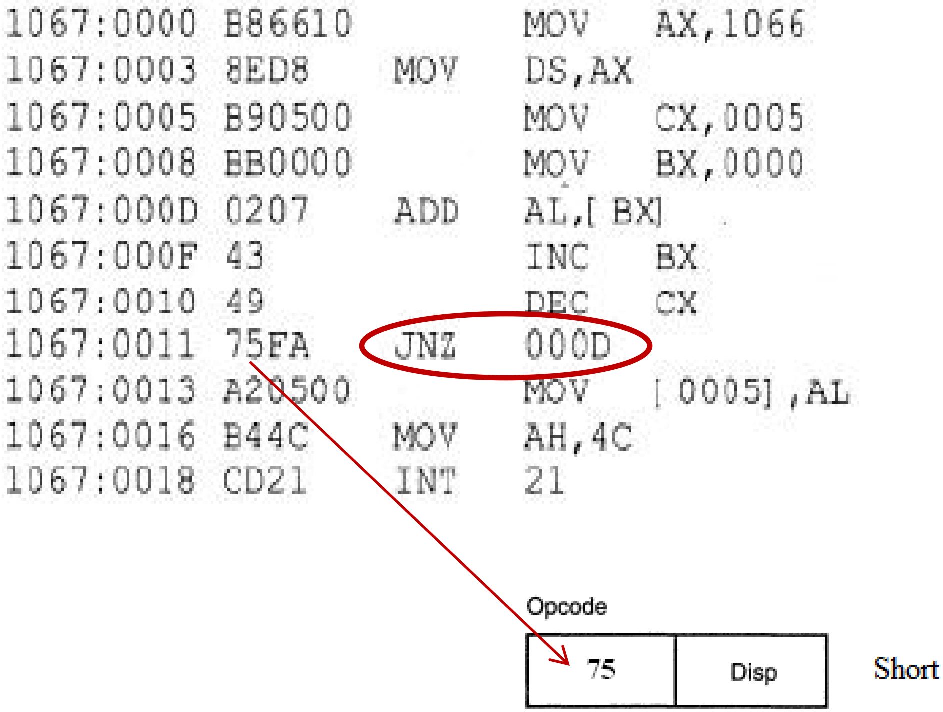
Mnemonic	Condition Tested	"Jump IF ..."
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not 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



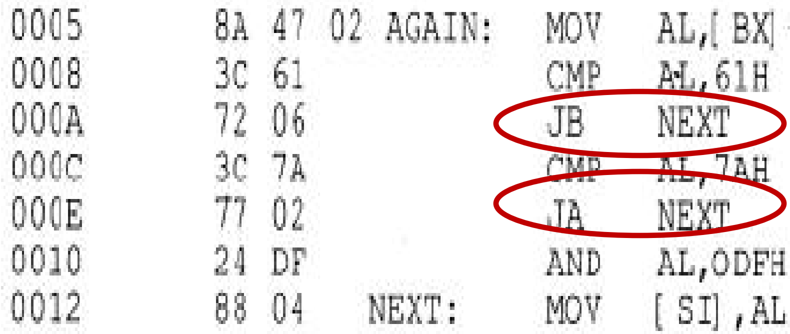
Note:
"Above" and "below" refer to the relationship of two unsigned values; "greater" and "less" refer to the relationship of two signed values.



```
          .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
          MOV    BX,OFFSET DATA_IN
          MOV    AL,0
AGAIN:    ADD    AL,[ BX]
          INC    BX
          DEC    CX
          JNZ    AGAIN
          MOV    SUM,AL
          MOV    AH,4CH
          INT    21H
MAIN      ENDP
          END    MAIN
```



```
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
```



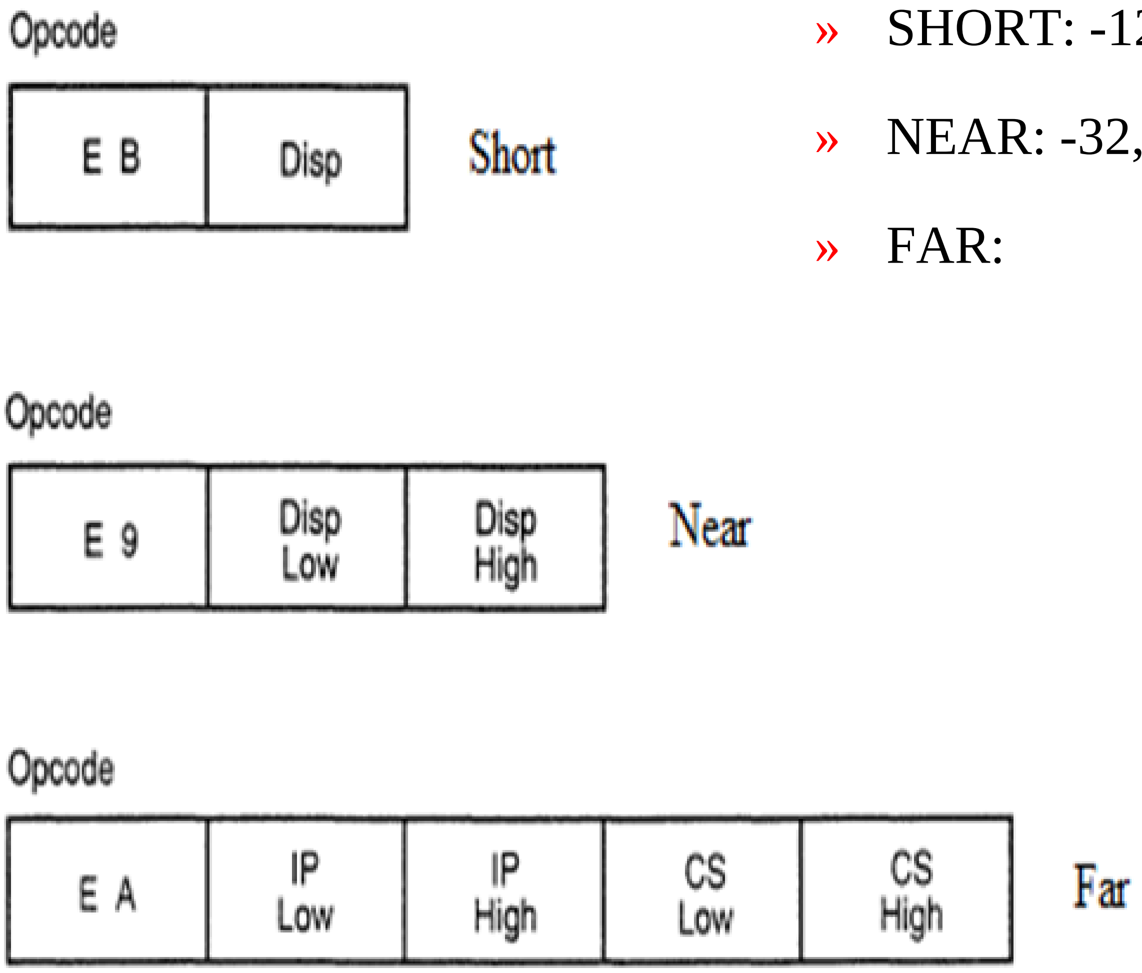
```
0005     8A 47 02  AGAIN:   MOV   AL,[ BX] +2
0008     3C 61              CMP   AL,61H
000A     72 06              JB    NEXT
000C     3C 7A              CMP   AL,7AH
000E     77 02              JA    NEXT
0010     24 DF              AND   AL,0DFH
0012     88 04     NEXT:    MOV   [ SI] ,AL
```

Opcode: 75

Displacement: FA

IP+Disp = 0013+FA = 0D

Unconditional Jumps



Opcode

E B	Disp	Short

Opcode

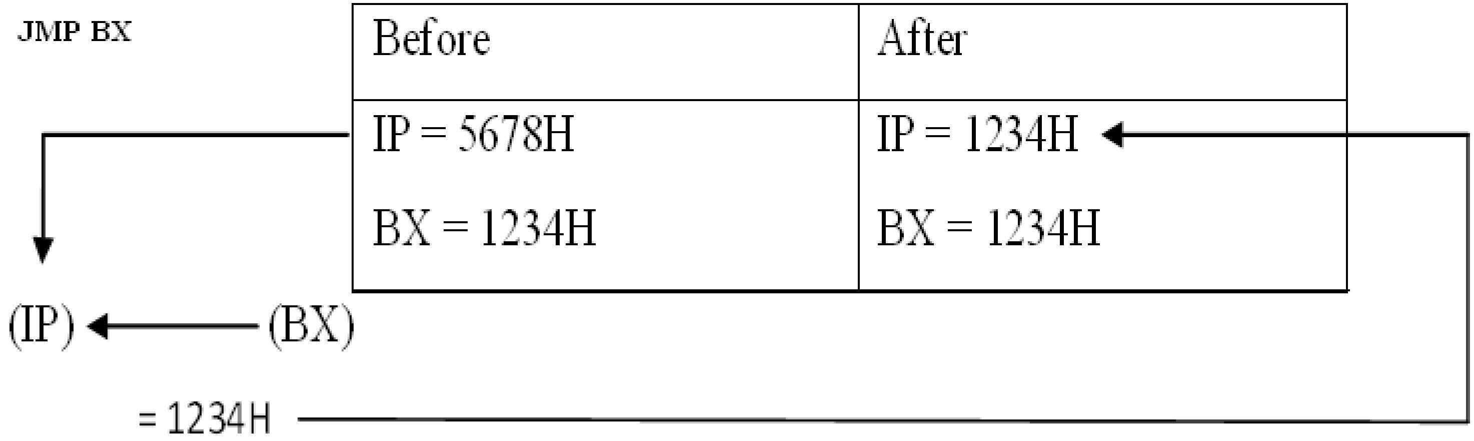
E 9	Disp Low	Disp High	Near

Opcode

E A	IP Low	IP High	CS Low	CS High	Far

- SHORT: -128 to +127
- NEAR: -32,768 to +32,767
- FAR:

JMP BX



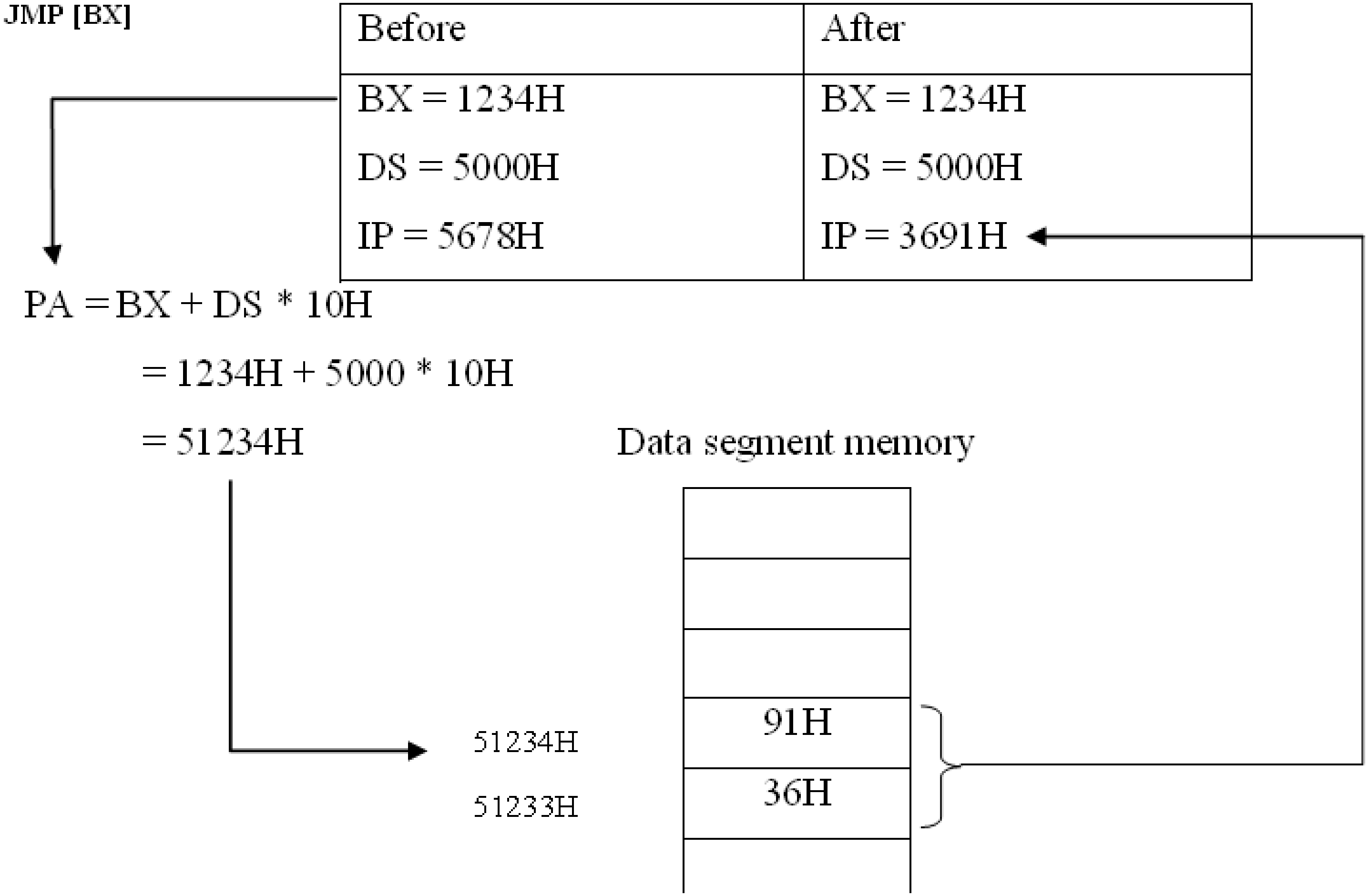
Before	After

IP = 5678H
BX = 1234H
	IP = 1234H
	BX = 1234H

(IP) ← (BX)

= 1234H

JMP [BX]



Before	After
BX = 1234H	BX = 1234H
DS = 5000H	DS = 5000H
IP = 5678H	IP = 3691H

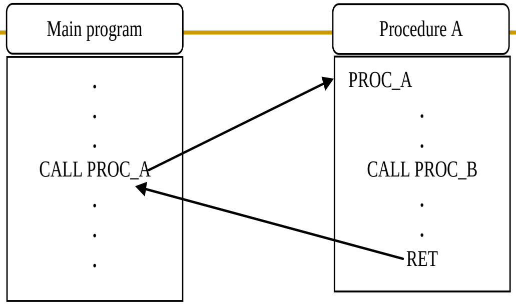
PA = BX + DS * 10H

= 1234H + 5000 * 10H

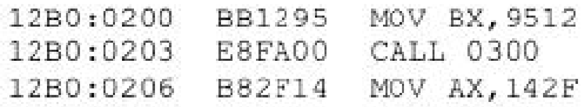
= 51234H

Data segment memory

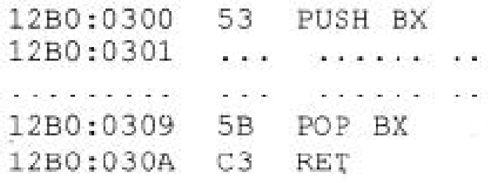
CALL Statement



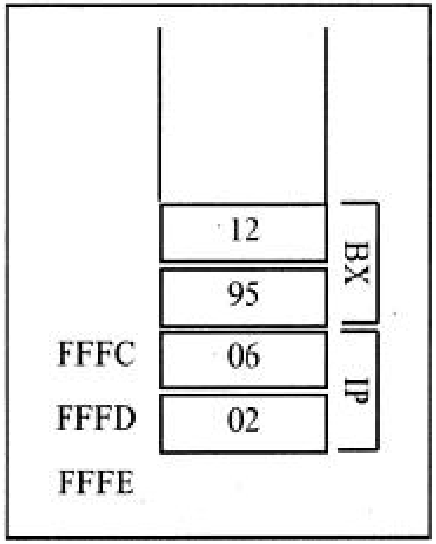
- Used to call a procedure
  - NEAR CALL – target address in the current segment
  - FAR CALL – target address outside the current CS
- Microprocessor automatically saves the address of the instruction following the call on the stack



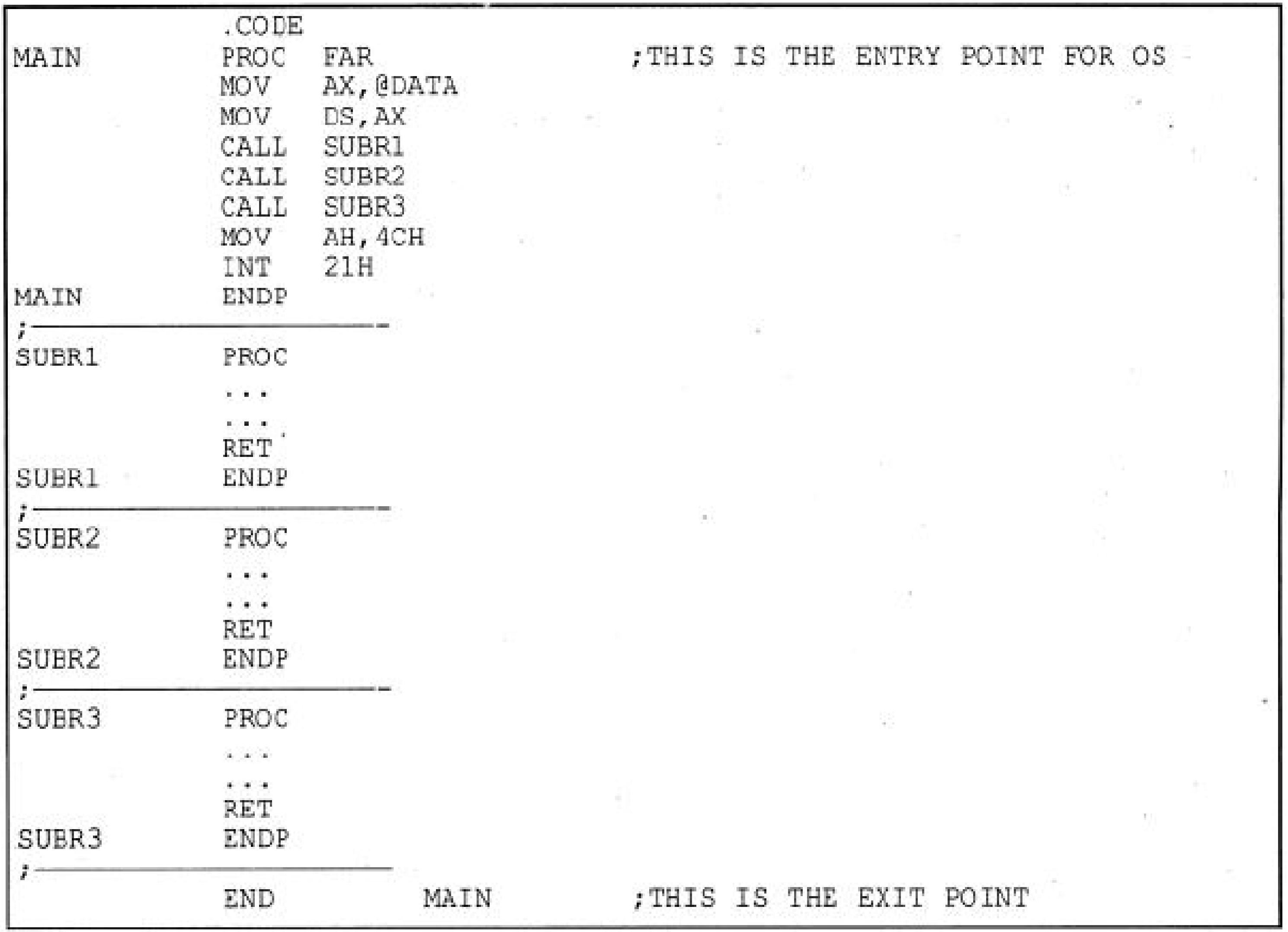
```
12B0:0200   BB1295   MOV  BX,9512
12B0:0203   E8FA00   CALL  0300
12B0:0206   B82F14   MOV  AX,142F
```



