

# **CPE 408330**

# **Assembly Language and**

# **Microprocessors**

[Computer Engineering Department,  
Hashemite University,]

By Dr. Awni Itradat

# **Chapter 1: Introduction to Microprocessors & Microcomputers**

# Microprocessor vs. Microcomputer

- ▶ A microprocessor is a central processing unit (CPU) on a single chip and is entirely useless on its own.
- ▶ A microcomputer is a *stand-alone system*\* based on
  - Microprocessor
  - Memory components
  - Interface components
  - Timing and control circuits
  - Power supply
  - An enclosure (e.g. a cabinet or package)

\**Stand-alone system* : A system that is able to operate independently

# Microprocessor

- ▶ A silicon chip that contains a CPU. In the world of personal computers, the terms microprocessor and CPU are used interchangeably. At the heart of all personal computers and most workstations sits a microprocessor. Microprocessors also control the logic of almost all digital devices, from clock radios to fuel-injection systems for automobiles.



# Microcomputer Categories

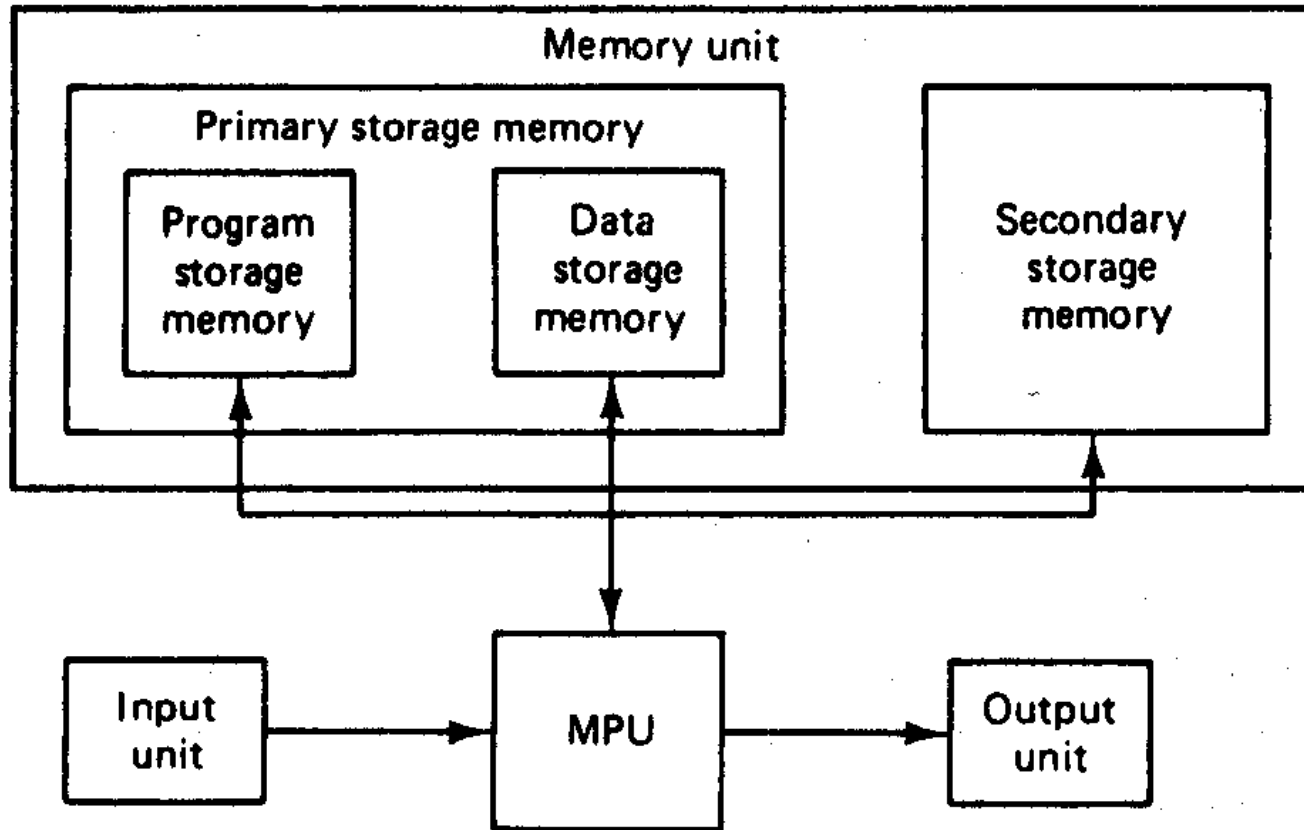
The microcomputer falls into 2 categories

1. The General-Purpose Digital Computer
2. The Embedded Computer
  - Dedicated to specific applications
  - Transparent (“invisible”) to the user. (eg. Automatic Bank Teller machine)

# 1.1 The IBM and IBM-Compatible Personal Computers (PCs).

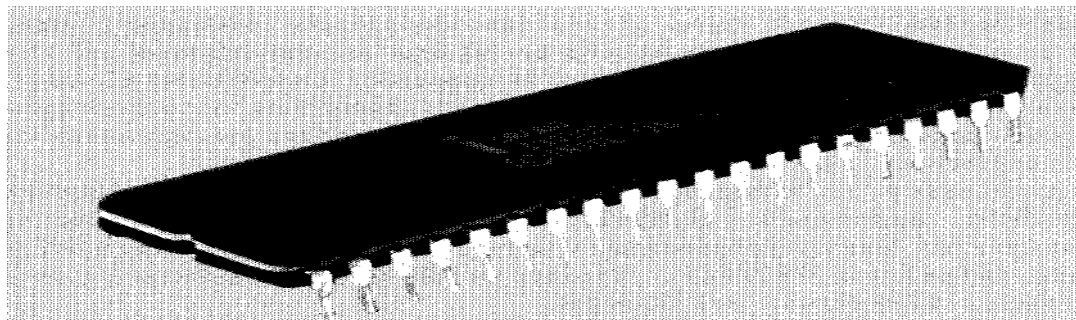
- ❑ Most important advances in computer technology: 16-bit and 32-bit microprocessors.
- ❑ Pioneered by Intel since 1970's and dominated by INTEL since 1980's:
  - 4-bit 4004 in 1971
  - 8-bit 8008 in 1972
  - 8-bit 8080 and 8085 in 1974
  - 16-bit 80286 and 8086, brains of famous IBM PC
  - 32-bit 80286 (1982), 80386 (1985), 80486 (1989), Pentium (1993), Pentium II (1997), Celeron and Pentium III (1999) and Pentium 4 (2000)
  - 64-bit Itanium (2001)
  - Latest 64-bit Pentium 4 and Xeon (2005)

# 1.2 General Architecture of a Microcomputer System



# 1.2 General Architecture of a Microcomputer System

- ❑ The 8088 and 8086 microprocessor:
  - 8088 – 8-bit external bus, 16-bit internal architecture.
  - 8086 – 16-bit external bus, 16-bit internal architecture.
- ❑ MPU performs arithmetic operation and logical decision



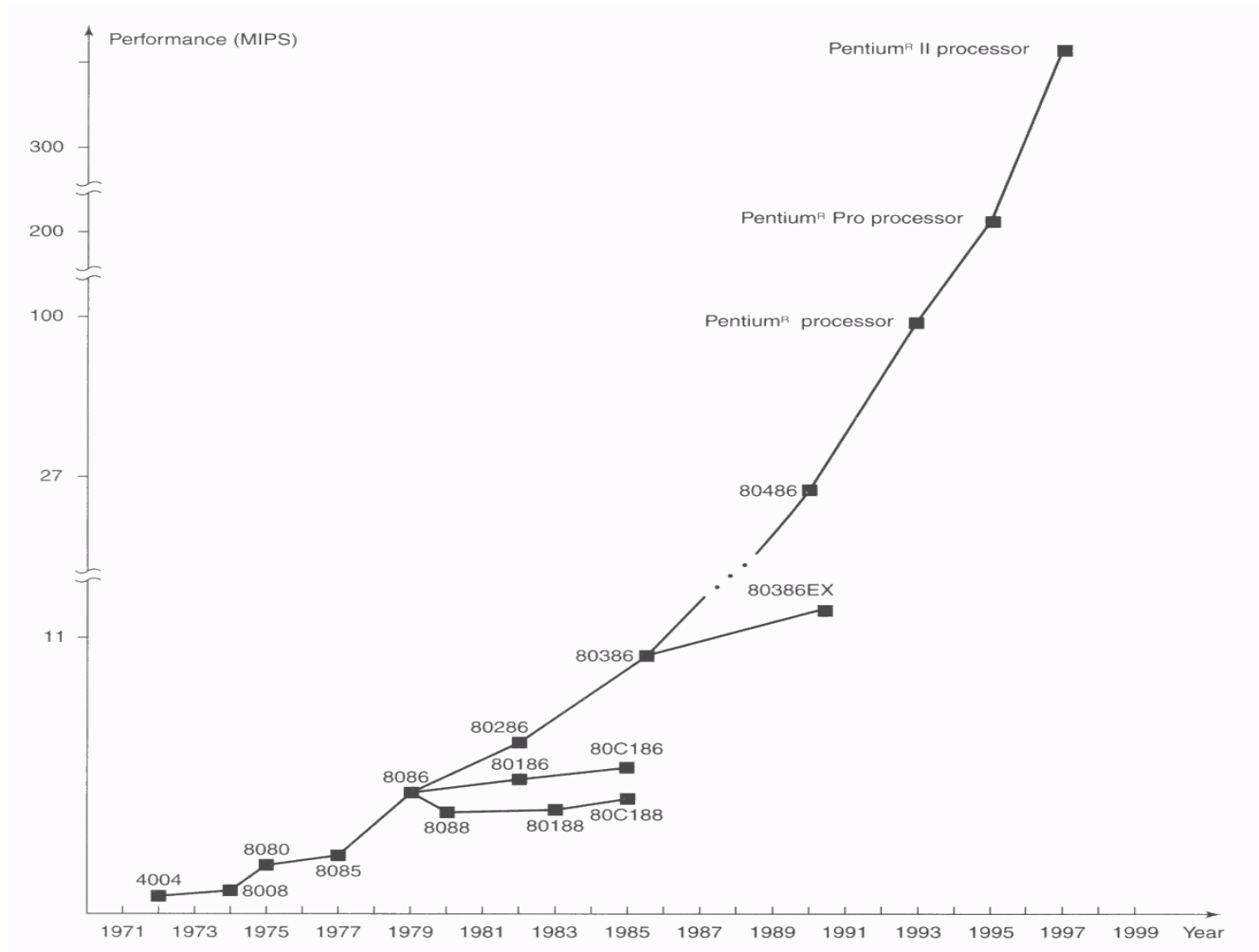
# 1.2 General Architecture of a Microcomputer System

- ❑ Input Unit:
  - Keyboard, joystick, mouse, scanner.
- ❑ Output Unit:
  - CRT display, LCD display, printer.
- ❑ Memory Unit:
  - Primary storage memory: ROM, RAM.
  - Secondary storage memory: floppy-diskette, hard disk drive, CD-ROM, CD-RW, magnetic tape

# 1.3 Evolution of the Intel Microprocessor Architecture

- ❑ 1971 Intel introduces its first microprocessor, the 4004, which contained 2250 transistors. The 4004 was designed to process data arranged as 4-bit words.
- ❑ Beginning in 1974, a second generation of microprocessors was introduced. These devices, the 8008, 8080, and 8085, were 8-bit microprocessors.

# 1.3 Evolution of the Intel Microprocessor Architecture



# 1.3 Evolution of the Intel Microprocessor Architecture

□ Moore's law is good for the last 26 years!

1971: 4004	2,250 transistors
1972: 8008	2,500 transistors
1974: 8080	5,000 transistors
1978: 8086	29,000 transistors
1982: 80286	120,000 transistors
1985: 80386	275,000 transistors
1989: 80486 DX	1,180,000 transistors
1993: Pentium	3,100,000 transistors
1997: Pentium II	7,500,000 transistors
1999: Pentium III	24,000,000 transistors
2000: Pentium IV	42,000,000 transistors
2006: Pentium D	376,000,000 transistors



# 1.4 Number Systems

## □ Decimal number system

- The number of symbols used is called the **base** or **radix** of the number system.
- **Most Significant Digit (MSD)** and **Least Significant Digit (LSD)**.

0							
1							
2							
3							
4							
5							
6							
7							
8							
9							

	MSD					LSD		
Weights	$10^{+3}$	$10^{+2}$	$10^{+1}$	$10^0$	.	$10^{-1}$	$10^{-2}$	$10^{-3}$
	1000	100	10	1	.	1/10	1/100	1/1000

Reference digit

(a) Decimal number system symbols. (b) Digit notation and weights.

# 1.4 Number Systems

## □ Binary number system

- $1100_2 = 1(2^{+3}) + 1(2^{+2}) + 0(2^{+1}) + 0(2^0)$   
 $= 1(8) + 1(4) + 0(2) + 0(1)$

- ▶  $= 12_{10}$

- ▶  $12_{10} = 00000000000001100_2$

		MSB					LSB		
	Weights	$2^{+3}$	$2^{+2}$	$2^{+1}$	$2^0$	.	$2^{-1}$	$2^{-2}$	$2^{-3}$
0		8	4	2	1	.	1/2	1/4	1/8
1									

Reference bit

(a) Binary number system symbols. (b) Bit notation and weights.

# 1.4 Number Systems

- Conversion between decimal and binary numbers

Decimal number	Binary number
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

# 1.4 Number Systems

## □ Example:

Evaluate the decimal equivalent of binary number  $101.01_2$ .

## □ Solution:

- $101.01_2 = 1(2^{+2}) + 0(2^{+1}) + 1(2^{+0}) + 0(2^{-1}) + 1(2^{-2})$   
 $= 1(4) + 0(2) + 1(1) + 0(1/2) + 1(1/4)$   
▶  $= 5.25_{10}$

# 1.4 Number Systems

- Example:

Convert the decimal number  $31_{10}$  to binary form. Also, express the answer as a byte-wide binary number.

- Solution:

$\underline{2}$	31	→	1	LSB
$\underline{2}$	15	→	1	
$\underline{2}$	7	→	1	
$\underline{2}$	3	→	1	
$\underline{2}$	1	→	1	MSB
	0			

$$31_{10} = 11111_2$$

# 1.4 Number Systems

- Example:

Convert the decimal fraction  $0.8125_{10}$  to binary form. Also, express the answer as a byte-wide binary number.

- Solution:

$2 * 0.8125$	$\rightarrow$	$1$	MSB
$2 * 0.625$	$\rightarrow$	$1$	
$2 * 0.25$	$\rightarrow$	$0$	
$2 * 0.5$	$\rightarrow$	$1$	
$2 * 0$			

$$0.8125_{10} = .1101_2$$

# 1.4 Number Systems

## □ Hexadecimal number system

- Machine language programs, addresses, and data are normally expressed as hexadecimal number.

0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
A  
B  
C  
D  
E  
F

	MSD				LSD		
Weights	$16^{+3}$	$16^{+2}$	$16^{+1}$	$16^0$	.	$16^{-1}$	$16^{-2}$
	4096	256	16	1	.	1/16	1/256

Reference digit

(a) Hexadecimal number system symbols. (b) Digit notation and weights.

# 1.4 Number Systems

## □ Example:

What the decimal number  $102A_{16}$  represent?

## □ Solution:

- $102A_{16} = 1(16^{+3}) + 0(16^{+2}) + 2(16^{+1}) + A(16^0)$   
 $= 1(4096) + 0(256) + 2(16) + A(1)$   
▶  $= 4138_{10}$



# 1.4 Number Systems

- Example:

Convert the decimal number  $4138_{10}$  to hexadecimal form.

- Solution:

<u>16</u>	<u>4138</u>		
16	258	→	A
16	16	→	2
16	1	→	0
	0	→	1

LSB

MSB

$$4138_{10} = 102A_{16}$$

# 1.4 Number Systems

- Conversion between hexadecimal and binary numbers.
  - An H is frequently used instead of a subscript 16 to denote that a value is a hexadecimal number.

Binary number	Hexadecimal number
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

MSB		LSB	
$2^{15}2^{14}2^{13}2^{12}$	$2^{11}2^{10}2^92^8$	$2^72^62^52^4$	$2^32^22^12^0$
$16^3$	$16^2$	$16^1$	$16^0$
MSD		LSD	

- (a) Equivalent binary and hexadecimal numbers.  
(b) Binary bits and hexadecimal digits.

# 1.4 Number Systems

- Conversion between decimal, binary, and hexadecimal numbers:

Decimal number	Binary number	Hexadecimal number
0	00000000	00
1	00000001	01
2	00000010	02
3	00000011	03
4	00000100	04
5	00000101	05
6	00000110	06
7	00000111	07
8	00001000	08
9	00001001	09
10	00001010	0A
11	00001011	0B
12	00001100	0C
13	00001101	0D
14	00001110	0E
15	00001111	0F

# 1.4 Number Systems

## □ Example:

Express the binary number  
 $1111100100001010_2$ .

## □ Solution:

- $1111100100001010_2 = 1111 \ 1001 \ 0000 \ 1010$   
= F 9 0 A  
=  $F90A_{16}$   
= F90AH

# 1.4 Number Systems

## □ Example:

What is the binary equivalent of the number  $C315_{16}$ ?

## □ Solution:

- $C315_{16} = 1100\ 0011\ 0001\ 0101$   
 $= 1100001100010101_2$



# **CPE 408330**

## **Assembly Language and Microprocessors**

### **Chapter 2: Software Architecture of the 8088 and 8086 Microcomputers**

[Computer Engineering Department,  
Hashemite University]

# Lecture Outline

- ▶ 2.1 Microarchitecture of the 8088/8086 Microprocessor
- ▶ 2.2 Software Model of the 8088/8086 Microprocessor
- ▶ 2.3 Memory Address Space and Data Organization
- ▶ 2.4 Data Types
- ▶ 2.5 Segment Registers and Memory Segmentation
- ▶ 2.6 Dedicated, Reserved, and General-Used Memory
- ▶ 2.7 Instruction Pointer
- ▶ 2.8 Data Registers

# Lecture Outline

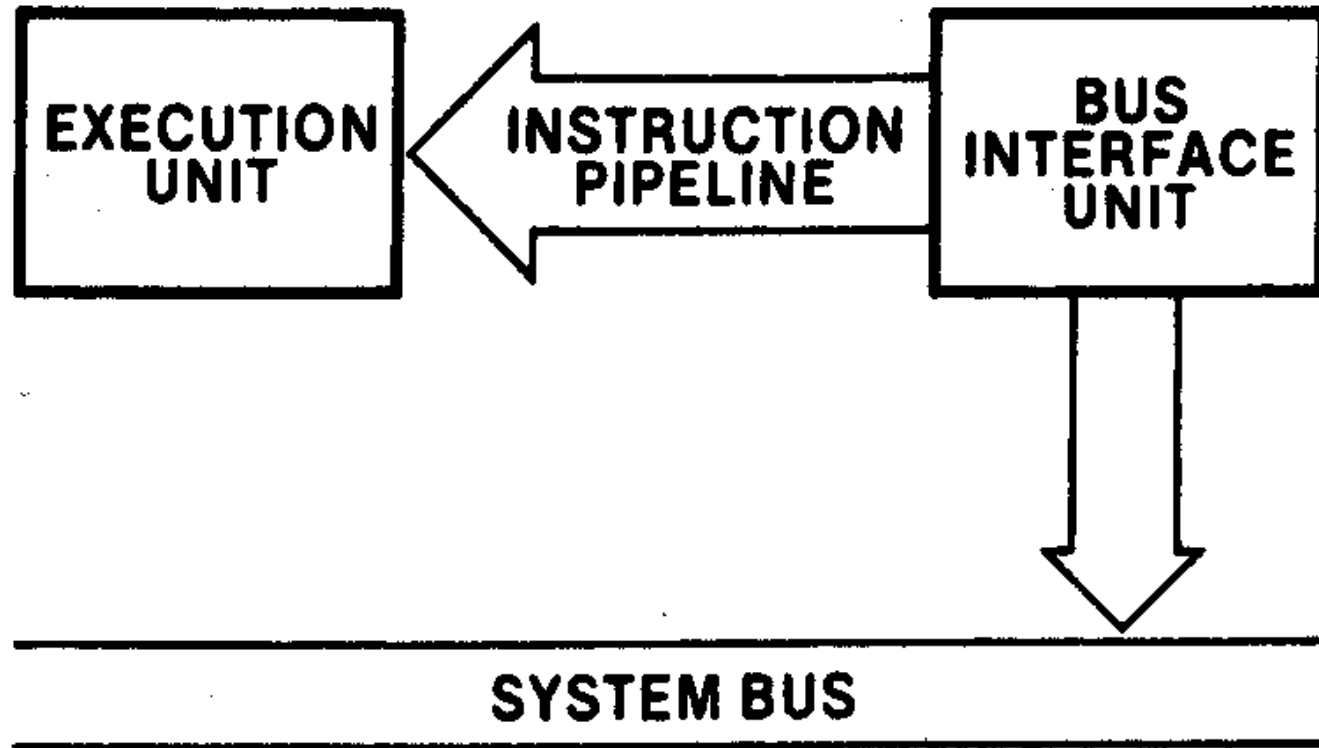
- ▶ 2.9 Pointer and Index Register
- ▶ 2.10 Status Register
- ▶ 2.11 Generating a Memory Address
- ▶ 2.12 The Stack
- ▶ 2.13 Input/output Address Space



## 2.1 Microarchitecture of the 8088/8086 Microprocessor

- ▶ 8088/8086 both employ **parallel processing**
- ▶ 8088/8086 contain two processing unit – the **bus interface unit (BIU) and execution unit (EU)**
- ▶ The bus interface unit is the path that 8088/8086 connects to **external devices**.
- ▶ The system bus includes an 8-bit bidirectional **data** bus for 8088 (16 bits for the 8086), a 20-bit **address** bus, and the signal needed to **control** transfers over the bus.

## *2.1 Microarchitecture of the 8088/8086 Microprocessor*

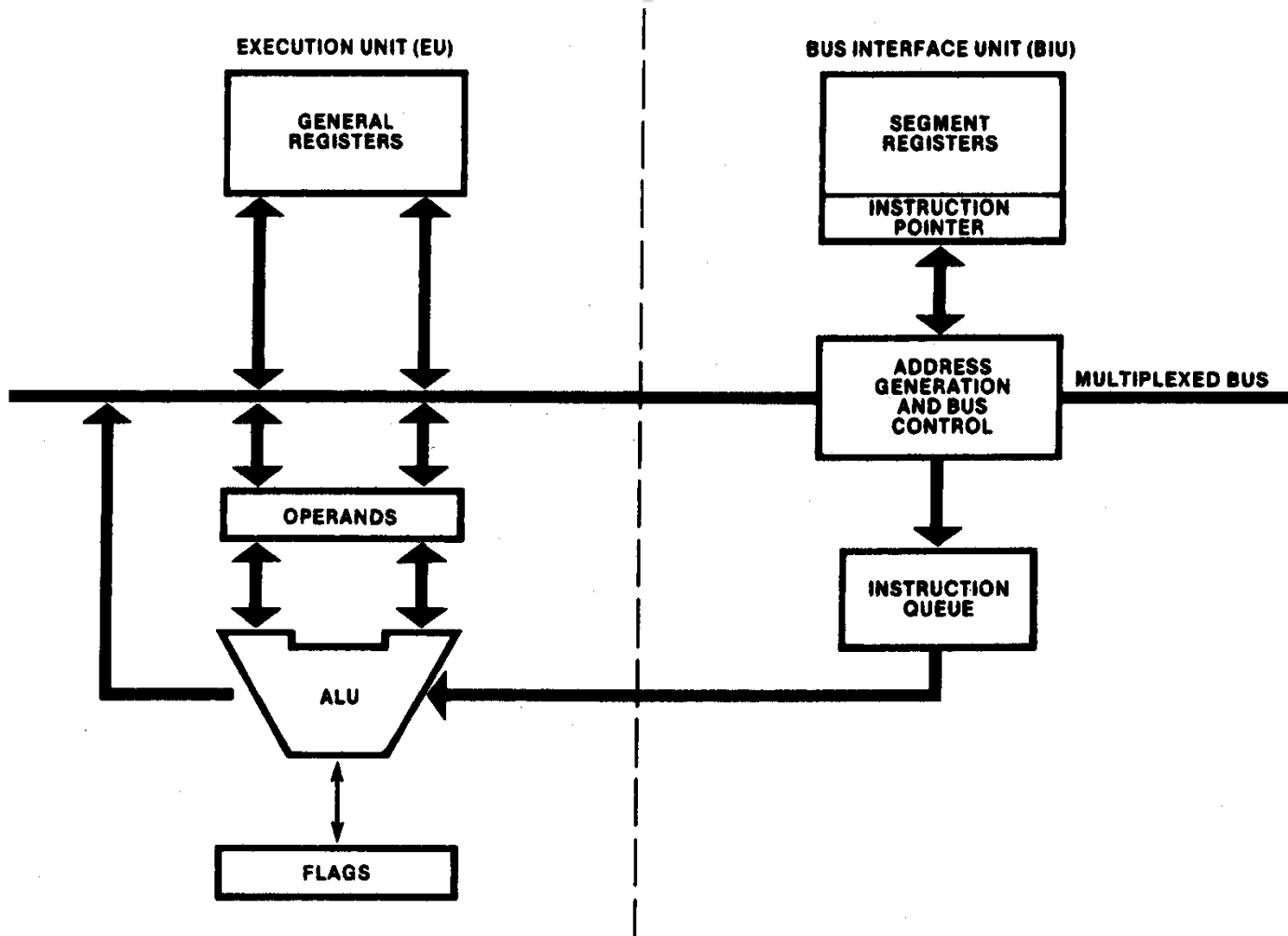


Pipeline architecture of the 8086/8088 microprocessors

# 2.1 Microarchitecture of the 8088/8086 Microprocessor

- ▶ BIU is responsible for: Instruction fetching, memory reading/writing and inputting/outputting data for peripherals.
- ▶ **Components in BIU**
  - Segment register
  - The instruction pointer
  - Address generation adder
  - Bus control logic
  - Instruction queue
- ▶ **Components in EU**
  - Arithmetic logic unit, ALU
  - Status and control flags
  - General-purpose registers
  - Temporary-operand registers

# 2.1 Microarchitecture of the 8088/8086 Microprocessor

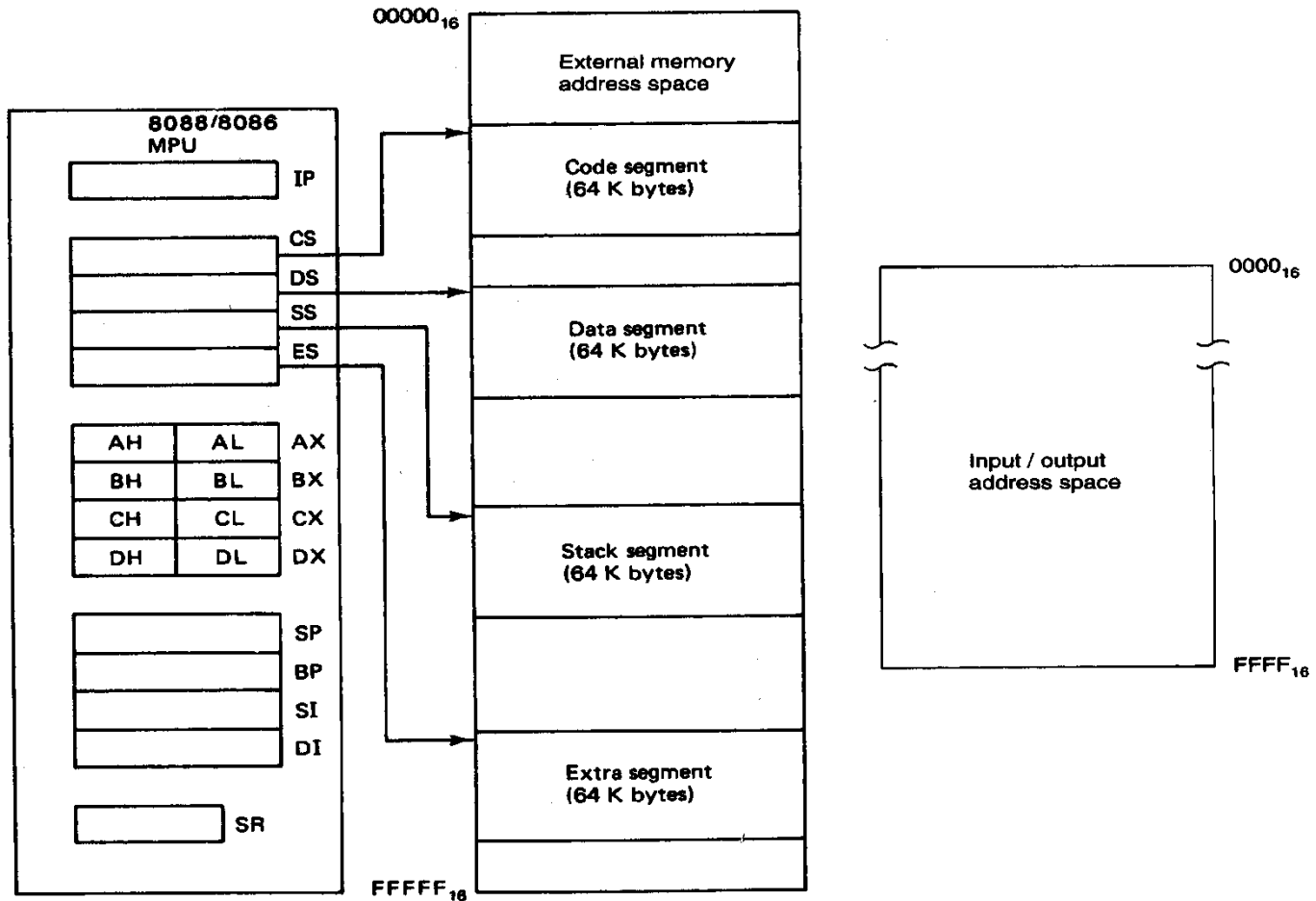


EU and BIU of the 8086/8088 microprocessors

## *2.2 Software Model of the 8088/8086 Microprocessor*

- ▶ Software model describes: available registers, memory address space and I/O address space.
- ▶ 8088 microprocessor includes 13 **16-bit** internal registers.
  - The instruction pointer, IP
  - Four data registers, AX, BX, CX, DX
  - Two pointer register, BP, SP
  - Two index register, SI, DI
  - Four segment registers, CS, DS, SS, ES
- ▶ The status register, SR, with **nine** of its bits implemented for status and control flags.
- ▶ The memory address space is 1 Mbytes and the I/O address space is 64 Kbytes in length.