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



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
```

REVIEW

1. Briefly describe the functions of CALL and RET instruction

2. State why the following label names are invalid:

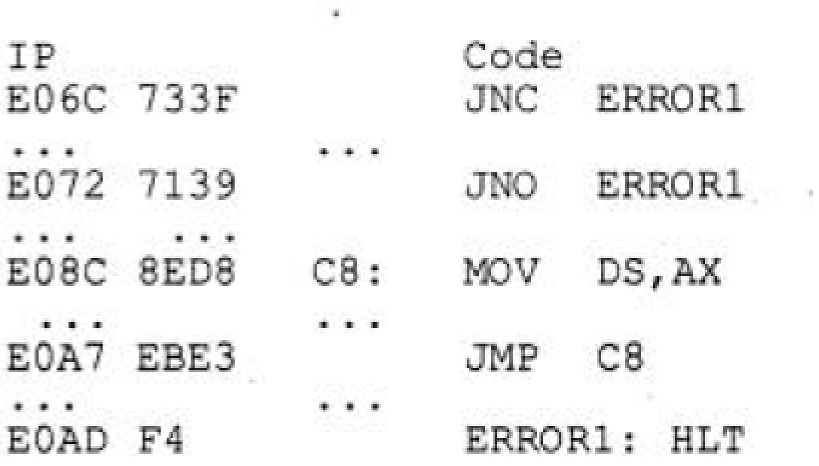
(a) GET.DATA (b) 1_NUM (c) TEST-DATA (d) RET

3. In the following code section, verify the address calculations of:

   (a) JNC ERROR1

   (b) JNO ERROR1

   (c) JMP C8

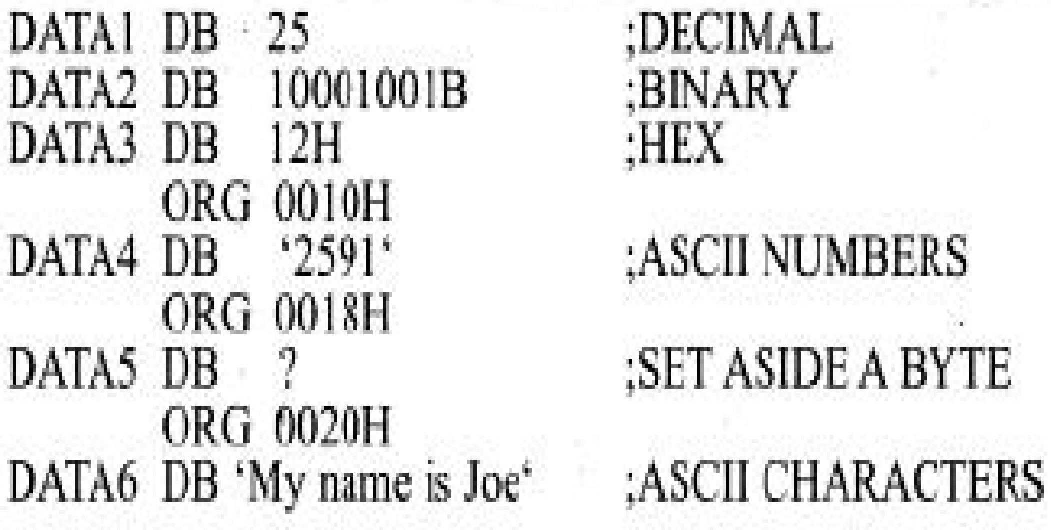


```
IP                          Code
E06C  733F                  JNC     ERROR1
...                 ...
E072  7139                  JNO     ERROR1
...   ...
E08C  8ED8          C8:     MOV     DS,AX
 ...                ...
E0A7  EBE3                  JMP     C8
...                 ...
E0AD  F4      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

DATA TYPES AND DATA DEFINITIONS

- The data types used by the 8088/86 can be 8-bit or 16-bit, positive or negative.
  - If a number is less than 8 bits wide, it still must be coded as an 8-bit register with the higher digits as zero
    - 5 is only 3 bits wide (101) in binary, but the 8088/86 will accept it as 05 or "0000 0101" in binary
  - if the number is less than 16 bits wide it must use all 16 bits, with the rest being 0s
    - 514 is "10 0000 0010" in binary, but the 8088/86 will accept it as "0000 0010 0000 0010" in binary

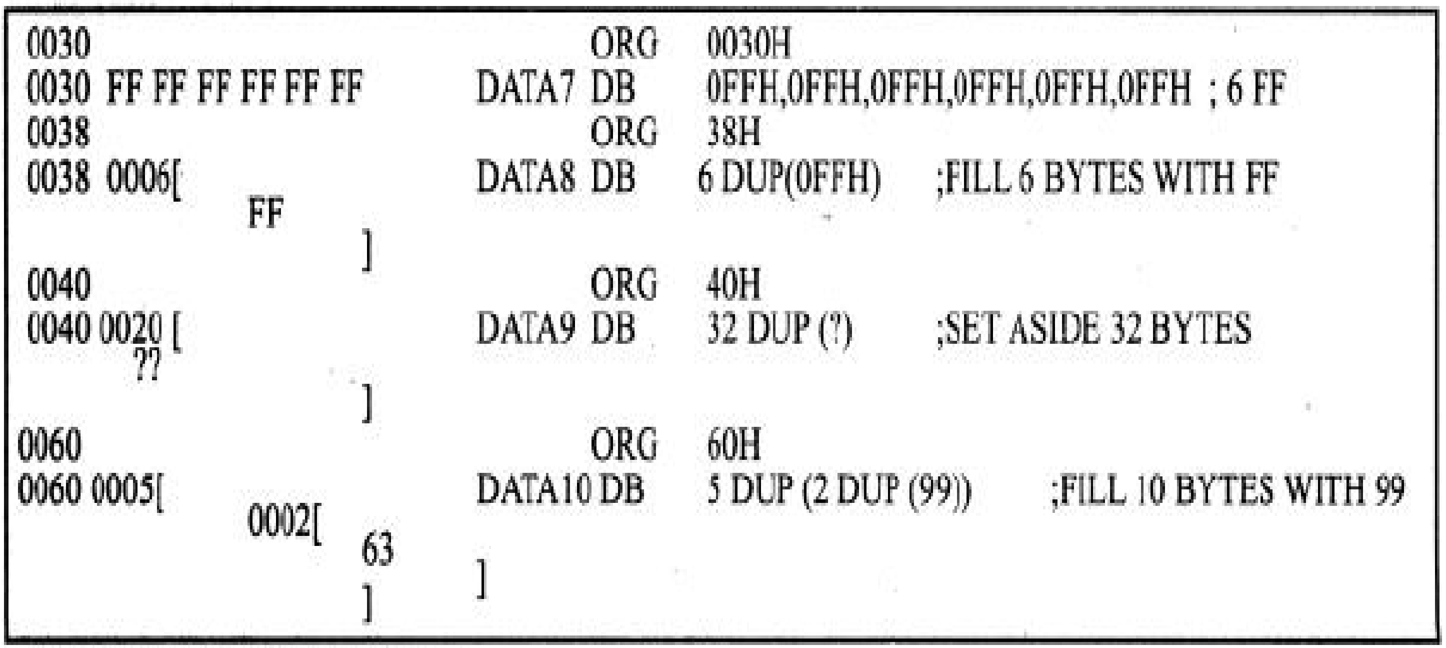
- **ORG** (origin) – used to indicate the beginning of the offset address
  - The number that comes after ORG can be either in hex or in decimal.
- **DB** (define byte) – directive allows allocation of memory in byte-sized chunks.
  - DB can be used to define numbers in decimal (D), binary (B), hex (H), and ASCII (‘quotation mark')



```
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
```

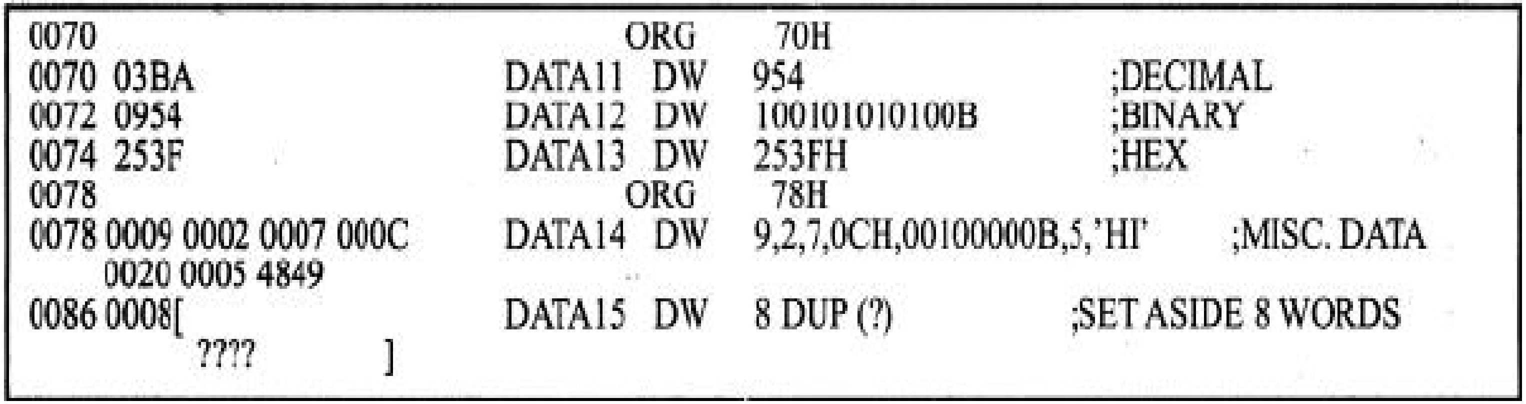
» **DUP (duplicate)** – used to duplicate a given number of characters.