# 2.2 Software Model of the 8088/8086 Microprocessor



Software model of the 8088/8086 microprocessors

## 2.3 Memory Address Space and Data Organization

- ▶ The 8088 microcomputer supports 1 Mbytes of external memory.
- ▶ The memory of an 8088-based microcomputer is organized as 8-bit bytes, not as 16-bit words.

Memory address space of the 8088/8086 Microcomputer

FFFFFF
FFFFFE
FFFFFD
FFFFFC
5
4
3
2
1
0

## 2.3 Memory Address Space and Data Organization

- ▶ The 8088 can access any two **consecutive** bytes as **word** of data.
- ▶ **Lower** address byte and **higher** address byte.
- ▶ The two bytes represent the word

Address

...  
00725  
00724  
...

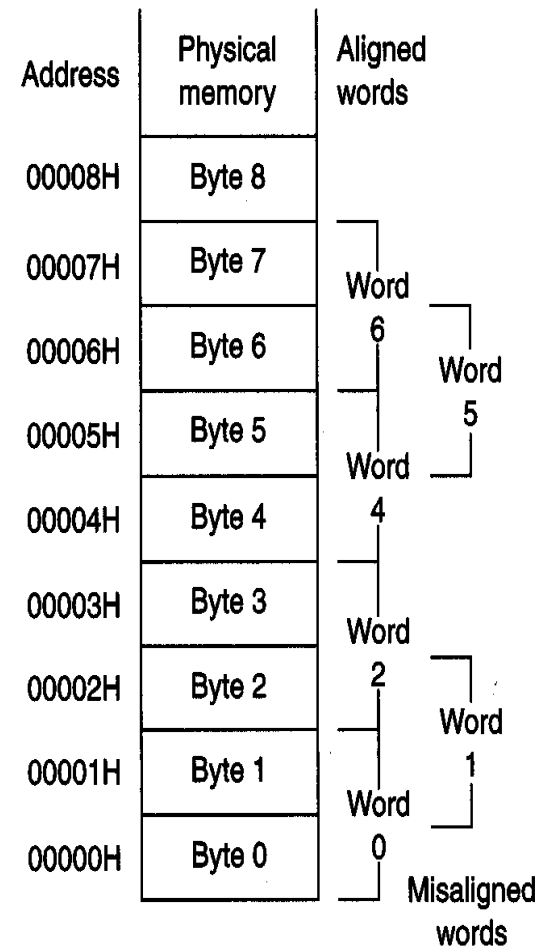
Memory

...  
0101 0101  
0000 0010 = 5502<sub>16</sub>  
...



## 2.3 Memory Address Space and Data Organization

- ▶ **Even-** or **odd-addressed** word  
If the least significant bit of the address is 0, the word is said to be held at an even-addressed boundary.
- ▶ **Aligned** word (even-address) or **misaligned** word (odd-address).
- ▶ Words 0, 2, 4 & 6: aligned
- ▶ Words 1 & 5: misaligned



## 2.3 Memory Address Space and Data Organization

### ▶ EXAMPLE

What is the data word shown below? Express the result in hexadecimal form. Is it stored at an even- or odd addressed word boundary? Is it an aligned or misaligned word of data?

<u>Address</u>	<u>Memory</u>
0072C	1111 1101
0072B	1010 1010

### ▶ Solution:

$$11111101_2 = FD_{16} = FD_H$$

$$10101010_2 = AA_{16} = AA_H$$

### ▶ Together the two bytes give the word

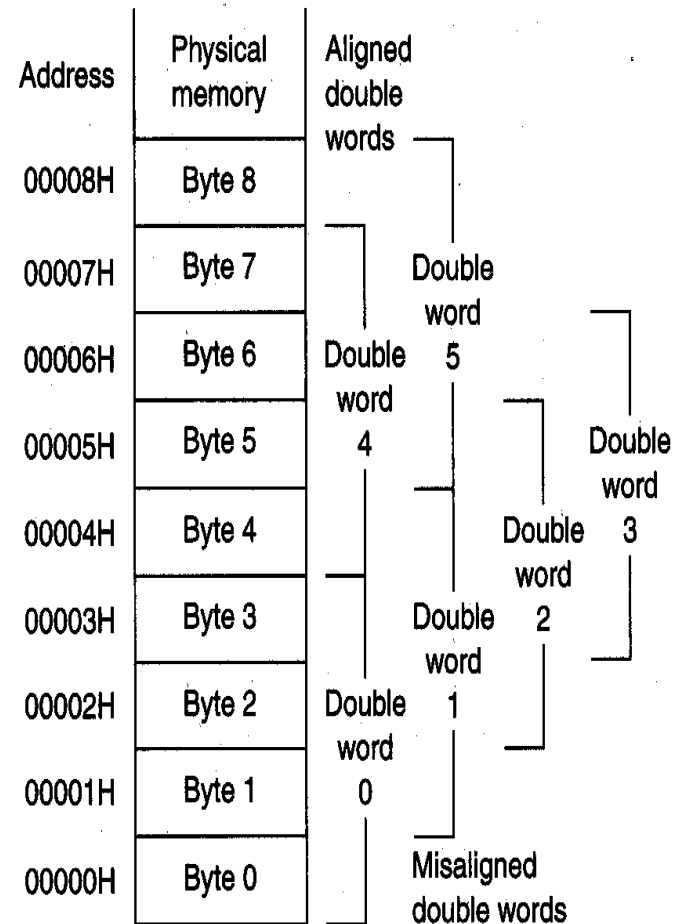
$$1111110110101010_2 = FDAA_{16} = FDAA_H$$

Expressing the address of the least significant byte in binary form gives  $0072B_H = 0072B_{16} = 00000000011100101011_2$

- ▶  $LSB = 1 \rightarrow$  the word is stored at odd-address boundary in memory.
- ▶ Therefore, it is misaligned word of data.

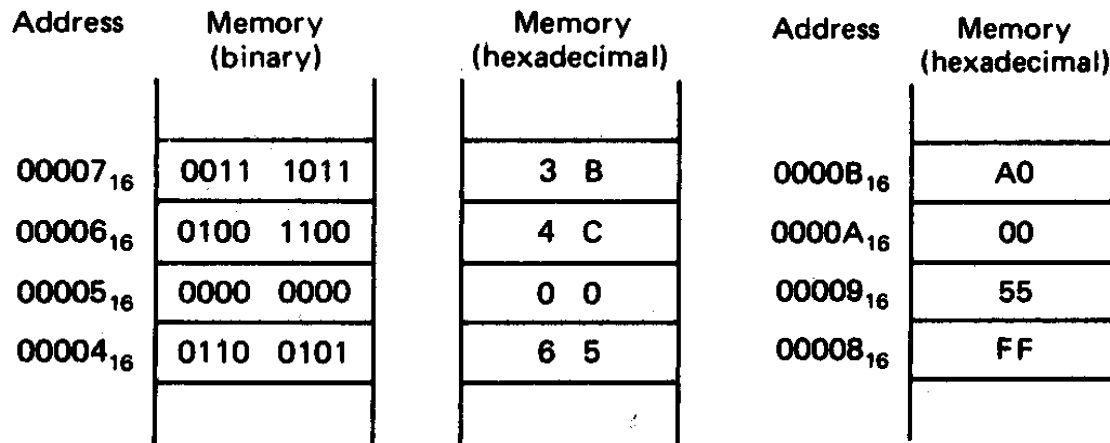
## 2.3 Memory Address Space and Data Organization

- ▶ A **double word** corresponds to **four** consecutive bytes of data stored in memory.
- ▶ Aligned double word is located at addresses of **multiples of 4**.
- ▶ Word 0 & 4: aligned only



## 2.3 Memory Address Space and Data Organization

- ▶ A **pointer** is a double word. The higher address word represents the **segment base address** while the lower address word represents the **offset**.



Example: Segment base address =  $3B4C_{16} = 0011101101001100_2$

- ▶ Offset value =  $0065_{16} = 0000000001100101_2$

## 2.3 Memory Address Space and Data Organization

- ▶ EXAMPLE

How should the pointer with **content** in segment base address equal to  $A000_{16}$  and **content** in offset address equals  $55FF_{16}$  be stored at an even-address boundary starting at  $00008_{16}$ ? Is the double word aligned or misaligned?

- ▶ Solution:

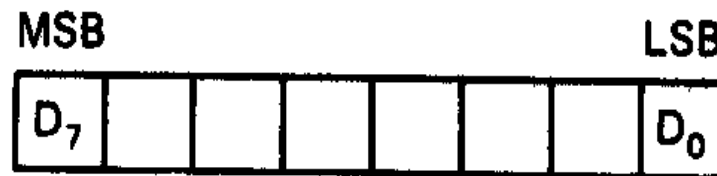
Storage of the two-word pointer requires four consecutive byte locations in memory, starting at address  $00008_{16}$ . The least significant byte of the offset is stored at address  $00008_{16}$  and is shown as  $FF_{16}$  in the previous figure. The most significant byte of the offset,  $55_{16}$ , is stored at address  $00009_{16}$ . These two bytes are followed by the least significant byte of the segment base address,  $00_{16}$ , at address  $0000A_{16}$ , and its most significant byte,  $A0_{16}$ , at address  $0000B_{16}$ . Since the double word is stored in memory starting at address  $00008_{16}$ , it is aligned.

# Previous Example Solution

Address	Memory (hexadecimal)
$0000B_{16}$	A0
$0000A_{16}$	00
$00009_{16}$	55
$00008_{16}$	FF

# 2.4 Data Types

- ▶ **Integer** data type
  - Unsigned or signed integer
  - Byte-wide or word-wide integer



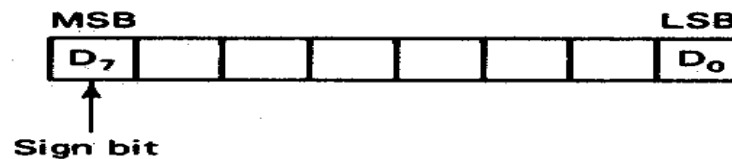
(a)



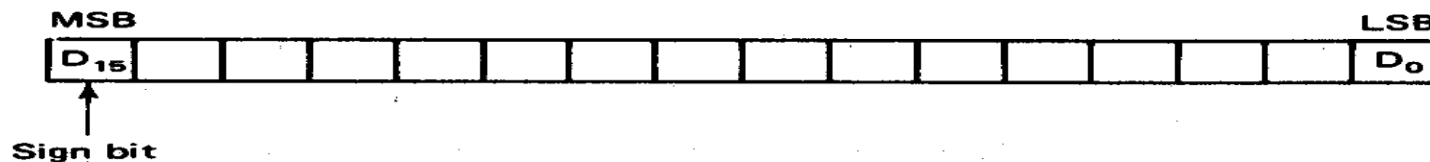
Unsigned byte and unsigned word integer

## 2.4 Data Types

- ▶ The most significant bit of a signed integer is a *sign bit*. A **zero** in this bit position identifies a **positive** number.
- ▶ The range of a **signed byte** integer is  $+127 \sim -128$ . The range of a **signed word** integer is  $+32767 \sim -32768$ .
- ▶ The 8088 always expresses **negative** numbers in **2's complement**.



(a)



(b)

signed byte and signed word integer



## 2.4 Data Types

### ▶ EXAMPLE

A signed word integer equals  $\text{FEFF}_{16}$ . What decimal number does it represent?

### ▶ Solution:

$$\text{FEFF}_{16} = 1111111011111111_2$$

- The most significant bit is 1, the number is negative and is in 2's complement form.
- Converting to its binary equivalent by subtracting 1 from the least significant bit and then complement all bits give

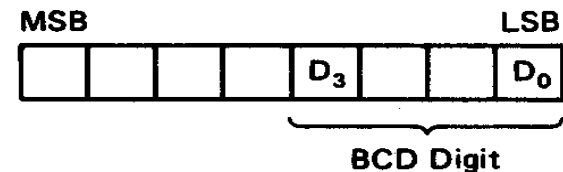
$$\begin{aligned}\text{FEFF}_{16} &= -0000000100000001_2 \\ &= -257\end{aligned}$$

# 2.4 Data Types

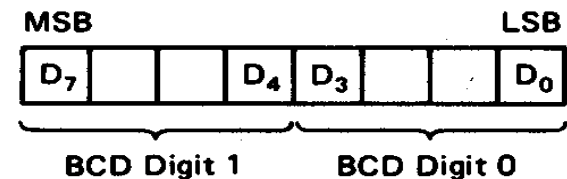
- ▶ The 8088 can also process data that is coded as *binary-coded decimal (BCD) numbers*.
- ▶ BCD data can be stored in **unpacked** (upper 4 bits = 0) or **packed** forms.

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

(a)



(b)



(c)

(a) BCD numbers (b) Unpacked BCD digit (c) Packed BCD digit

## 2.4 Data Types

### ▶ EXAMPLE

The packed BCD data stored at byte address  $01000_{16}$  equals  $10010001_2$ . What is the two digit decimal number?

### ▶ Solution:

Writing the value  $10010001_2$  as separate BCD digits gives

$$10010001_2 = 1001_{\text{BCD}}0001_{\text{BCD}} = 91_{10}$$

# 2.4 Data Types

- ▶ The ASCII (American Standard Code for Information Interchange) digit
- ▶ The 8088 can process ASCII characters too.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	

## 2.4 Data Types

### ▶ EXAMPLE

Byte addresses  $01100_{16}$  through  $01104_{16}$  contain the ASCII data  $01000001$ ,  $01010011$ ,  $01000011$ ,  $01001001$ , and  $01001001$ , respectively. What do the data stand for?

### ▶ Solution:

Using the ASCII table, the data are converted to ASCII code:

$(01100H) = 01000001_2 = A$

$(01101H) = 01010011_2 = S$

$(01102H) = 01000011_2 = C$

$(01103H) = 01001001_2 = I$

$(01104H) = 01001001_2 = I$

## 2.5 Segment Registers and Memory Segmentation

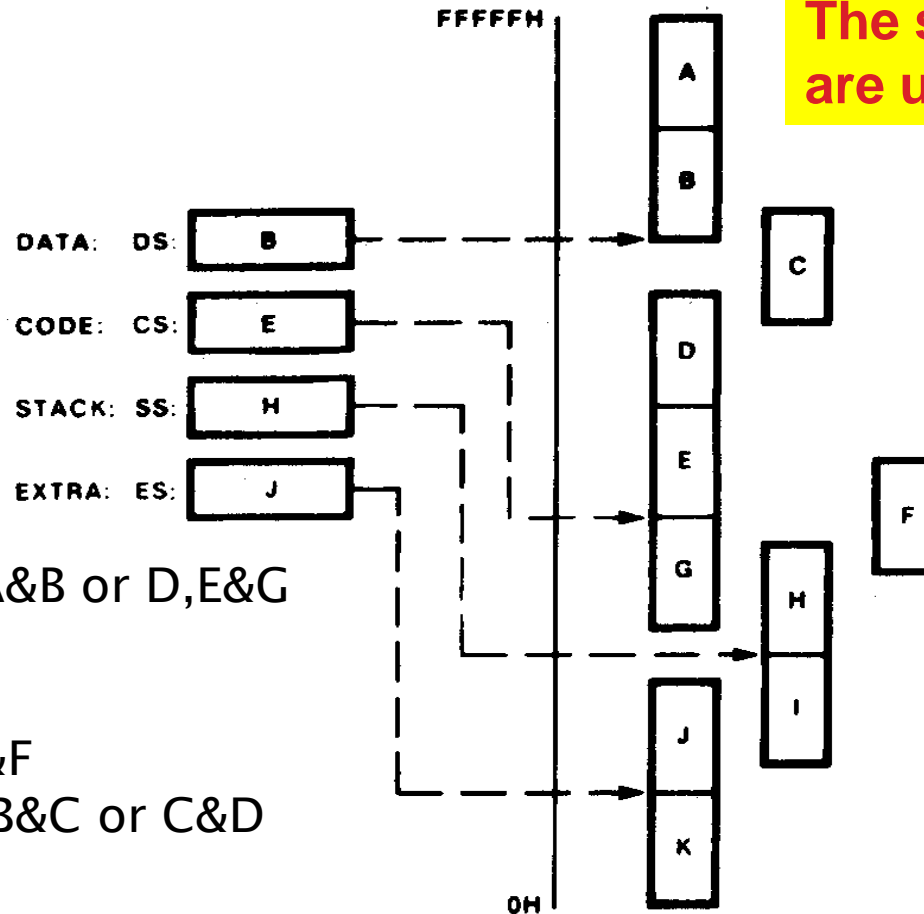
- ▶ A **segment** represents an independently **addressable** unit of memory consisting of **64K** consecutive byte wide storage locations.
- ▶ Each segment is assigned a **base address** that identifies its starting point.
- ▶ Only **four** segments can be **active** at a time:
  - The code segment
  - The stack segment
  - The data segment
  - The extra segment
- ▶ The addresses of the active segments are stored in the four internal **segment registers**: CS, SS, DS, ES.