- » This can avoid a lot of typing. For example, contrast the following two methods of filling six memory locations with FFH



```
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
                    ]    ]
```

- **DW (define word)** – used to allocate memory 2 bytes (one word) at a time. The following are some examples of DW



```
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
          ????         ]
```

- **EQU (equate)** – used to define a constant without occupying a memory location.
  - EQU does not set aside storage for a data item
  - EQU associates a constant value with a data label, so that when the label appears in the program, its constant value will be substituted
  - EQU can also be used outside the data segment, even in the middle of a code segment

- Using EQU for the counter constant in the immediate addressing mode:

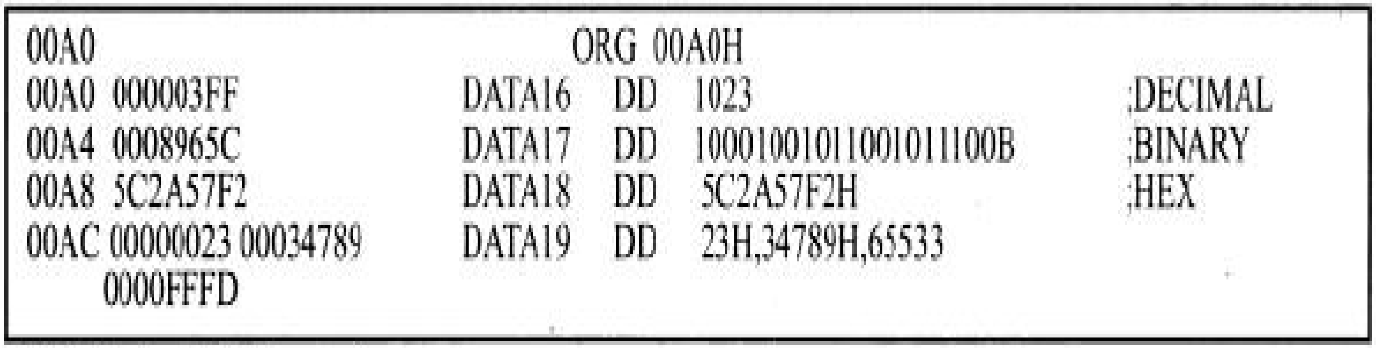
COUNT EQU 25	*COUNT DB 25*
When executing the instructions "MOV CX, COUNT", the register CX will be loaded with the value 25.	When executing the same instruction "MOV CX, COUNT" it will be in the direct addressing mode.

- Advantage of EQU?

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

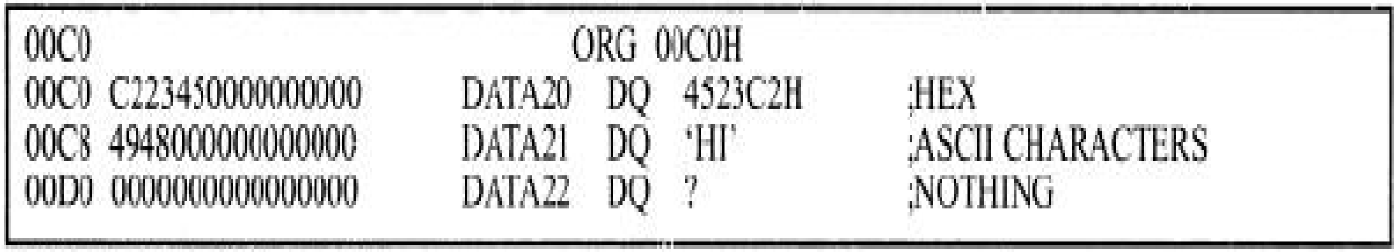
  - Assume that there is a constant (a fixed value) used in many different places in the data and code segments.
  - By the use of EQU, one can change it once and the assembler will change all of them, rather than making the programmer tries to find every location and correct it

» **DD (define double word)** – used to allocate memory locations that are 4 bytes (two words) in size.



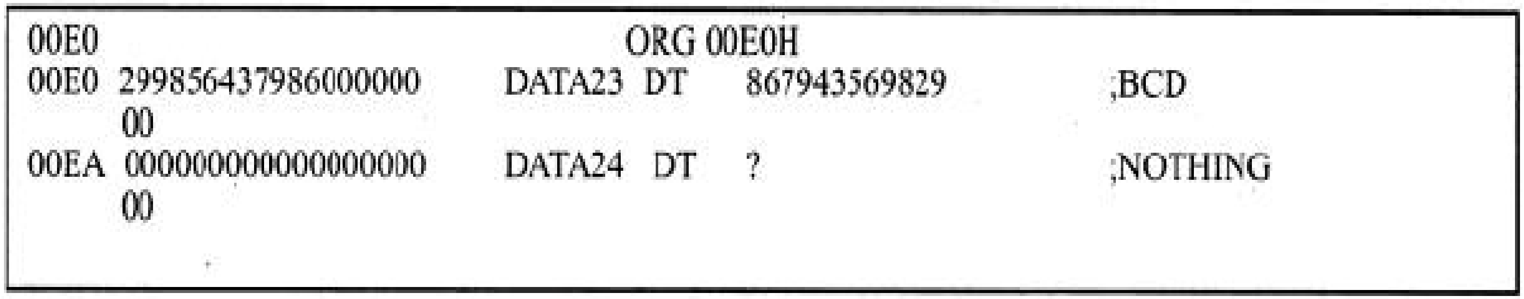
```
00A0                              ORG  00A0H
00A0  000003FF              DATA16    DD    1023                      ;DECIMAL
00A4  0008965C              DATA17    DD    10001001011001011100B     ;BINARY
00A8  5C2A57F2              DATA18    DD    5C2A57F2H                 ;HEX
00AC 00000023 00034789      DATA19    DD    23H,34789H,65533
      0000FFFD
```

» **DQ (define quad word)** – used to allocate memory 8 bytes (four words) in size.



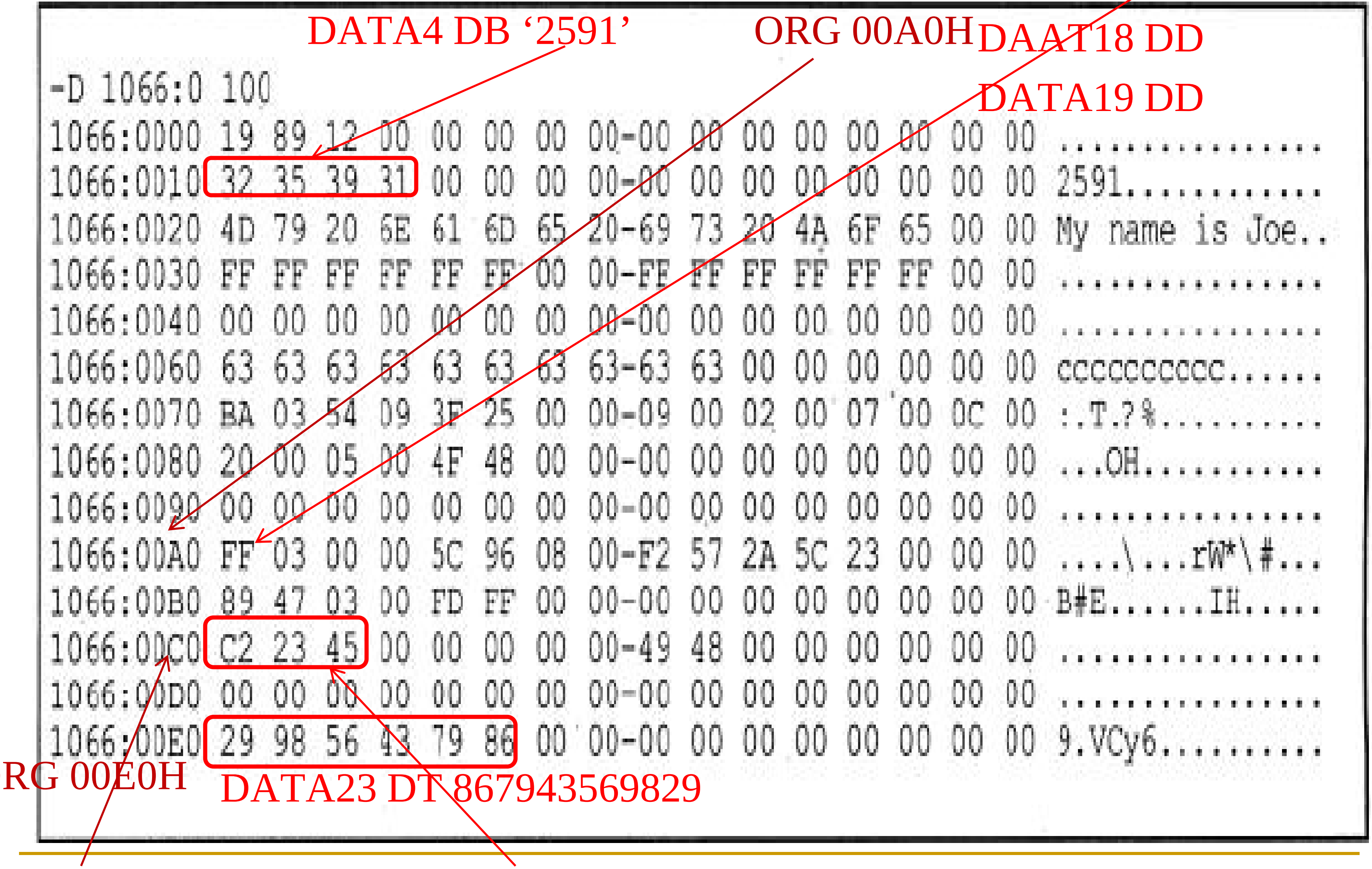
```
00C0                              ORG  00C0H
00C0  C223450000000000      DATA20   DQ   4523C2H       ;HEX
00C8  4948000000000000      DATA21   DQ   'HI'          ;ASCII CHARACTERS
00D0  0000000000000000      DATA22   DQ   ?             ;NOTHING
```

- **DT (define ten bytes)** – is used for memory allocation of packed BCD numbers (multibyte addition)
  - This directive allocates 10 bytes, but a maximum of 18 digits can be entered



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

» Memory dump of the data section



```
-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-FE 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..........
```

DATA16 DD 1023

DATA17 DD 10001…B

DAAT18 DD

DATA19 DD

DATA4 DB ‘2591’

ORG 00A0H

ORG 00E0H

DATA23 DT 867943569829

ORG 00C0H

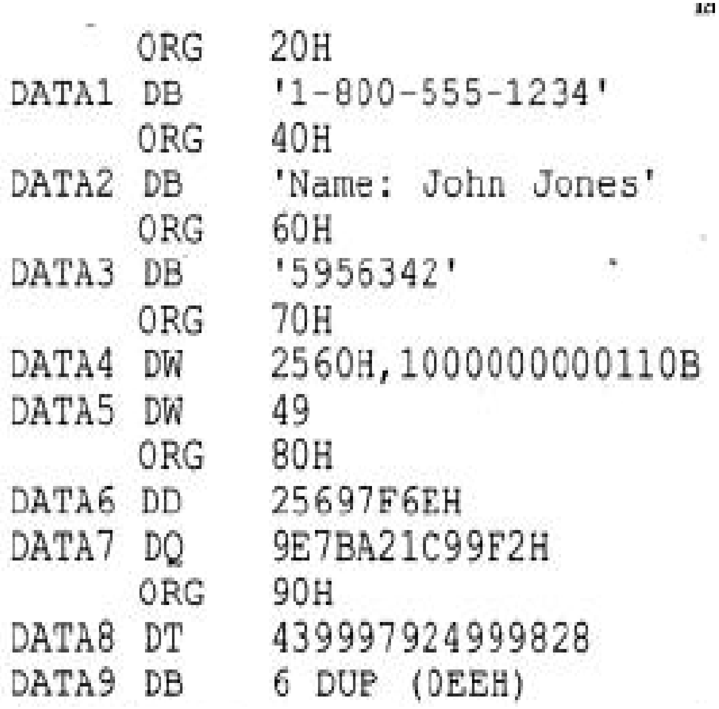
DATA20 DQ 4523C2H

REVIEW

1. Briefly state the purpose of ORG directive
2. What is advantage of using the EQU directive to define a constant value?
3. How many bytes are set aside by:

   (a) ASC_DATA DB ‘1234’

   (b) HEX_DATA DW 1234H
4. Find the precise offset location of each ASCII character or data in the following:



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

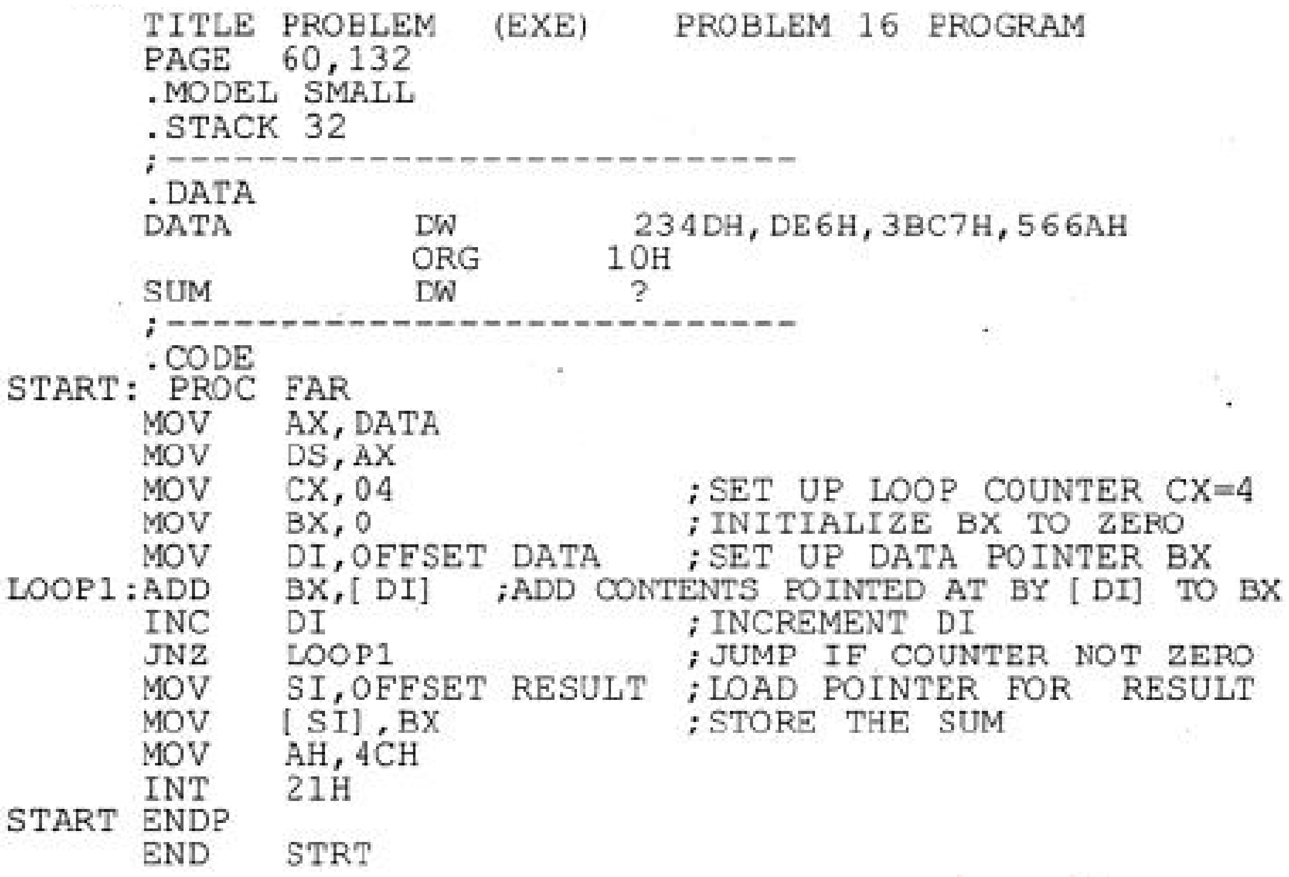
REVIEW

5. Do the following two data segment definitions result in same storage in bytes at offset 10H and 11H? If not, explain why

```
          ORG 10H                               ORG 10H
    DATA1 DB 72                           DATA1 DW 7204H
    DATA2 DB 04H
```

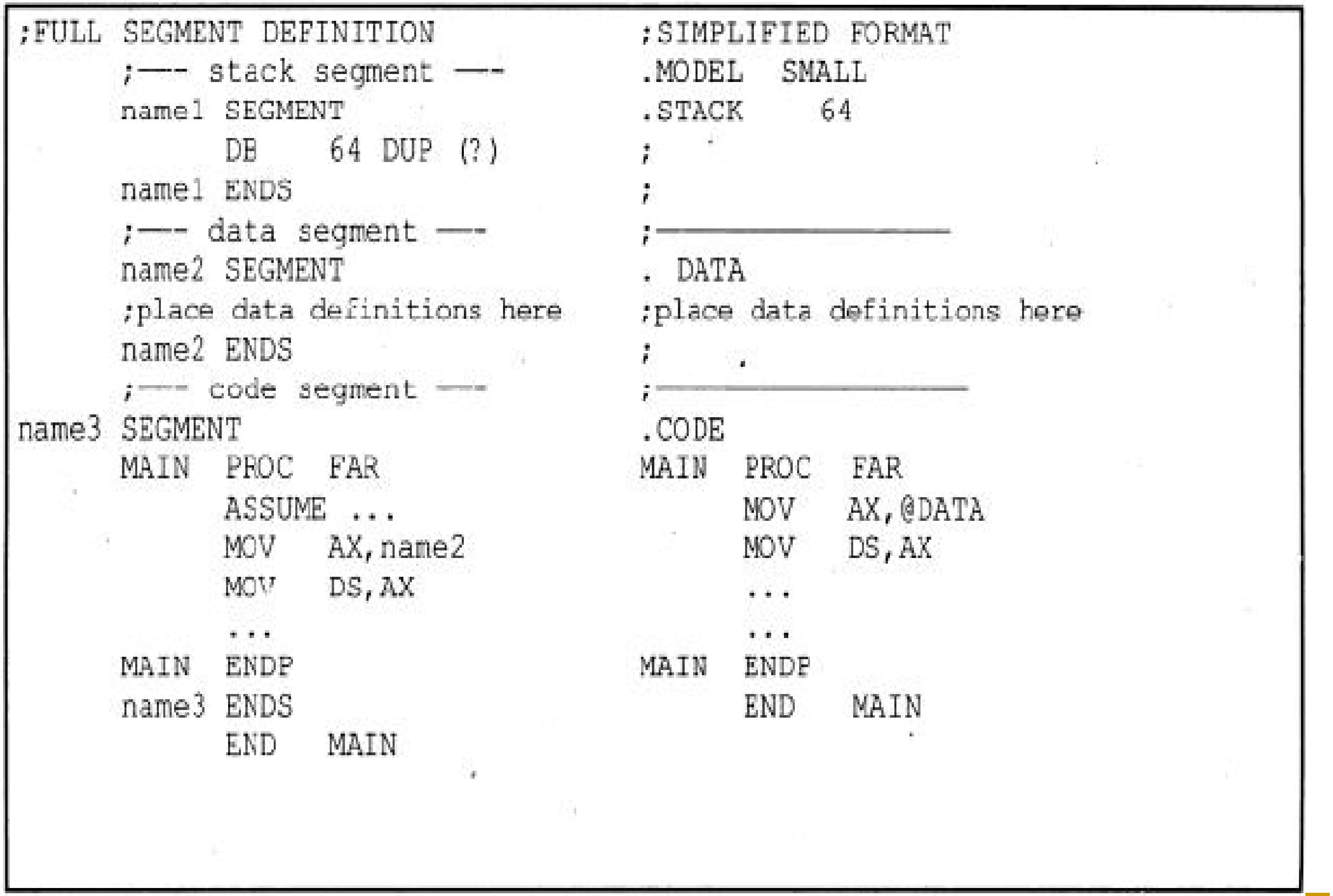
REVIEW

6. The following program contains some errors. Fix the errors and run the program correctly.



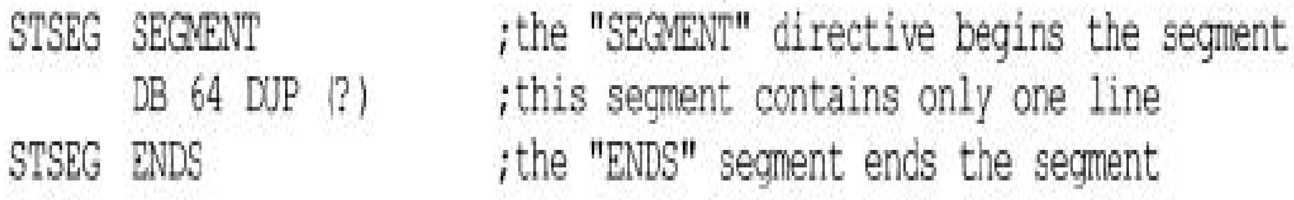
```
          TITLE PROBLEM   (EXE)    PROBLEM 16 PROGRAM
          PAGE   60,132
          .MODEL SMALL
          .STACK 32
          ;-----------------------------
          .DATA
          DATA         DW       234DH,DE6H,3BC7H,566AH
                       ORG     10H
          SUM          DW        ?
          ;-----------------------------
          .CODE
START:     PROC  FAR
          MOV    AX,DATA
          MOV    DS,AX
          MOV    CX,04                ;SET UP LOOP COUNTER CX=4
          MOV    BX,0                 ;INITIALIZE BX TO ZERO
          MOV    DI,OFFSET DATA       ;SET UP DATA POINTER BX
LOOP1:ADD    BX,[ DI]   ;ADD CONTENTS POINTED AT BY [ DI]  TO BX
          INC    DI                   ;INCREMENT DI
          JNZ    LOOP1                ;JUMP IF COUNTER NOT ZERO
          MOV    SI,OFFSET RESULT     ;LOAD POINTER FOR  RESULT
          MOV    [ SI] ,BX            ;STORE THE SUM
          MOV    AH,4CH
          INT    21H
START ENDP
          END    STRT
```