## *2.5 Segment Registers and Memory Segmentation*

- ▶ **Four** segments give a maximum of **256Kbytes** of active memory.
  - Code segment – 64K
  - Stack – 64K
  - Data storage – 128K
- ▶ The base address of a segment must reside on a **16-byte address boundary**.
- ▶ User accessible segments can be set up to be contiguous, adjacent, disjointed, or even overlapping.

# 2.5 Segment Registers and Memory Segmentation

The segments registers are user accessible



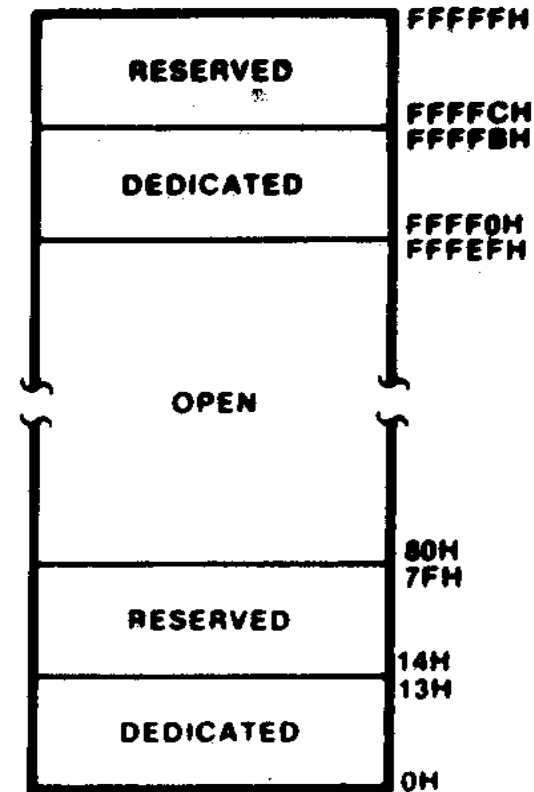
- Contiguous—A&B or D,E&G or J & K
- Adjacent
- Disjointed—C&F
- Overlapping—B&C or C&D

Contiguous, adjacent, disjointed, and overlapping segments



## 2.6 Dedicated, Reserved, and General-Used Memory

- ▶ The **dedicated memory** ( $00000_{16} \sim 00013_{16}$ ) are used for storage of the **pointers** to 8088's **internal** interrupt service routines and exceptions.
- ▶ The **reserved memory** ( $00014_{16} \sim 0007F_{16}$ ) are used for storage of the **pointers** to **user-defined** interrupts.
- ▶ The 128-byte dedicated and reserved memory can contain 32 interrupt pointers (double word 4-bytes) ( $128/4$ ).
- ▶ The **general-use memory** ( $00080_{16} \sim FFFE_{16}$ ) stores **data** or **instructions** of the program.
- ▶ The **dedicated memory** ( $FFFE0_{16} \sim FFFEB_{16}$ ) are used for hardware **reset jump** instruction.
- ▶ The **reserved memory** ( $FFFCF_{16} \sim FFFFF_{16}$ ) are preserved for **future** use.



## 2.7 Instruction Pointer

- ▶ The *instruction pointer (IP)* identifies the location of the *next* word of instruction code to be fetched from the current code segment of memory.
- ▶ The offset in IP is combined with the current value in CS to generate the address of the instruction code. *CS:IP* forms 20-bit physical address of next word of instruction code.
- ▶ During normal operation, the 8088 fetches instructions from the code segment of memory, stores them in its instruction queue (*why?*), and executes them one after the other.

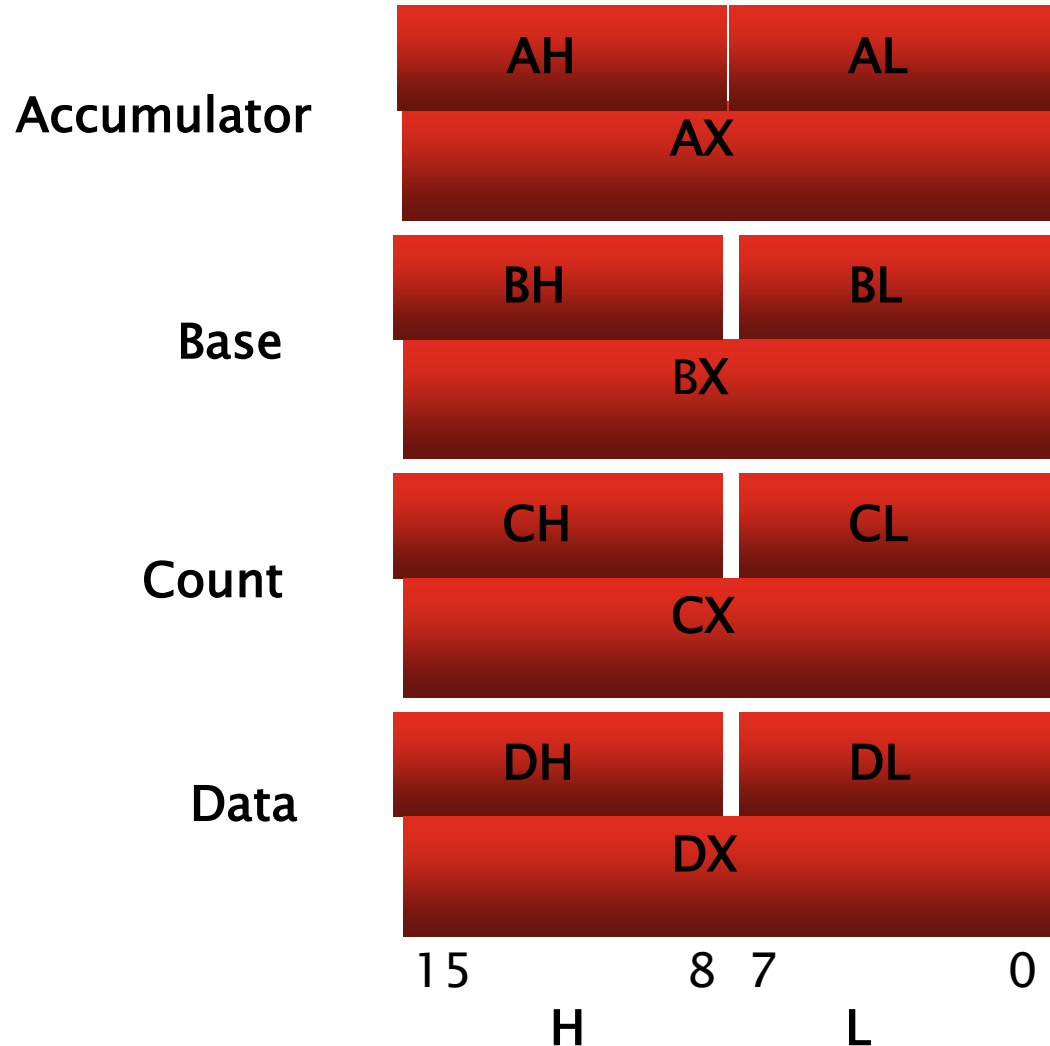
# 2.7 Instruction Pointer

- ▶ **Instruction fetch** sequence:
  - 8088/8086 fetches a word of instruction code from code segment in memory
    - Increments value in **IP** by **2**
    - Word placed in the instruction queue to wait for execution
    - 8088 prefetches up to **4** bytes of code
- ▶ **Instruction execution** sequence:
  - Instruction is read from output of instruction queue and executed
    - **Operands** read from data memory, internal registers
    - **Operation** specified by the instruction performed on operands
    - Result written back to either data memory or internal register

## *2.8 Data Registers*

- ▶ The **Data registers** are used for temporary storage of frequently used intermediate results.
- ▶ The contents of the data registers can be read, loaded, or modified through software.
- ▶ The four data registers are:
  - **Accumulator** register, A
  - **Base** register, B
  - **Counter** register, C
  - **Data** register, D
- ▶ Each register can be accessed either as a whole (16 bits) for word data or as **8-bit** data for byte-wide operation.

# 2.8 Data Registers



General-purpose data registers of 8088 microprocessor

## 2.8 Data Registers

- ▶ Uses:
  - Hold **data** such as source or destination operands for most operations—ADD, AND, SHL (**faster access**)
  - Hold **address** pointer for accessing memory
- ▶ Some also have dedicated special uses
  - C—count for loop, repeat string, shift, and rotate operations
  - B—Table look-up translations, base address
  - D—indirect I/O and string I/O

## 2.8 Data Registers

Register	Operations
AX	Word multiply, word divide, word I/O
AL	Byte multiply, byte divide, byte I/O, translate, decimal arithmetic
AH	Byte multiply, byte divide
BX	Translate
CX	String operations, loops
CL	Variable shift and rotate
DX	Word multiply, word divide, indirect I/O

Dedicated register functions

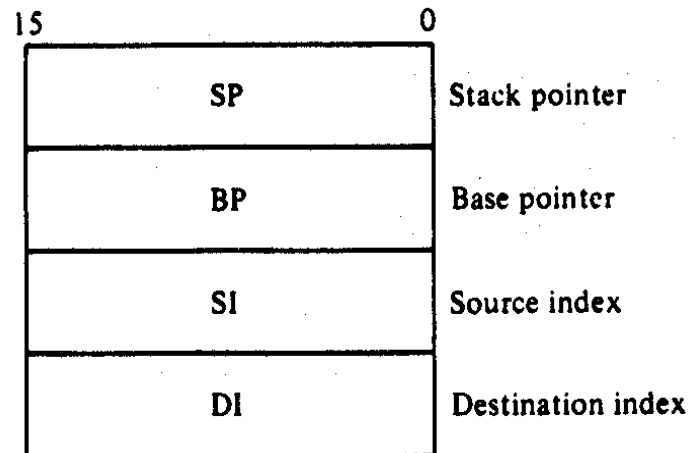
## 2.9 *Pointer and Index Registers*

- ▶ The pointer registers and index registers are used to store **offset** addresses.
- ▶ Values held in the **index** registers are used to reference data relative to the **data segment** or extra segment.
- ▶ The **pointer** registers are used to store offset addresses of memory location relative to the **stack segment** register.
- ▶ Combining **SP** with the value in **SS** (SS:SP) results in a 20-bit address that points to the ***top of the stack*** (TOS).
- ▶ **BP** is used to access data within the stack segment of memory. It is commonly used to reference subroutine parameters.



# 2.9 Pointer and Index Registers

- ▶ The **index** registers are used to hold offset addresses for instructions that access data in the data segment.
- ▶ The source index register (**SI**) is used for a **source** operand, and the destination index (**DI**) is used for a **destination** operand.
  - DS:SI—points to source operand in data segment
  - DS:DI—points to destination operand in data segment
- ▶ Also used to access information in the extra segment (ES)
- ▶ The four registers must always be used for **16-bit** operations.

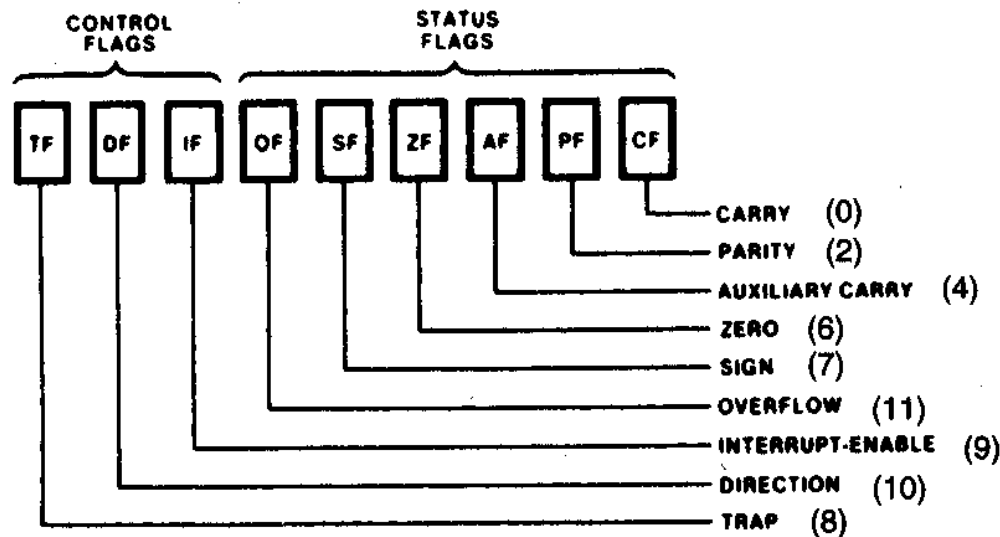


## 2.10 Status Register

- ▶ The *status register*, also called the *flags register*, indicate conditions that are produced as the result of executing an instruction.
- ▶ Only **nine** bits of the register are implemented. **Six** of these bits represent *status flags* and the other **three** bits represent *control flags*
- ▶ The 8088 provides instructions within its instruction set that are able to use these flags to alter the sequence in which the program is executed.

## 2.10 Status Register

- ▶ Status and control bits maintained in the flags register



- ▶ Generally Set and Tested Individually
- ▶ 9 1-bit flags in 8086; 7 are unused

# 2.10 Status Register

- ▶ Status flags indicate current processor status.

<b>CF</b>	Carry Flag	Arithmetic Carry/Borrow
<b>OF</b>	Overflow Flag	Arithmetic Overflow
<b>ZF</b>	Zero Flag	Zero Result; Equal Compare
<b>SF</b>	Sign Flag	Negative Result; Non-Equal Compare
<b>PF</b>	Parity Flag	Even Number of “1” bits
<b>AF</b>	Auxiliary Carry	Used with BCD Arithmetic

Odd parity

## *2.10 Status Register*

- ▶ Control flags influence the 8086 during execution phase

<b>DF</b>	Direction Flag	Auto-Increment/Decrement used for “string operations”
<b>IF</b>	Interrupt Flag	Enables Interrupts allows “fetch-execute” to be interrupted. Used to enable/disable external maskable interrupt requests
<b>TF</b>	Trap Flag	Allows Single-Step for debugging; causes interrupt after each op

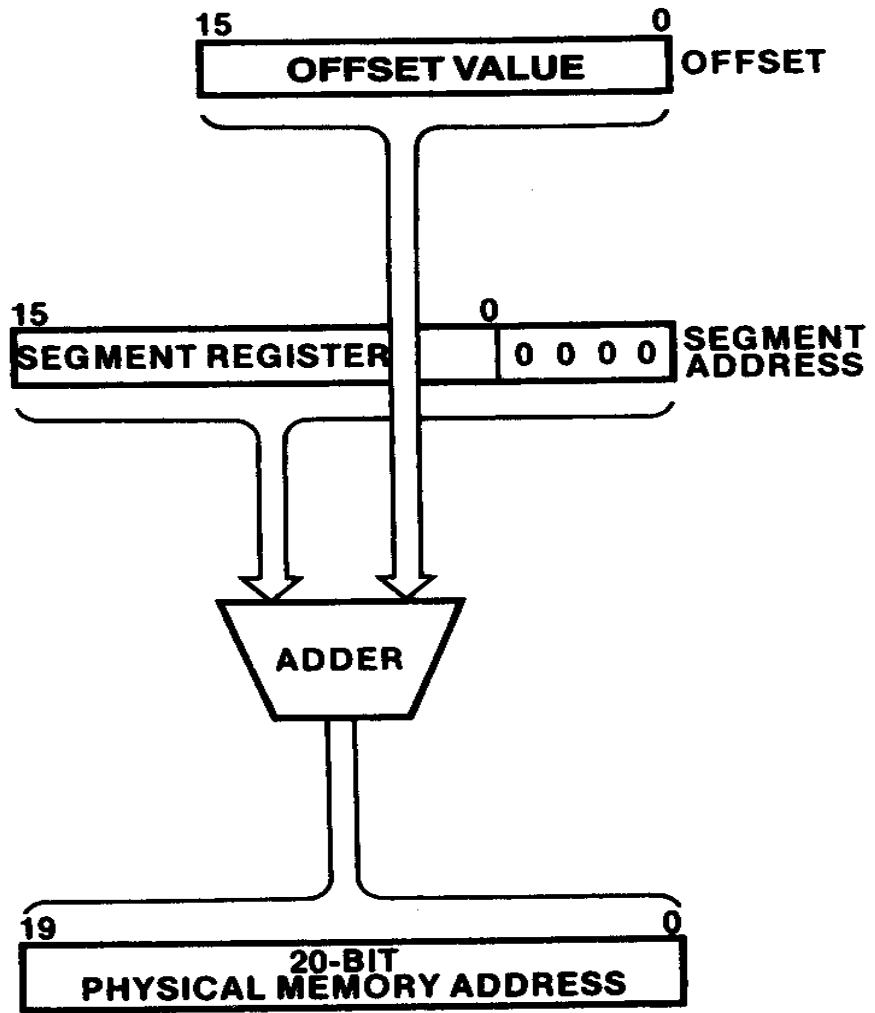
## 2.11 *Generating a Memory Address*

- ▶ A *logical address in the 8088 microcomputer* system is described by a segment base and an offset.
- ▶ The *physical addresses that are used to access* memory are 20 bits in length.
- ▶ The generation of the physical address involves combining a 16-bit offset value that is located in the instruction pointer, a base pointer, an index register, or a pointer register and a 16-bit segment base value that is located in one of the segment register.
- ▶ **Segment** base address (CS, DS, ES, SS) are 16 bit quantities.
- ▶ **Offsets** (IP, SI, DI, BX, DX, SP, BP, etc.) are 16 bit quantities.

# 2.11 Generating a Memory Address

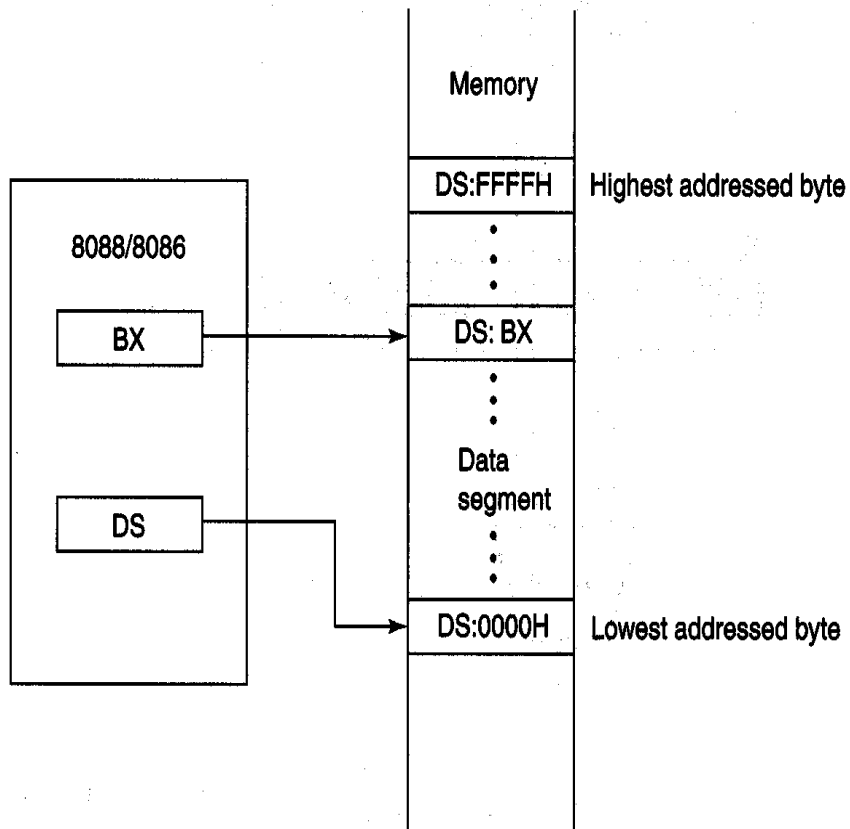
□ **Physical Address**: actual address used for accessing memory: 20-bits in length Formed by:

- Shifting the value of the 16-bit segment base address left 4 bit positions
- Filling the vacated four LSBs with 0s
- Adding the 16-bit offset



Generating a physical address

# 2.11 Generating a Memory Address



Boundary of a segment

- Four active segments CS, DS, ES, and SS
  - Each 64-k bytes in size → maximum of 256K-bytes of active memory
  - 64K-bytes for code
  - 64K-bytes for stack
  - 128K-bytes for data (data and extra)
  - **Starting** address of a data segment DS:0H → lowest addressed byte
  - **Ending** address of a data segment DS:FFFFH → highest addressed byte
  - Address of an element of data in a data segment
    - DS:BX → address of byte, word, or double word element of data in the data segment



# 2.11 Generating a Memory Address

## □ Example:

Segment base address = 1234H

Offset = 0022H

1234H = 0001 0010 0011 0100<sub>2</sub>

0022H = 0000 0000 0010 0010<sub>2</sub>

Shifting base address,

00010010001101000000<sub>2</sub> = 12340H

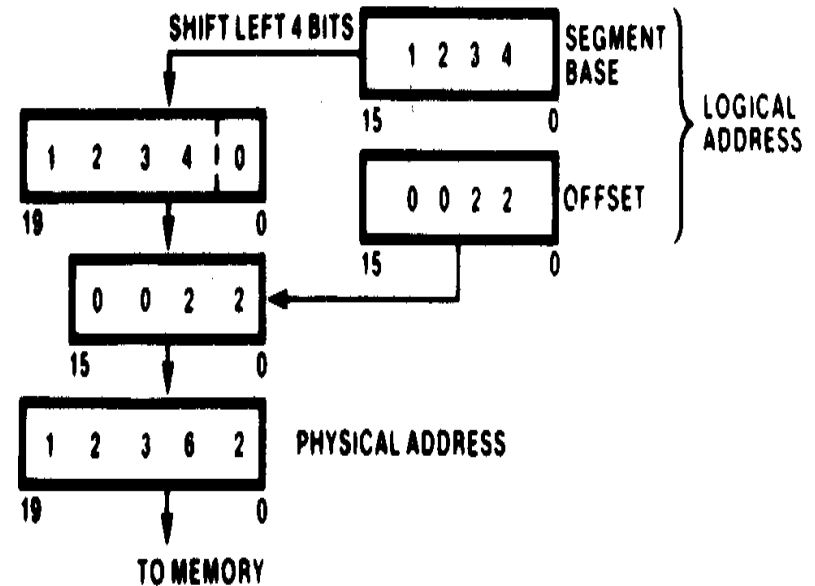
Adding segment address and offset

00010010001101000000<sub>2</sub> +

0000000000100010<sub>2</sub> =

= 00010010001101100010<sub>2</sub>

= 12362H



## 2.11 Generating a Memory Address

### ▶ EXAMPLE

What would be the offset required to map to physical address location  $002C3_{16}$  if the contents of the corresponding segment register are  $002A_{16}$ ?

### ▶ Solution:

The offset value can be obtained by shifting the contents of the segment of the segment register left by four bit positions and then subtracting from the physical address. Shifting left give

$002A0_{16}$

Now subtracting, we get the value of the offset:

$$002C3_{16} - 002A0_{16} = 0023_{16}$$

# 2.11 Generating a Memory Address

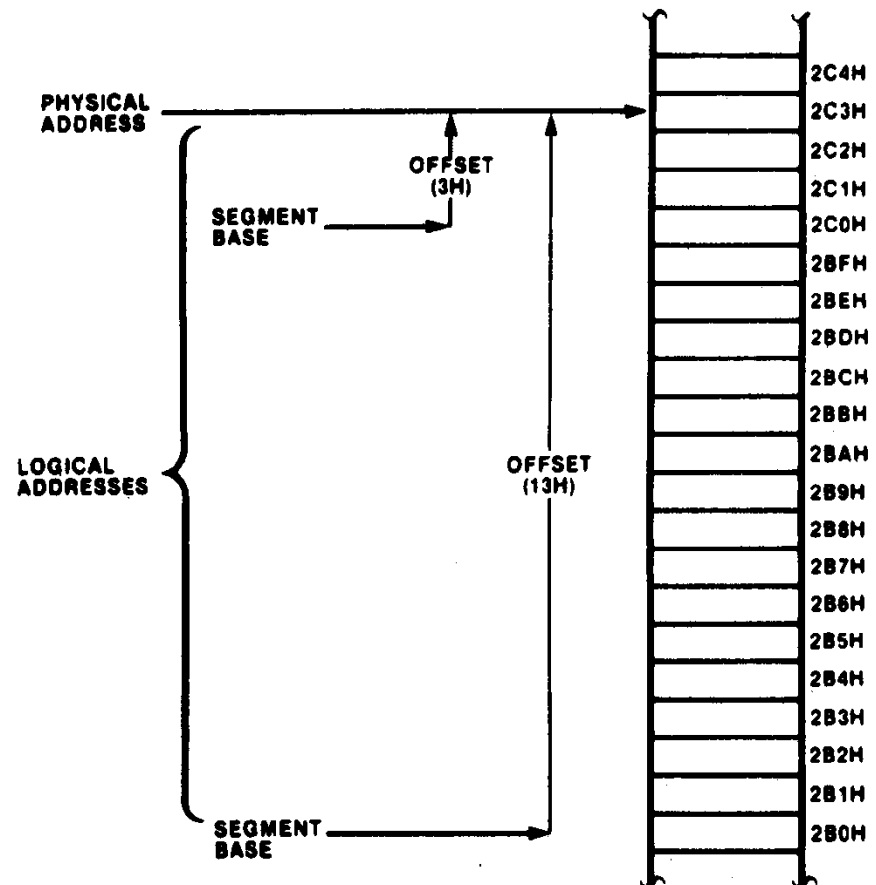
- ▶ **Different** logical addresses can be mapped to the **same** physical address location in memory.

□ Examples:

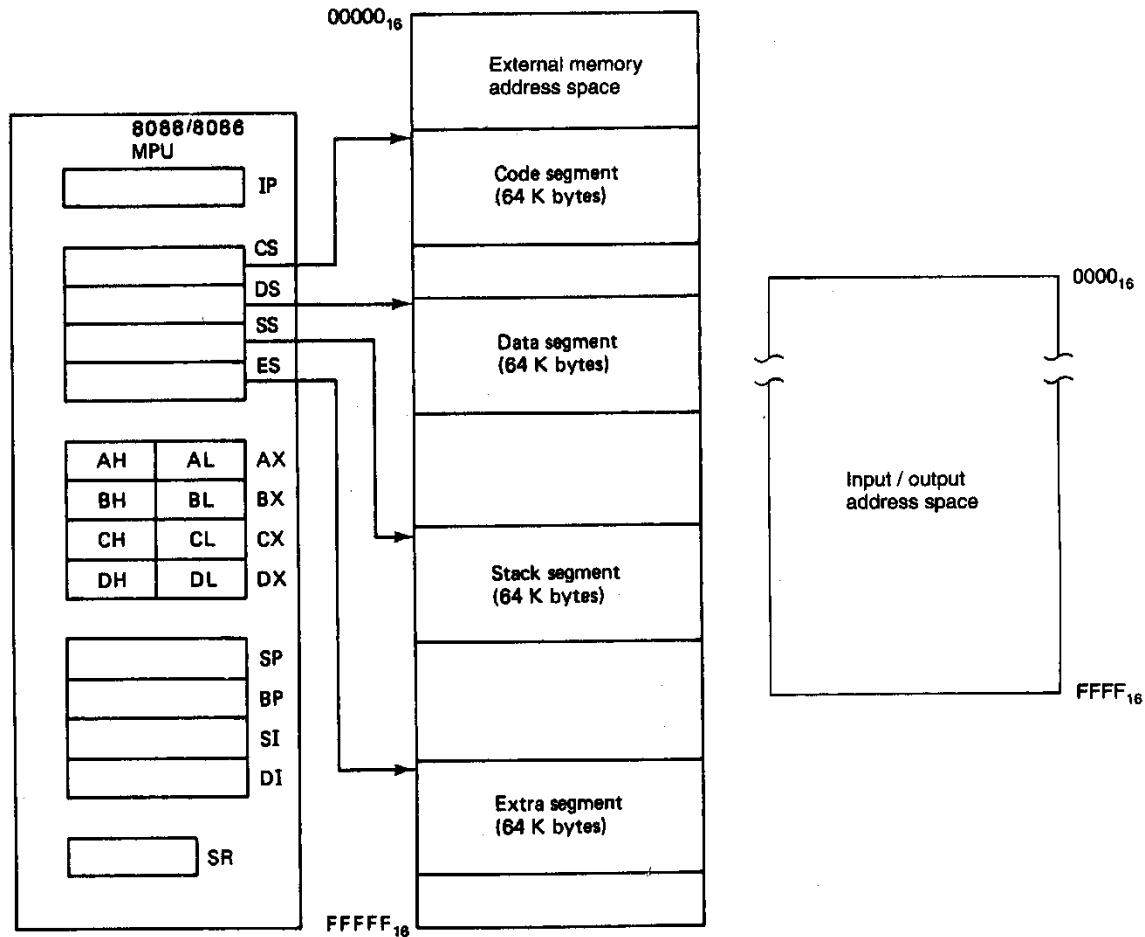
$$2BH:13H = 002B0H + 0013H = 002C3H$$

$$2CH:3H = 002C0H + 0003H = 002C3H$$

□ These logical addresses are called “**aliases**”



# 2.2 Software Model of the 8088/8086 Microprocessor



Software model of the 8088/8086 microprocessors

## 2.12 The Stack

- ▶ The **stack** is implemented for temporary storage of information such as data or addresses.
- ▶ The stack is **64KBytes** long and is organized from a software point of view as **32K words**.
- ▶ The contents of the **SP** and **BP** registers are used as offsets into the stack segment memory while the segment base value is in the **SS** register.
- ▶ Push instructions (**PUSH**) and pop instructions (**POP**)
- ▶ **Top of the stack (TOS) and bottom of the stack (BOS)**
- ▶ The 8088 can push **word-wide data and address** information onto the stack from registers or memory.
- ▶ **Many stacks** can exist but only one is active at a time.