FULLSEGMENT DEFINITION



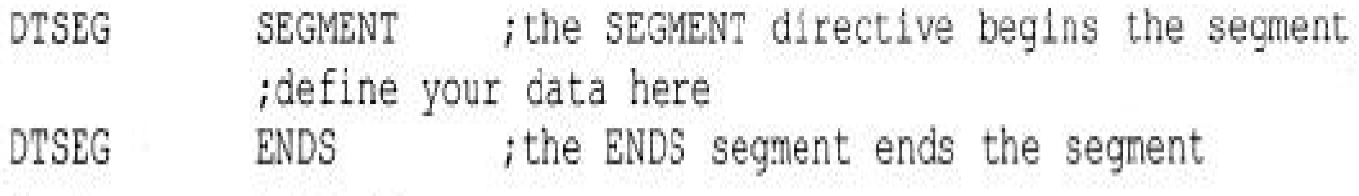
```
;FULL SEGMENT DEFINITION              ;SIMPLIFIED FORMAT
      ;--- stack segment ---          .MODEL   SMALL
      name1 SEGMENT                   .STACK     64
            DB    64 DUP (?)          ;
      name1 ENDS                      ;
      ;--- data segment ---           ;-----------------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

Stack Segment Definition



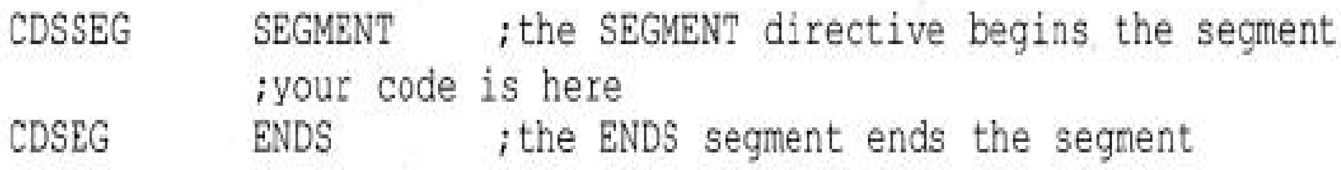
```
STSEG  SEGMENT              ;the "SEGMENT" directive begins the segment
       DB 64 DUP (?)        ;this segment contains only one line
STSEG  ENDS                 ;the "ENDS" segment ends the segment
```

Data Segment Definition

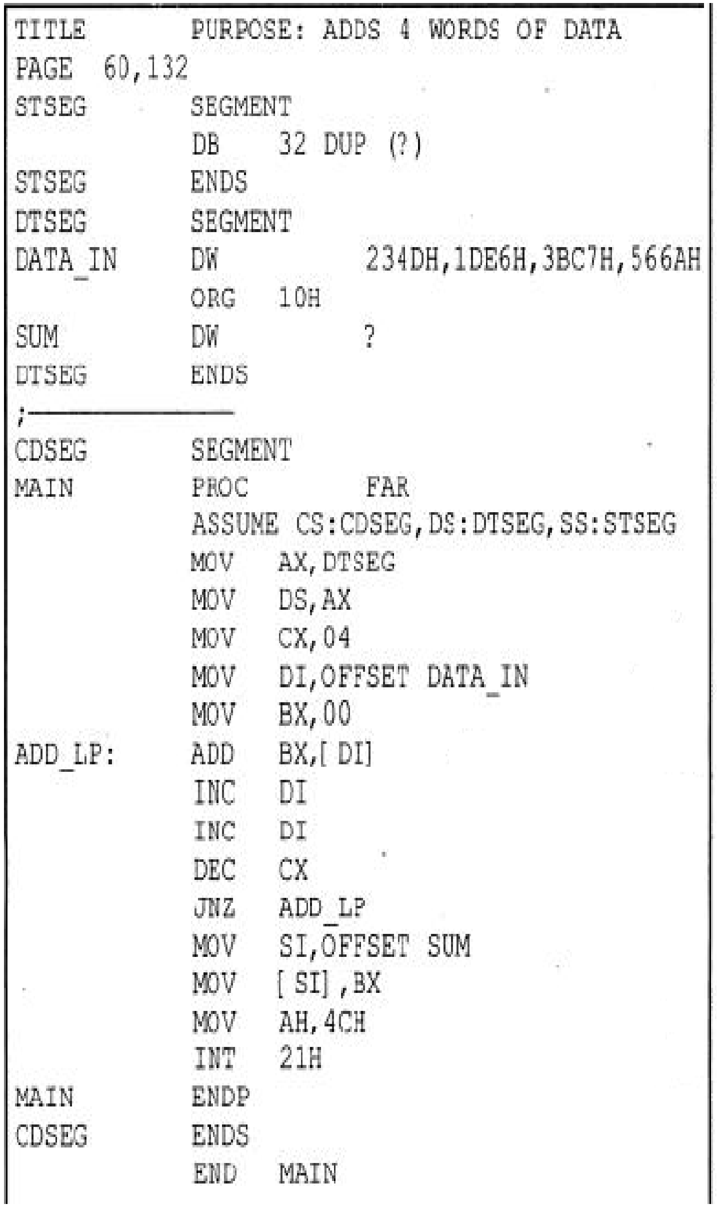


```
DTSEG       SEGMENT     ;the SEGMENT directive begins the segment
            ;define your data here
DTSEG       ENDS        ;the ENDS segment ends the segment
```

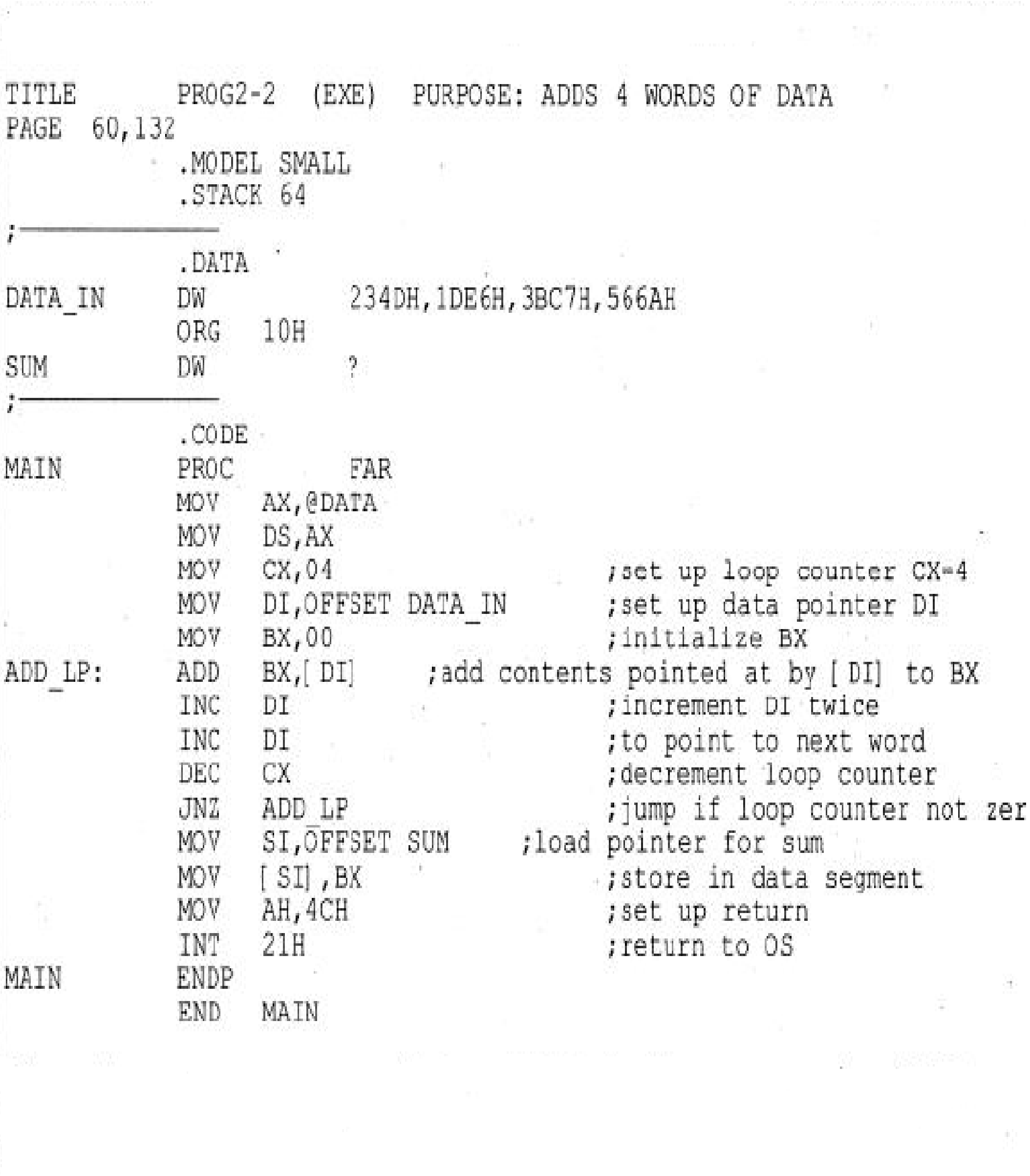
Stack Segment Definition



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