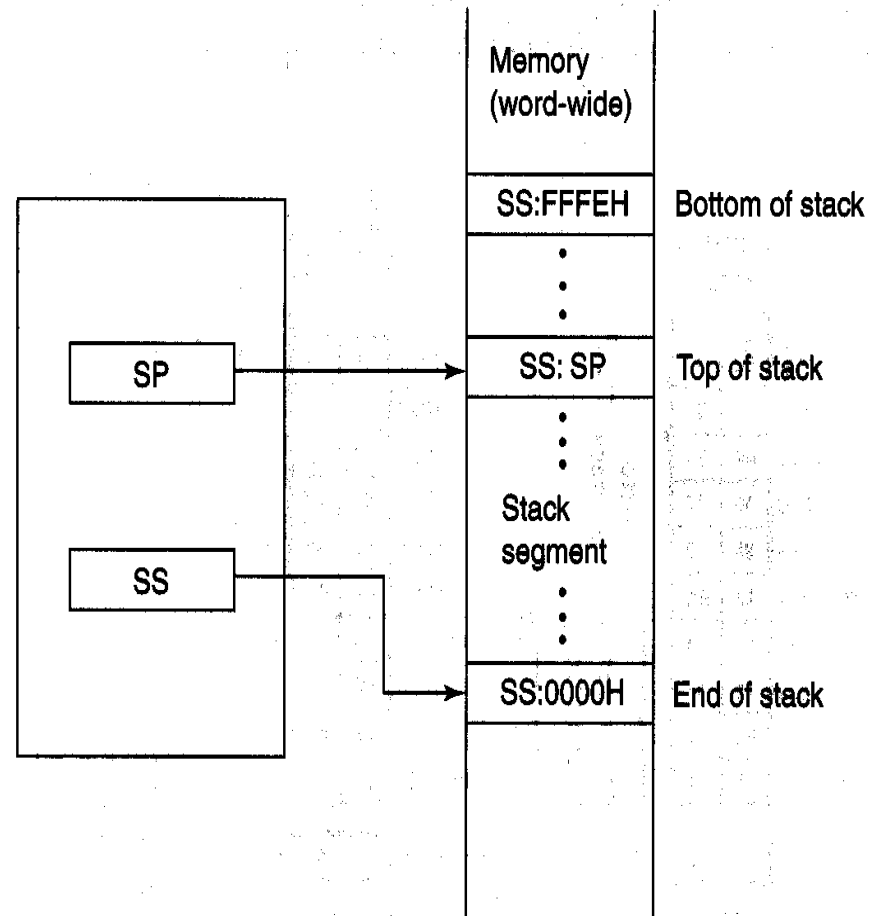
# 2.12 The Stack

Organization of stack:

- **SS:0000H** → end of stack (lowest addressed word)
- **SS:FFFEH** → bottom of stack (highest addressed word)
- **SS:SP** → top of stack (last stack location to which data was pushed)
- Stack grows down from higher to lower address
- Used by call, push, pop, and return operations
- Examples

**PUSH SI** → causes the current content of the SI register to be pushed onto the “top of the stack”

**POP SI** → causes the value at the “top of the stack” to be popped back into the SI register



Stack segment of memory

# 2.12 The Stack

□ Stack status prior to execution of the instruction **PUSH AX**:

AX = 1234H

SS = 0105H

**AEOS** = SS:00 → 01050H = end of stack

SP = 0008H

**ABOS** =  $01050_{16} + FFFE_{16}$   
 = SS:FFEH → 1104EH

**ATOS** =  $01050_{16} + 0008_{16}$   
 = SS:SP → 01058H = current top of stack

BBAAH = Last value pushed to stack

Addresses < 01058H = **invalid** stack data

Addresses ≥ 01058H = **valid** stack data

□ In response to the execution of **PUSH AX** instruction:

1. SP → 0006H decremented by 2

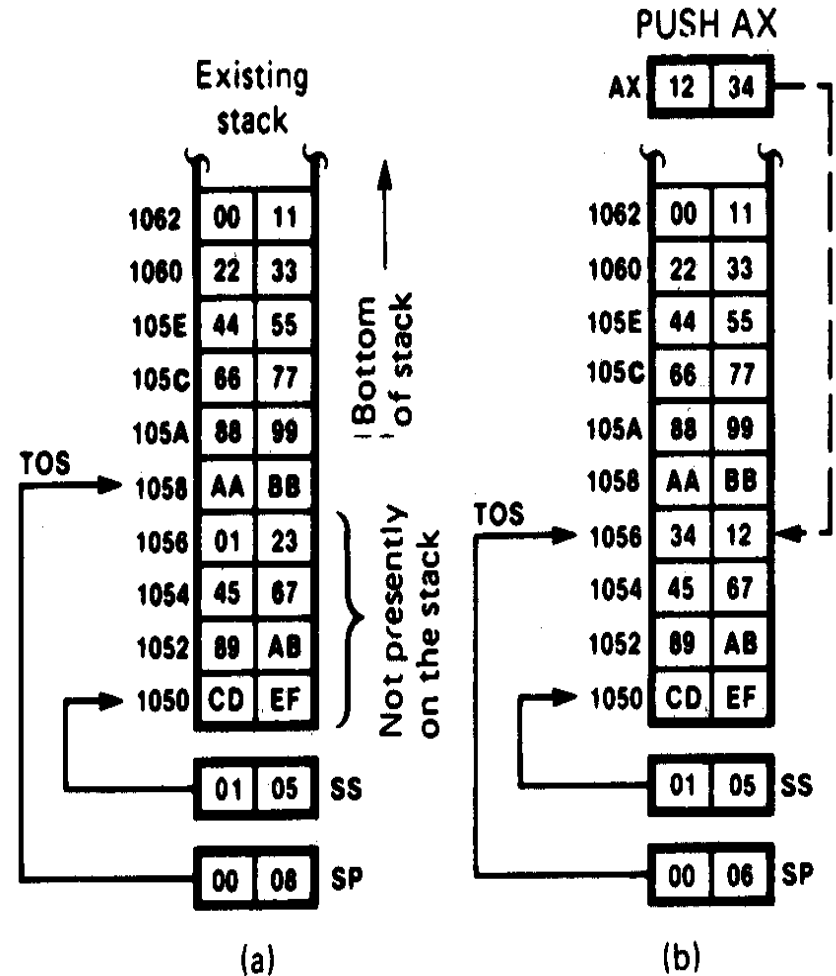
ATOP → 01056H

2. Memory write to stack segment

AL = 34H → 01056H

AH = 12H → 01057H

- How many free spaces ?
- When the stack get full ?



# 2.12 The Stack

## • EXAMPLE: Pop operation

□ Status of the stack prior to execution of the instruction POP AX:

AX = XXXXH

SS = 0105H

SP = 0006H

ATOS = SS:SP → 01056H

1234H = Last value pushed to stack

Addresses < 01056H = invalid stack data

Addresses ≥ 01056H = valid stack data

□ In response to the execution of POP AX instruction

1. Memory read to AX

01056H = 34H → AL

01057H = 12H → AH

2. SP → 0008H incremented by 2

ATOP → 01058H

□ In response to the execution of POP BX instruction

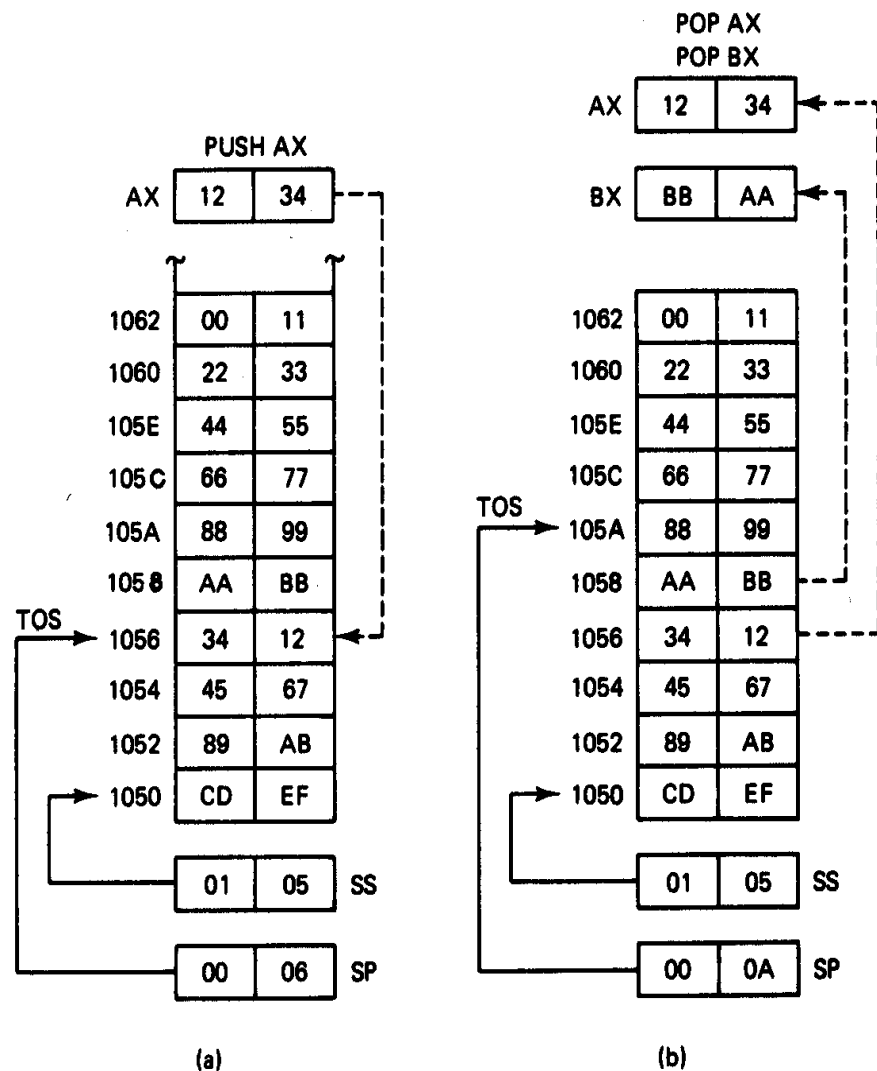
1. Memory read to BX

01058H = AAH → BL

01059H = BBH → BH

2. SP → 000AH incremented by 2:

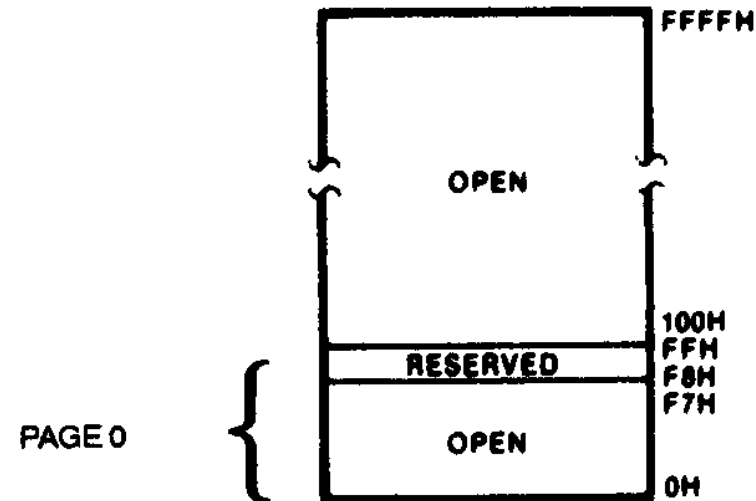
ATOP → 0105AH



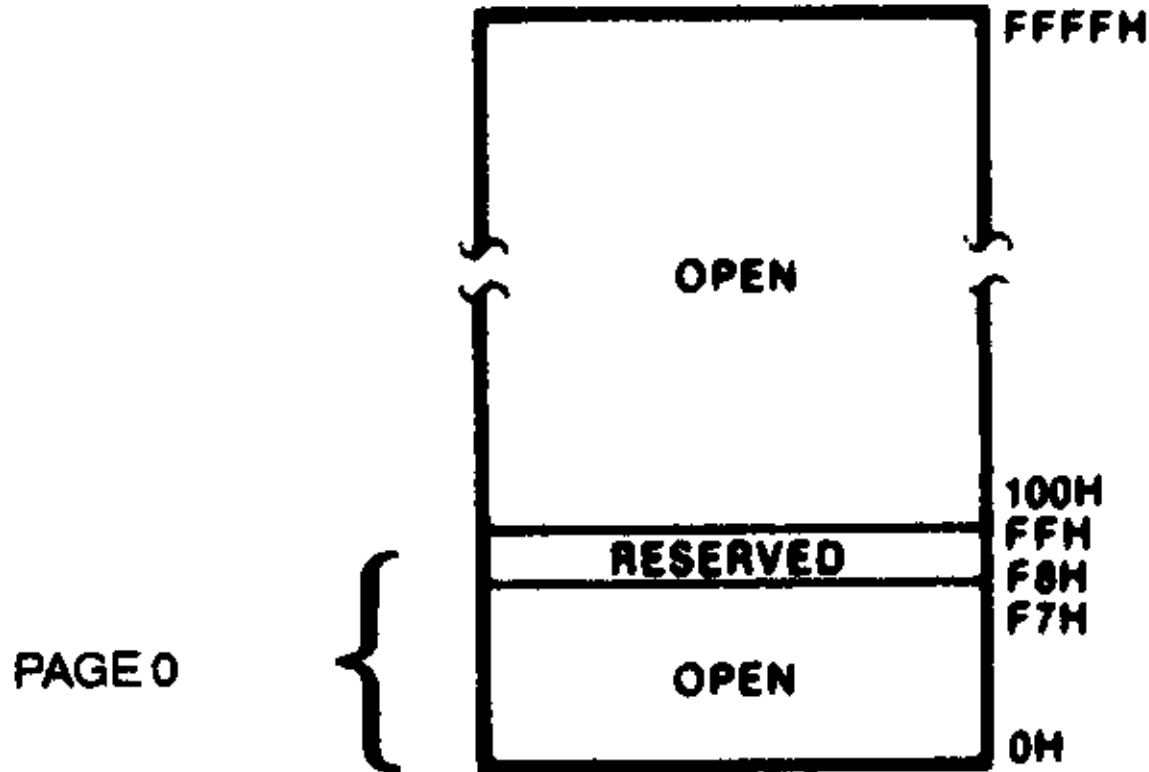


## 2.13 Input/Output Address Space

- ▶ The 8088 has separate memory and input/output (I/O) address space.
- ▶ The I/O address space is the place where I/O interfaces, such as printer and terminal ports, are implemented.
- ▶ The I/O address range is from  $0000_{16}$  to  $FFFF_{16}$ . This represents **64KByte** addresses.
- ▶ The I/O addresses are **16** bits long. Each of these addresses corresponds to one byte-wide I/O port.
- ▶ Certain I/O instructions can only perform operations to addresses  $0000_{16}$  thru  $00FF_{16}$  (*page 0*).
- ▶ Ports F8H through FF **reserved**



## 2.13 Input/Output Address Space



I/O address space

# H.W. #2

- Solve the following problems from Chapter 2 from the course textbook:

3, 9, 14, 19, 26, 32, 34, 37, 45, 49, 55, 60, 65

If  $SS = C000$ ,  $SP = FF00$

1) How many data words currently in the stack

2) How the value  $EE11$  will be pushed in the stack

# **CPE 408330**

## **Assembly Language and Microprocessors**

### **Chapter 3: Assembly Language Programming**

[Computer Engineering Department,  
Hashemite University]

# Lecture Outline

- ▶ 3.1 Software: The Microcomputer Program
- ▶ 3.2 Assembly Language Programming Development on the PC
- ▶ 3.3 The Instruction Set
- ▶ 3.4 The MOV Instruction
- ▶ 3.5 Addressing Modes

# 3.1 Software : The Microcomputer Program

- ▶ A **program** is a sequence of commands that tell the microprocessor what to do.
- ▶ Each command is called “**instruction**”.
- ▶ An instruction can be divided into **two** parts:
  - Operation code (**opcode**) – one- to five-letter *mnemonic*
- ▶ **Operands**: Identify whether the elements of data to be processed are in registers or memory.



- ▶ Source operand– location of one operand to be processed
- ▶ Destination operand—location of the other operand to be processed and the location of the result
- ▶ Format of an assembly statement:

**LABEL: INSTRUCTION ; COMMENT**

# *3.1 Software : The Microcomputer Program*

- ▶ Label—address identifier for the statement
- ▶ Instruction—the operation to be performed
- ▶ Comment—documents the purpose of the statement
- ▶ Example:

**START: MOV AX, BX ; COPY BX into AX**

- ▶ Other examples:

**INC SI ;Update pointer**

**ADD AX, BX**

- Few instructions have a label—usually marks a point to jump
- Not all instructions need a comment
- ▶ What is the “MOV part of the instruction called?
- ▶ What is the BX part of the instruction called?
- ▶ What is the AX part of the instruction called?

# 3.1 Software : The Microcomputer Program

- ▶ Assembly language program
  - **Assembly language** program (.asm) file—known as “source code”
  - Converted to **machine code** by a process called “assembling”
  - Assembling performed by a software program — an “8088/8086 assembler”
  - “Machine (object ) code” that can be run on a PC is output in the executable (.exe) file
  - “Source listing” output in (.lst) file—printed and used during execution and debugging of program
- ▶ **DEBUG**—part of “disk operating system (DOS)” of the PC
  - Permits programs to be assembled and disassembled
  - Line-by-line assembler
  - Also permits program to be run and tested
- ▶ **MASM**—Microsoft 80x86 macroassembler
  - Allows a complete program to be assembled in one step



# 3.1 Software : The Microcomputer Program

## ▶ Assembly source program

```
TITLE    BLOCK-MOVE PROGRAM
        PAGE ,132
```

```
COMMENT *This program moves a block of specified number of bytes
from one place to another place*
```

```
;Define constants used in this program
```

```
        N = 16 ;Bytes to be moved
        BLK1ADDR= 100H ;Source block offset address
        BLK2ADDR= 120H ;Destination block offset addr
        DATASEGADDR= 2000H ;Data segment start address
```

```
STACK_SEG      SEGMENT      STACK 'STACK'
                DB          64 DUP(?)
                STACK_SEG    ENDS
```

```
CODE_SEG      SEGMENT      'CODE'
                BLOCK      PROC          FAR
                ASSUME CS:CODE_SEG,SS:STACK_SEG
```

# 3.1 Software : The Microcomputer Program

## ▶ Assembly source program (continued)

;To return to DEBUG program put return address on the stack

```
PUSH    DS
MOV     AX, 0
PUSH    AX
```

;Set up the data segment address

```
MOV     AX, DATASEGADDR
MOV     DS, AX
```

;Set up the source and destination offset addresses

```
MOV     SI, BLK1ADDR
MOV     DI, BLK2ADDR
```

;Set up the count of bytes to be moved

```
MOV     CX, N
```

;Copy source block to destination block

```
NXTPT:  MOV     AH, [SI] ;Move a byte
        MOV     [DI], AH
        INC     SI ;Update pointers
        INC     DI
        DEC     CX ;Update byte counter
        JNZ    NXTPT ;Repeat for next byte
        RET     ;Return to DEBUG program
```

```
BLOCK   ENDP
```

```
CODE_SEG ENDS
```

```
END     BLOCK ;End of program
```

# 3.1 Software : The Microcomputer Program

- ▶ Assembly language must be converted by an *assembler to an equivalent machine language* program for execution by the 8088.
- ▶ A **directive** is a statement that is used to control the translation process of the assembler.

e.g.           DB    64 DUP(?)

- Defines and leaves un-initialized a block of 64 bytes in memory for use as a stack
- ▶ The machine language output produced by the assembler is called *object code*.

# 3.1 Software : The Microcomputer Program

## ▶ Listing of an assembled program

Microsoft (R) Macro Assembler Version 5.10  
BLOCK-MOVE PROGRAM

5/17/92 18:10:04  
Page 1-1

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62

TITLE BLOCK-MOVE PROGRAM

PAGE          ,132
COMMENT *This program moves a block of specified number of bytes
        from one place to another place*

;Define constants used in this program

N=          16          ;Bytes to be moved
BLK1ADDR=   100H        ;Source block offset address
BLK2ADDR=   120H        ;Destination block offset address
DATASEGADDR=1020H      ;Data segment start address

STACK_SEG   SEGMENT    STACK 'STACK'
            DB          64 DUP(?)
            ]

STACK_SEG   ENDS

CODE_SEG    SEGMENT    'CODE'
            BLOCK      PROC    FAR
            ASSUME     CS:CODE_SEG,SS:STACK_SEG

;To return to DEBUG program put return address on the stack

            PUSH      DS
            MOV       AX, 0
            PUSH      AX

;Setup the data segment address

            MOV       AX, DATASEGADDR
            MOV       DS, AX

;Setup the source and destination offset addresses

            MOV       SI, BLK1ADDR
            MOV       DI, BLK2ADDR

;Setup the count of bytes to be moved

            MOV       CX, N

;Copy source block to destination block

NXTPT:MOV    AH, [SI]          ;Move a byte
        MOV    [DI], AH
        INC    SI              ;Update pointers
        INC    DI
        DEC    CX              ;Update byte counter
        JNZ   NXTPT          ;Repeat for next byte
        RET                   ;Return to DEBUG program

            BLOCK      ENDP
            CODE_SEG   ENDS
            END         BLOCK          ;End of program
```

# *3.1 Software : The Microcomputer Program*

- ▶ Listing of the assembled program

e.g. 0013 8A 24 NXTPT: MOV AH, [SI] ; Move a byte

- ▶ Where:

- 0013 = offset address (IP) of first byte of code in the CS
- 8A24 = machine code of the instruction
- NXTPT: = Label
- MOV = instruction mnemonic
- AH = destination operand
- [SI] = source operand in memory

# 3.1 Software : The Microcomputer Program

## ▶ Listing of an assembled program

```

Segments and Groups:
Name          Length  Align  Combine Class
CODE_SEG     001D  PARA  NONE  'CODE'
STACK_SEG    0040  PARA  STACK 'STACK'

Symbols:
Name          Type  Value  Attr
BLK1ADDR     NUMBER 0100
BLK2ADDR     NUMBER 0120
BLOCK        F PROC 0000  CODE_SEG  Length = 001D
DATASEGADDR  NUMBER 1020
N            NUMBER 0010
NXTPT        L NEAR 0013  CODE_SEG

@CPU        TEXT 0101h
@FILENAME    TEXT block
@VERSION     TEXT 510

59 Source Lines
59 Total Lines
15 Symbols

47222 + 347542 Bytes symbol space free

0 Warning Errors
0 Severe Errors

(b)

```

- Other information provided in the listing
  - **Size** of code segment and stack
    - What is the size of the code segment?
    - At what offset address does it begin? End?
  - Names, types, and values of **constants** and variables
    - At what line of the program is the symbol “N” define?
    - What value is it assigned?
    - What is the offset address of the instruction that uses N?
  - # lines and symbols used in the program
    - Why is the value of N given as 0010?
  - # errors that occurred during assembly

# *3.1 Software : The Microcomputer Program*

- ▶ Assembly language versus high-level language
- ▶ It is **easier** to write program with high-level language.
- ▶ Program written in assembly language usually takes up **less memory** space and executes much **faster**.
- ▶ Device **service routines** are usually written in assembly language.
- ▶ Assembly language is used to write those parts of the application that must perform **real-time** operations, and high-level language is used to write those parts that are **not time critical**.

# Memory Models (new slide)

Model	Data	Code	Definition
Tiny*	near		CS=DS=SS
Small	near**	near	DS=SS
Medium	near**	far	DS=SS, multiple code segments
Compact	far	near	single code segment, multiple data segments
Large	far	far	multiple code and data segments
Huge	huge	huge	multiple code and data segments; single array may be >64 KB

\* In the Tiny model, all four segment registers point to the same segment.

\*\* In all models with *near* data pointers, **SS** equals **DS**.

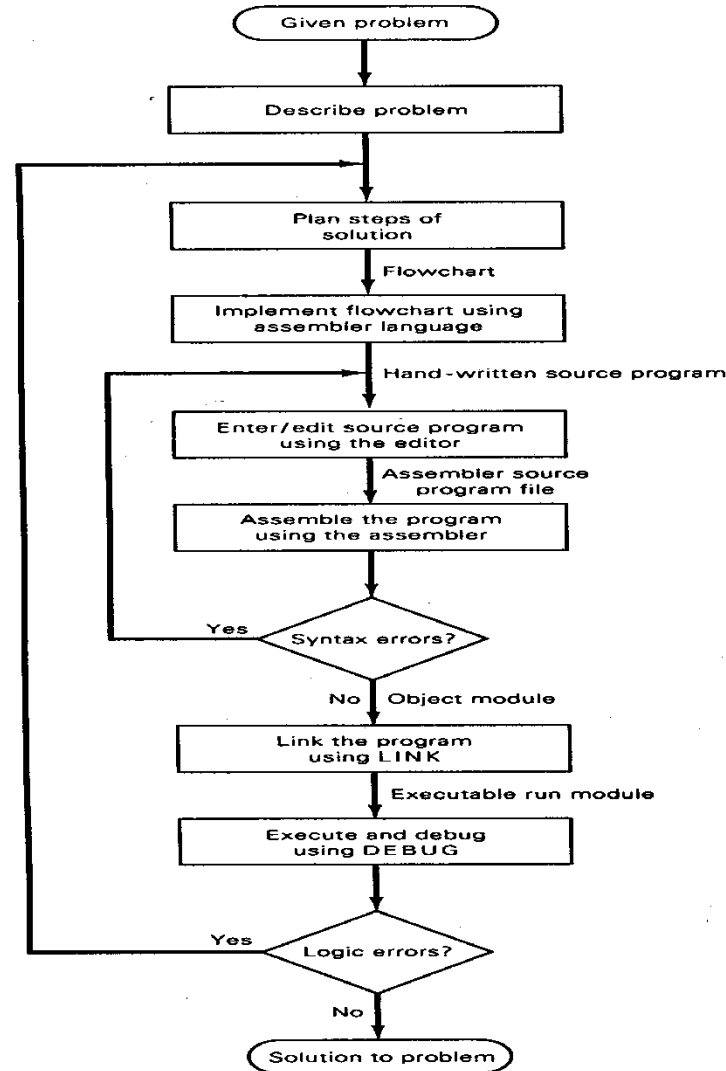


## *3.2 Assembly Language Programming Development on the PC*

- ▶ Describing the problem
- ▶ Planning the solution
- ▶ Coding the solution with assembly language
- ▶ Creating the source program
- ▶ Assembling the source program into an object module
- ▶ Producing a run module
- ▶ Verifying the solution
- ▶ Programs and files involved in the program development cycle

# 3.2 Assembly Language Programming Development on the PC

Program development cycle

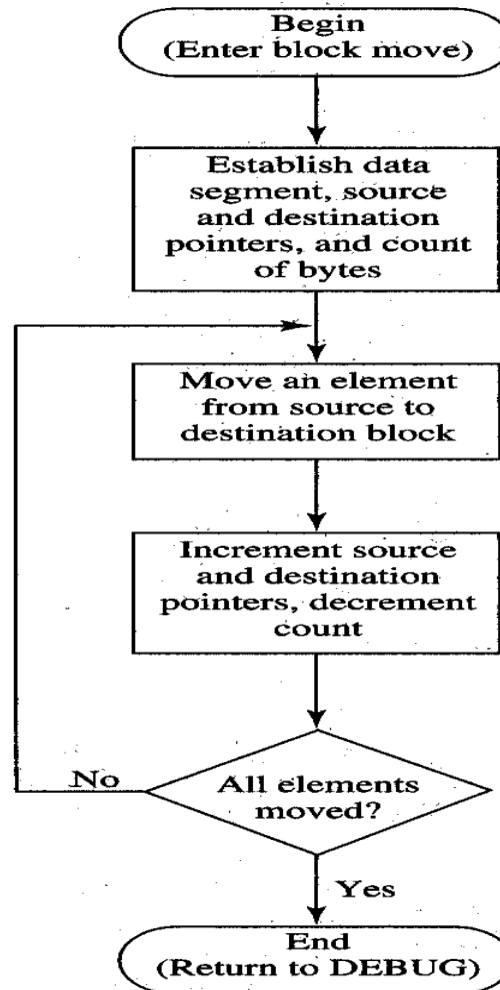


## *3.2 Assembly Language Programming Development on the PC*

- ▶ Describing the problem
  - Most applications are described with a written document called an *application specification*.
- ▶ Planning the solution
  - A *flowchart is an outline that both documents the operations that must be performed by software to implement the planned solution and shows the sequence in which they are performed.*

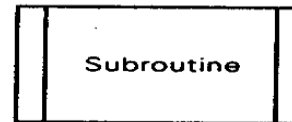
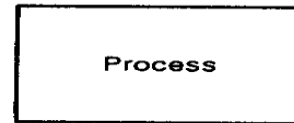
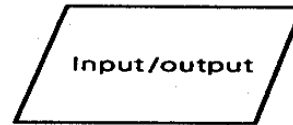
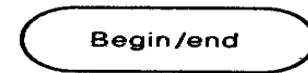
# 3.2 Assembly Language Programming Development on the PC

Flow chart of a block-move program



# 3.2 Assembly Language Programming Development on the PC

Commonly used flowchart symbols



Connection  
within  
a flowchart



Connections  
to another  
flowchart

## *3.2 Assembly Language Programming Development on the PC*

- ▶ Coding the solution with assembly language
  - Two types of statements are used in the source program:
    - The *assembly language instructions* are used to tell the microprocessor what operations are to be performed to implement the application.
    - A *directive* is the instruction to the assembler program used to convert the assembly language program into machine code.

## 3.2 Assembly Language Programming Development on the PC

- ▶ Coding the solution with assembly language
  - The *assembly language instructions*

[Example]