```
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
              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
MAIN          ENDP
CDSEG         ENDS
              END   MAIN
```



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

EXE vs COM Files

- The COM file, similar to the EXE file, contains the executable machine code and can be run at the OS level
- The EXE file can be of any size. The COM files are used because of their compactness, since they cannot be greater than 64K bytes
  - The COM file must fit into a single segment, and since in the x86 the size of a segment is 64K bytes, the COM file cannot be larger than 64K
- To limit the size of the file to 64K bytes requires
  - defining the data inside the code segment and
  - also using an area (the end area) of the code segment for the stack

EXE File	**COM File**
1. Unlimited size	1. Maximum size 64K bytes
2. Stack segment is defined	2. No stack segment definition
3. Data segment is defined	3. Data segment is defined in code segment
4. Larger file (takes more memory)	4. Smaller file (takes less memory)
5. Header block (contains information such as size, address location in memory, and stack address of the EXE module), which occupies 512 bytes of memory precedes every EXE file	5. Does not have a header file

FLOWCHARTS AND PSEUDOCODE

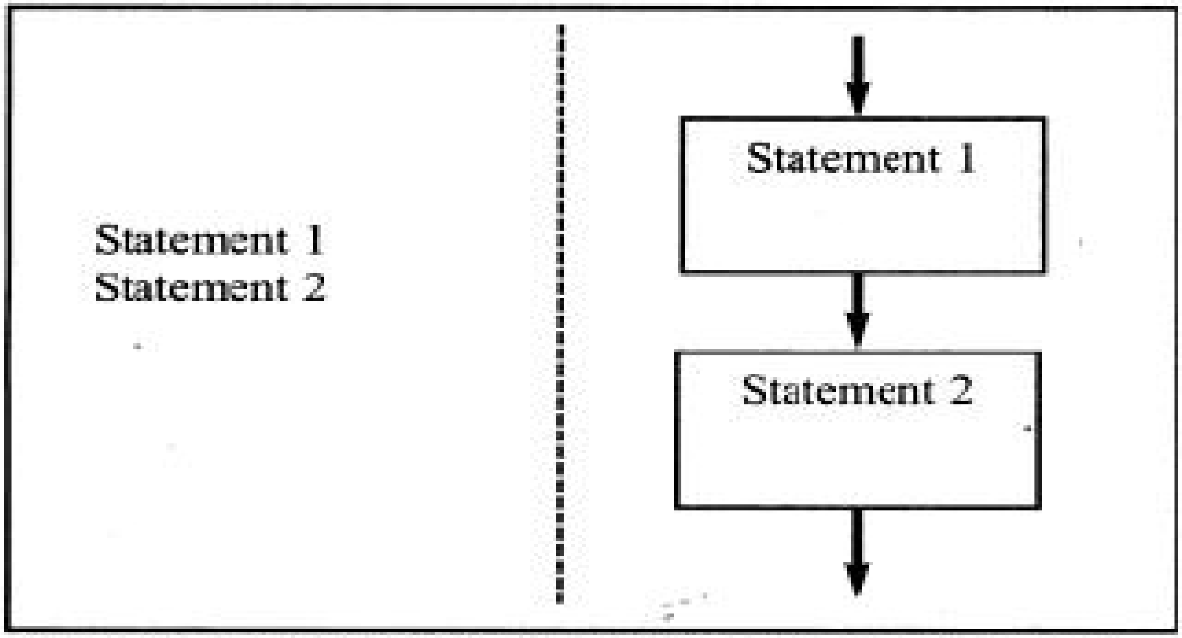
» Structured programming – a programming technique that can make a program easier to code, debug, and maintain over time. Principles:

1. The program should be designed before it is coded, by using techniques of flowcharting or pseudocode
2. Using comments within the program and documentation accompanying the program
3. The main routine should consist of calls to subroutines that perform the work of the program. This is sometimes called top-down programming.
4. Data control is very important. The programmer should document the purpose of each variable, and which subroutines might alter its value.
5. Each subroutine should document its input and output variables, and which input variables might be altered within it.

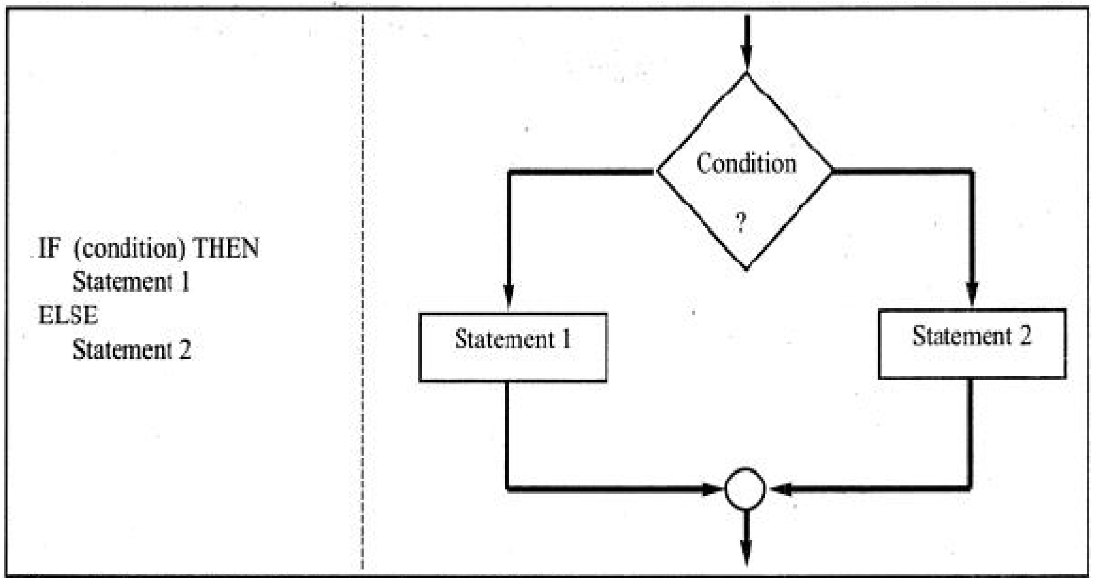
Flowcharts & Pseudocode

- Flowcharts use graphic symbols to represent different types of program operations.
- These symbols are connected together into a flowchart to show the flow of execution of the program
- The limitations of flowchart are –
    - We can’t write much in the little boxes
    - We can’t get the clear picture of the program without getting bogged down in the details.
- An alternative to using flowchart is pseudocode, which involves writing brief descriptions of the flow of the code

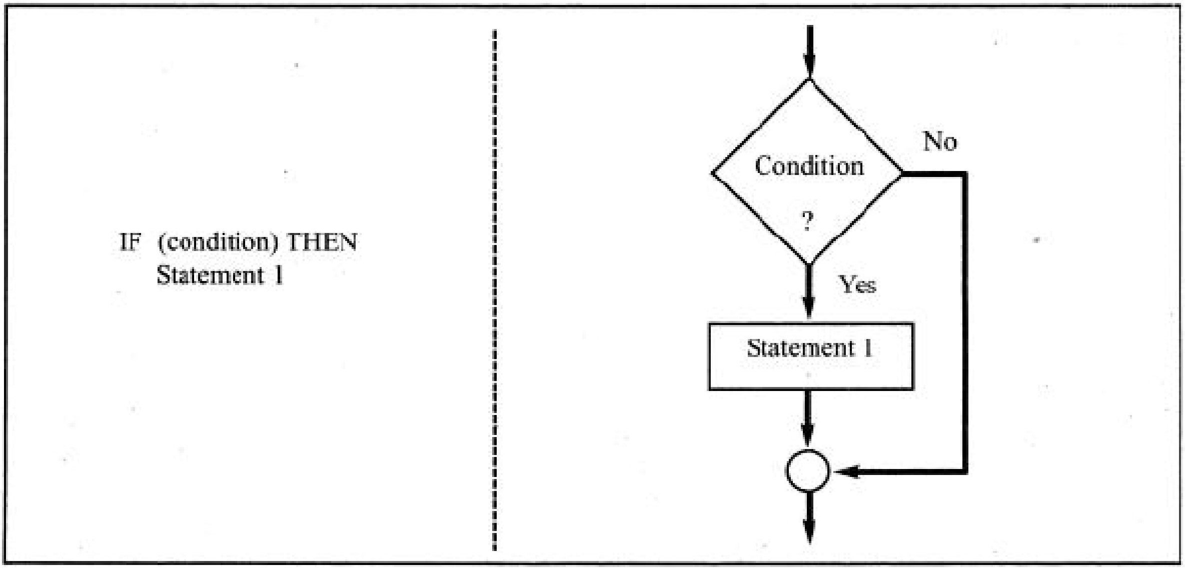
Control Structure: Sequence



Control Structure: Control

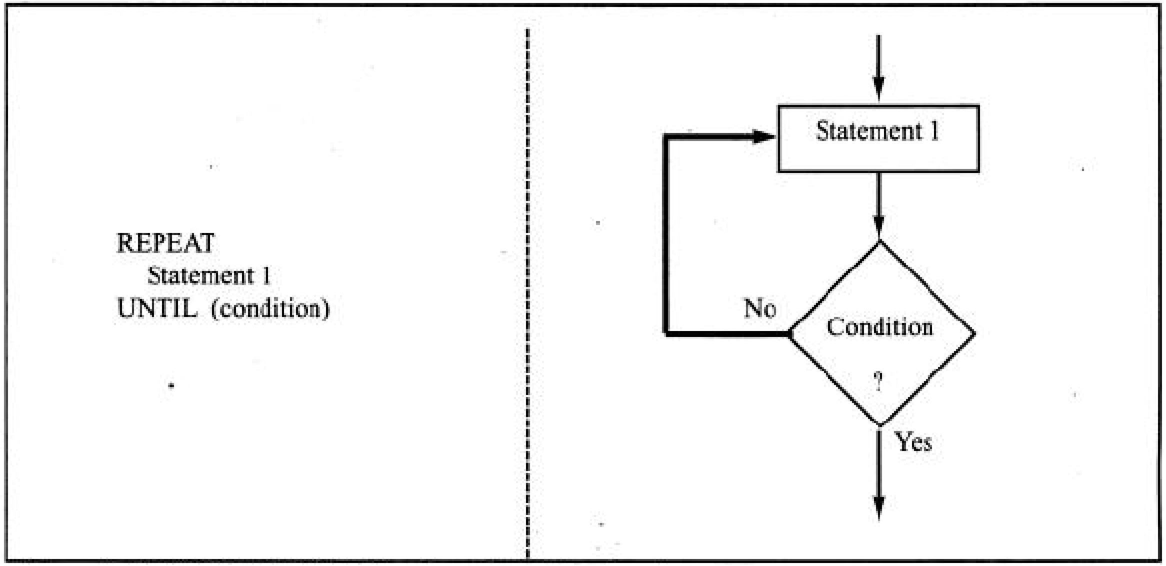


```
IF (condition) THEN
    Statement 1
ELSE
    Statement 2
```

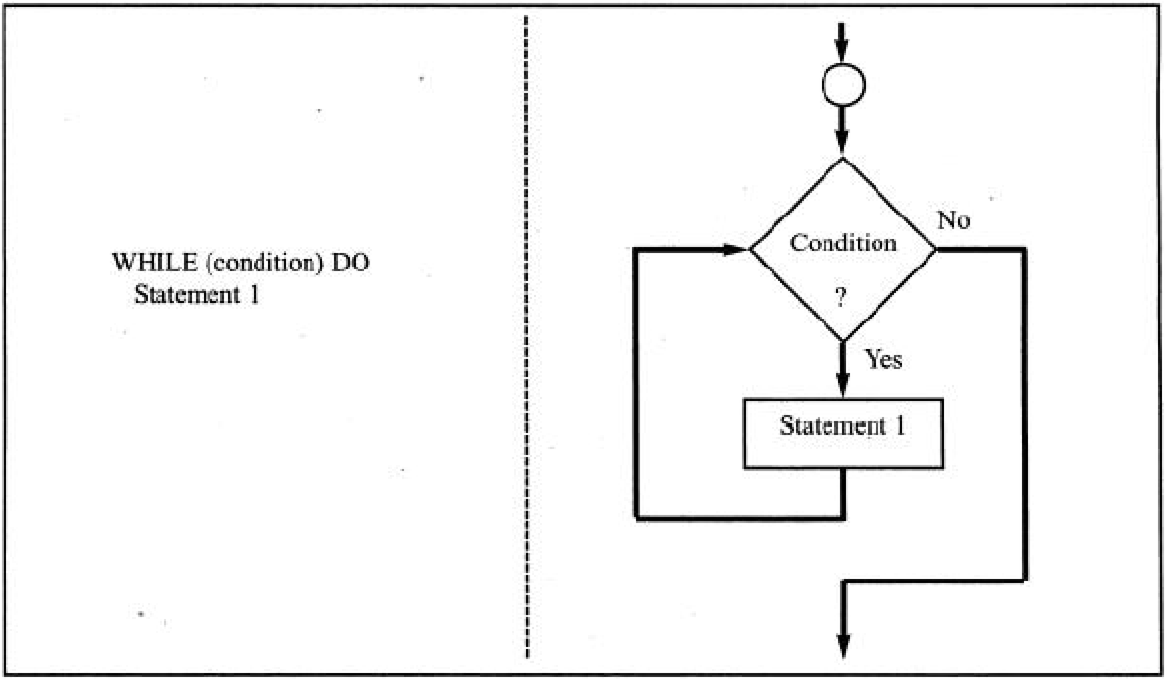


```
IF (condition) THEN
    Statement 1
```

Control Structure: Iteration

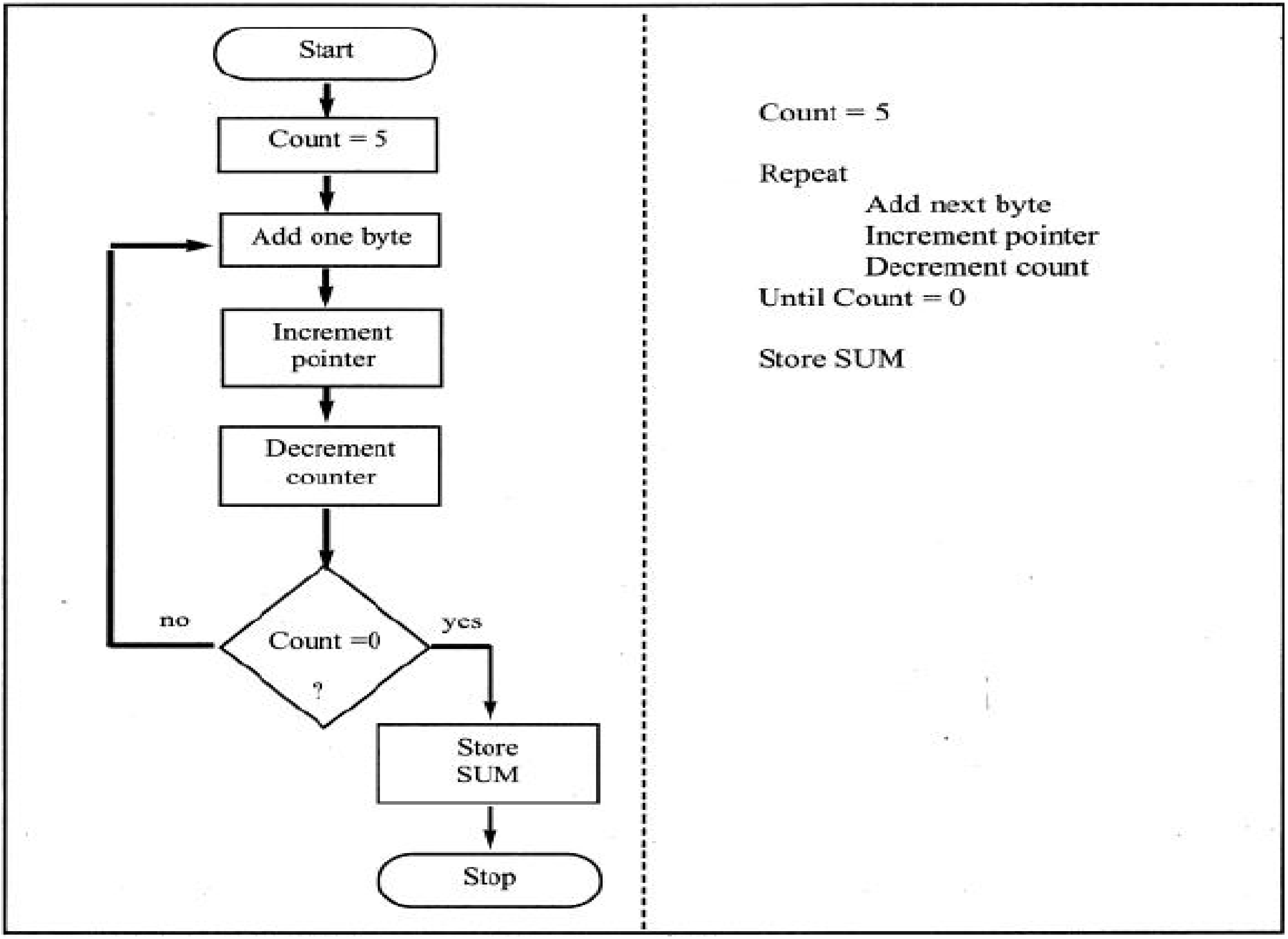


```
REPEAT
   Statement 1
UNTIL  (condition)
```



```
WHILE (condition) DO
   Statement 1
```

Flowcharts vs Pseudocode for Program 2-1



```
Count = 5

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

Store SUM
```

15CS – 44

MICROPROCESSORS AND MICROCONTROLLERS

MODULE 1 – QUIZ 2

THE x86 MICROPROCESSOR

Mahesh Prasanna K.

Dept. of CSE, VCET.

1. The _____ are translated by the assembler into machine code, whereas the _____ are not
2. The input file to the MASM assembler program has the extension _____
3. The input file to the LINK program has the extension _____
4. The linking process comes after assembling (TRUE/FALSE)
5. In calculating the target address to jump to, a displacement is added to the contents of _____

6. A(n) ______ jump is within -128 to +127 bytes of the current IP

7. A(n) ______ jump is within –current code segment

8. A(n) ______ jump is within outside the current code segment

9. In a FAR CALL ______ and ______ are saved on the stack

10. The ______ directive is always used for the ASCII strings longer than 2 bytes

11. The DD directive is used to allocate memory locations that are _____ bytes in length; the DQ directive is used to allocate memory locations that are _____ bytes in length

12. The ASSUME directive is used in full segment definition (TRUE/FALSE)

13. In full segment definition, each segment begins with the ______ directive and ends with a matching ______ directive

1. The ______ are translated by the assembler into machine code, whereas the ______ are not (instructions, pseudo-instructions or directives)
2. The input file to the MASM assembler program has the extension ______ (.asm)
3. The input file to the LINK program has the extension ______ (.obj)
4. The linking process comes after assembling (TRUE/FALSE)
5. In calculating the target address to jump to, a displacement is added to the contents of ______ (IP)

6. A(n) ______ jump is within -128 to +127 bytes of the current IP (SHORT)

7. A(n) ______ jump is within –current code segment (NEAR)

8. A(n) ______ jump is within outside the current code segment (FAR)

9. In a FAR CALL ______ and ______ are saved on the stack (IP, CS)

10. The ______ directive is always used for the ASCII strings longer than 2 bytes (DB)

11. The DD directive is used to allocate memory locations that are _____ bytes in length; the DQ directive is used to allocate memory locations that are _____ bytes in length (4, 8)

12. The ASSUME directive is used in full segment definition (TRUE/FALSE)

13. In full segment definition, each segment begins with the ______ directive and ends with a matching ______ directive (SEGMENT, ENDS)