```
MOV  AX, DATASEGMENT
MOV  DS, AX
MOV  SI, BLK1ADDR
MOV  DI, BLK2ADDR
MOV  CX, N
```

- The *directive*

[Example]

```
BLOCK          PROC FAR
or
BLOCK          ENDP
```

## *3.2 Assembly Language Programming Development on the PC*

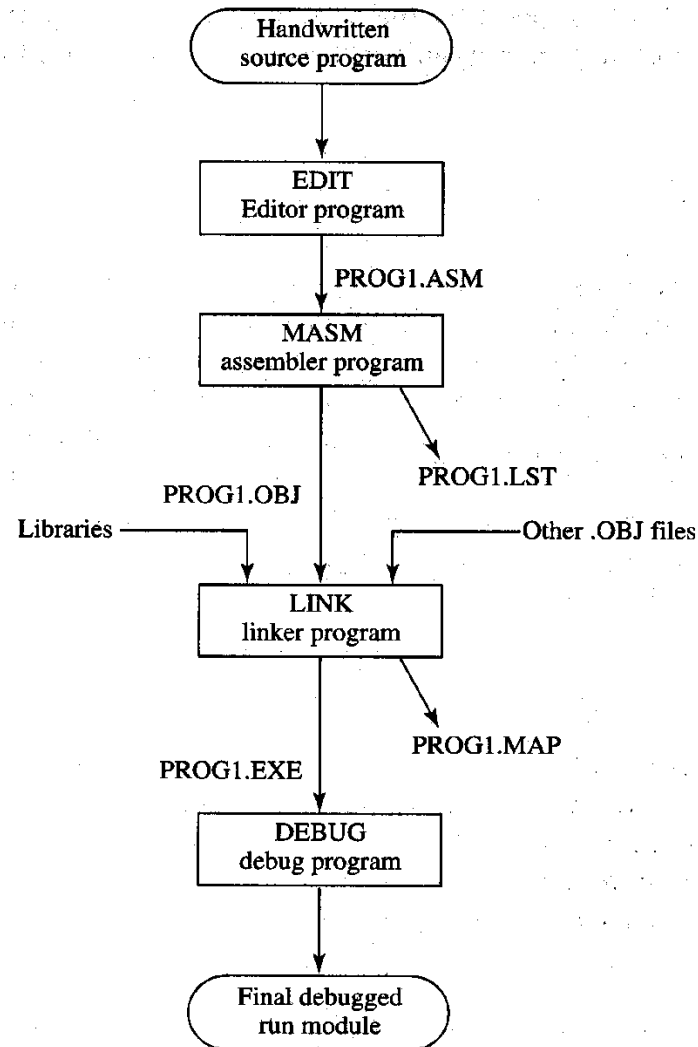
- ▶ **Creating the source program**
  - The EDIT editor
  - The Notepad editor in Windows
  - The Microsoft PWB (Programmer's Work Bench)
- ▶ **Assembling the source program into an object module**
  - The Microsoft MASM assembler
  - The Microsoft PWB (Programmer's Work Bench)
    - Used for writing and compiling code
  - The *assembler source file* and the *object module*



## *3.2 Assembly Language Programming Development on the PC*

- ▶ Producing a run module
  - The object module must be processed by the LINK program to produce an executable *run module*.
- ▶ Verifying the solution
- ▶ Programs and files involved in the program development cycle
  - PROG1.ASM           (Editor)
  - PROG1.OBJ           (Assembler)
  - PROG1.LST           (Assembler)
  - PROG1.EXE           (Linker)
  - PROG1.MAP           (Linker)

# 3.2 Assembly Language Programming Development on the PC



The development programs and users files

## 3.3 The Instruction Set

- ▶ The instruction set of a microprocessor defines the **basic operations** that a programmer can specify to the device to perform
- ▶ Instruction set groups
  - Data transfer instructions (moving data between registers and/or memory locations).
  - Arithmetic instructions (ADD,SUB,DIV,MUL,INC,DEC)
  - Logic instructions (AND,OR,XOR,ROL,NOT)
  - String manipulation instructions (REP,MOVS,LODS,STDS)
  - Control transfer instructions (JMP,RET,LOOP)
  - Processor control instructions (CLC,CMC,STC,HLT,)

## 3.3 The Instruction Set

- ▶ In assembly language each instruction is represented by a “**mnemonic**” that describes its operation and is called its “operation code (**opcode**)”
  - MOV = move → data transfer
  - ADD = add → arithmetic
  - JMP = unconditional jump → control transfer
- ▶ **Operands**: Identify whether the elements of data to be processed are in **registers** or **memory**
  - Source operand– location of one operand to be processed
  - Destination operand—location of the other operand to be processed and the location of the result

# 3.3 The Instruction Set

## ▶ Data transfer instructions

Mnemonic and Description	Instruction Code			
<b>DATA TRANSFER</b>				
<b>MOV – Move:</b>	<b>7 6 5 4 3 2 1 0</b>	<b>7 6 5 4 3 2 1 0</b>	<b>7 6 5 4 3 2 1 0</b>	<b>7 6 5 4 3 2 1 0</b>
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
<b>PUSH – Push:</b>				
Register/Memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment Register	0 0 0 reg 1 1 0			
<b>POP – Pop:</b>				
Register/Memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment Register	0 0 0 reg 1 1 1			
<b>XCHG – Exchange:</b>				
Register/Memory with Register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with Accumulator	1 0 0 1 0 reg			

## 3.3 The Instruction Set

### ▶ Data transfer instructions

Mnemonic and Description	Instruction Code	
<b>DATA TRANSFER</b>		
<b>IN – Input from:</b>		
Fixed Port	1110010w	port
Variable Port	1110110w	
<b>OUT – Output to:</b>		
Fixed Port	1110011w	port
Variable Port	1110111w	
<b>XLAT</b> – Translate Byte to AL	11010111	
<b>LEA</b> – Load EA to Register	10001101	mod reg r/m
<b>LDS</b> – Load Pointer to DS	11000101	mod reg r/m
<b>LES</b> – Load Pointer to ES	11000100	mod reg r/m
<b>LAHF</b> – Load AH with Flags	10011111	
<b>SAHF</b> – Store AH into Flags	10011110	
<b>PUSHF</b> – Push Flags	10011100	
<b>POPF</b> – Pop Flags	10011101	

# 3.3 The Instruction Set

## ▶ Arithmetic instructions

Mnemonic and Description	Instruction Code			
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
<b>ARITHMETIC</b>				
<b>ADD – Add:</b>				
Reg./Memory with Register to Either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s: w – 01
Immediate to Accumulator	0 0 0 0 0 1 0 w	data	data if w – 1	
<b>ADC – Add with Carry:</b>				
Reg./Memory with Register to Either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s: w – 01
Immediate to Accumulator	0 0 0 1 0 1 0 w	data	data if w – 1	
<b>INC – Increment:</b>				
Register/Memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
<b>AAA – ASCII Adjust for Add</b>	0 0 1 1 0 1 1 1			
<b>BAA – Decimal Adjust for Add</b>	0 0 1 0 0 1 1 1			
<b>SUB – Subtract:</b>				
Reg./Memory and Register to Either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s: w – 01
Immediate from Accumulator	0 0 1 0 1 1 0 w	data	data if w – 1	
<b>SSB – Subtract with Borrow</b>				
Reg./Memory and Register to Either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s: w – 01
Immediate from Accumulator	0 0 0 1 1 1 w	data	data if w – 1	

# 3.3 The Instruction Set

## ▶ Arithmetic instructions

Mnemonic and Description	Instruction Code			
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
<b>ARITHMETIC</b>				
<b>DEC – Decrement:</b>				
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
<b>NEG – Change sign</b>	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
<b>CMP – Compare:</b>				
Register/Memory and Register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with Register/Memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w = 01
Immediate with Accumulator	0 0 1 1 1 1 0 w	data	data if w = 1	
<b>AAS – ASCII Adjust for Subtract</b>	0 0 1 1 1 1 1 1			
<b>DAS – Decimal Adjust for Subtract</b>	0 0 1 0 1 1 1 1			
<b>MUL – Multiply (Unsigned)</b>	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
<b>IMUL – Integer Multiply (Signed)</b>	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
<b>AAM – ASCII Adjust for Multiply</b>	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
<b>DIV – Divide (Unsigned)</b>	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
<b>IDIV – Integer Divide (Signed)</b>	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
<b>AAD – ASCII Adjust for Divide</b>	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
<b>CBW – Convert Byte to Word</b>	1 0 0 1 1 0 0 0			
<b>CWD – Convert Word to Double Word</b>	1 0 0 1 1 0 0 1			



# 3.3 The Instruction Set

## ► Logic instructions

Mnemonic and Description	Instruction Code			
<b>LOGIC</b>	<b>7 6 5 4 3 2 1 0</b>	<b>7 6 5 4 3 2 1 0</b>	<b>7 6 5 4 3 2 1 0</b>	<b>7 6 5 4 3 2 1 0</b>
<b>NOT</b> – Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
<b>SHL/SAL</b> – Shift Logical/Arithmetic Left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
<b>SHR</b> – Shift Logical Right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
<b>SAR</b> – Shift Arithmetic Right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
<b>ROL</b> – Rotate Left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
<b>ROR</b> – Rotate Right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
<b>RCL</b> – Rotate Through Carry Flag Left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
<b>RCR</b> – Rotate Through Carry Right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
<b>AND</b> – And:				
Reg./Memory and Register to Either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 0 0 1 0 w	data	data if w = 1	
<b>TEST</b> – And Function to Flags, No Result:				
Register/Memory and Register	1 0 0 0 1 0 w	mod reg r/m		
Immediate Data and Register/Memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate Data and Accumulator	1 0 1 0 1 0 0 w	data	data if w = 1	
<b>OR</b> – Or:				
Reg./Memory and Register to Either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w = 1
Immediate to Accumulator	0 0 0 0 1 1 0 w	data	data if w = 1	
<b>XOR</b> – Exclusive or:				
Reg./Memory and Register to Either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 1 0 1 0 w	data	data if w = 1	

## 3.3 The Instruction Set

- ▶ String manipulation instructions

Mnemonic and Description	Instruction Code
<b>STRING MANIPULATION</b>	
REP – Repeat	1111001z
MOVS – Move Byte/Word	1010010w
CMPS – Compare Byte/Word	1010011w
SCAS – Scan Byte/Word	1010111w
LODS – Load Byte/Wd to AL/AX	1010110w
STOS – Stor Byte/Wd from AL/A	1010101w

# 3.3 The Instruction Set

## ▶ Control transfer instructions

Mnemonic and Description	Instruction Code		
<b>CONTROL TRANSFER</b>			
<b>CALL – Call:</b>			
Direct within Segment	1 1 1 0 1 0 0 0	disp-low	disp-high
Indirect within Segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	
Direct Intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	
<b>JMP – Unconditional Jump:</b>			
Direct within Segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within Segment-Short	1 1 1 0 1 0 1 1	disp	
Indirect within Segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct Intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	

## 3.3 The Instruction Set

### ▶ Control transfer instructions

Mnemonic and Description	Instruction Code		
<b>RET – Return from CALL:</b>			
Within Segment	1 1 0 0 0 0 1 1		
Within Seg Adding Immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment Adding Immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
<b>JE/JZ – Jump on Equal/Zero</b>	0 1 1 1 0 1 0 0	disp	
<b>JL/JNGE – Jump on Less/Not Greater or Equal</b>	0 1 1 1 1 1 0 0	disp	
<b>JLE/JNG – Jump on Less or Equal/Not Greater</b>	0 1 1 1 1 1 1 0	disp	
<b>JB/JNAE – Jump on Below/Not Above or Equal</b>	0 1 1 1 0 0 1 0	disp	
<b>JBE/JNA – Jump on Below or Equal/Not Above</b>	0 1 1 1 0 1 1 0	disp	
<b>JP/JPE – Jump on Parity/Parity Even</b>	0 1 1 1 1 0 1 0	disp	
<b>JO – Jump on Overflow</b>	0 1 1 1 0 0 0 0	disp	
<b>JS – Jump on Sign</b>	0 1 1 1 1 0 0 0	disp	
<b>JNE/JNZ – Jump on Not Equal/Not Zero</b>	0 1 1 1 0 1 0 1	disp	
<b>JNL/JGE – Jump on Not Less/Greater or Equal</b>	0 1 1 1 1 1 0 1	disp	
<b>JNLE/JG – Jump on Not Less or Equal/Greater</b>	0 1 1 1 1 1 1 1	disp	

## 3.3 The Instruction Set

### ▶ Control transfer instructions

Mnemonic and Description	Instruction Code	
<b>JNB/JAE</b> – Jump on Not Below/Above or Equal	0 1 1 1 0 0 1 1	disp
<b>JNBE/JA</b> – Jump on Not Below or Equal/Above	0 1 1 1 0 1 1 1	disp
<b>JNP/JPO</b> – Jump on Not Par/Par Odd	0 1 1 1 1 0 1 1	disp
<b>JNO</b> – Jump on Not Overflow	0 1 1 1 0 0 0 1	disp
<b>JNS</b> – Jump on Not Sign	0 1 1 1 1 0 0 1	disp
<b>LOOP</b> – Loop CX Times	1 1 1 0 0 0 1 0	disp
<b>LOOPZ/LOOPE</b> – Loop While Zero/Equal	1 1 1 0 0 0 0 1	disp
<b>LOOPNZ/LOOPNE</b> – Loop While Not Zero/Equal	1 1 1 0 0 0 0 0	disp
<b>JCXZ</b> – Jump on CX Zero	1 1 1 0 0 0 1 1	disp
<b>INT – Interrupt</b>		
Type Specified	1 1 0 0 1 1 0 1	type
Type 3	1 1 0 0 1 1 0 0	
<b>INTO</b> – Interrupt on Overflow	1 1 0 0 1 1 1 0	
<b>IRET</b> – Interrupt Return	1 1 0 0 1 1 1 1	

## 3.3 The Instruction Set

### ▶ Process control instructions

Mnemonic and Description	Instruction Code	
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
<b>PROCESSOR CONTROL</b>		
<b>CLC</b> – Clear Carry	1 1 1 1 1 0 0 0	
<b>CMC</b> – Complement Carry	1 1 1 1 0 1 0 1	
<b>STC</b> – Set Carry	1 1 1 1 1 0 0 1	
<b>CLD</b> – Clear Direction	1 1 1 1 1 1 0 0	
<b>STD</b> – Set Direction	1 1 1 1 1 1 0 1	
<b>CLI</b> – Clear Interrupt	1 1 1 1 1 0 1 0	
<b>STI</b> – Set Interrupt	1 1 1 1 1 0 1 1	
<b>HLT</b> – Halt	1 1 1 1 0 1 0 0	
<b>WAIT</b> – Wait	1 0 0 1 1 0 1 1	
<b>ESC</b> – Escape (to External Device)	1 1 0 1 1 x x x	mod x x x r/m
<b>LOCK</b> – Bus Lock Prefix	1 1 1 1 0 0 0 0	

## 3.4 The MOV Instruction

- ▶ The move (MOV) instruction is used to transfer a byte or a word of data from a source operand to a destination operand.
- ▶ e.g. **MOV DX, CS**  
**MOV [SUM], AX**

• Note that the MOV instruction cannot transfer data directly between external memory.

Mnemonic	Meaning	Format	Operation	Flags affected
MOV	Move	MOV D,S	(S) → (D)	None

(a)

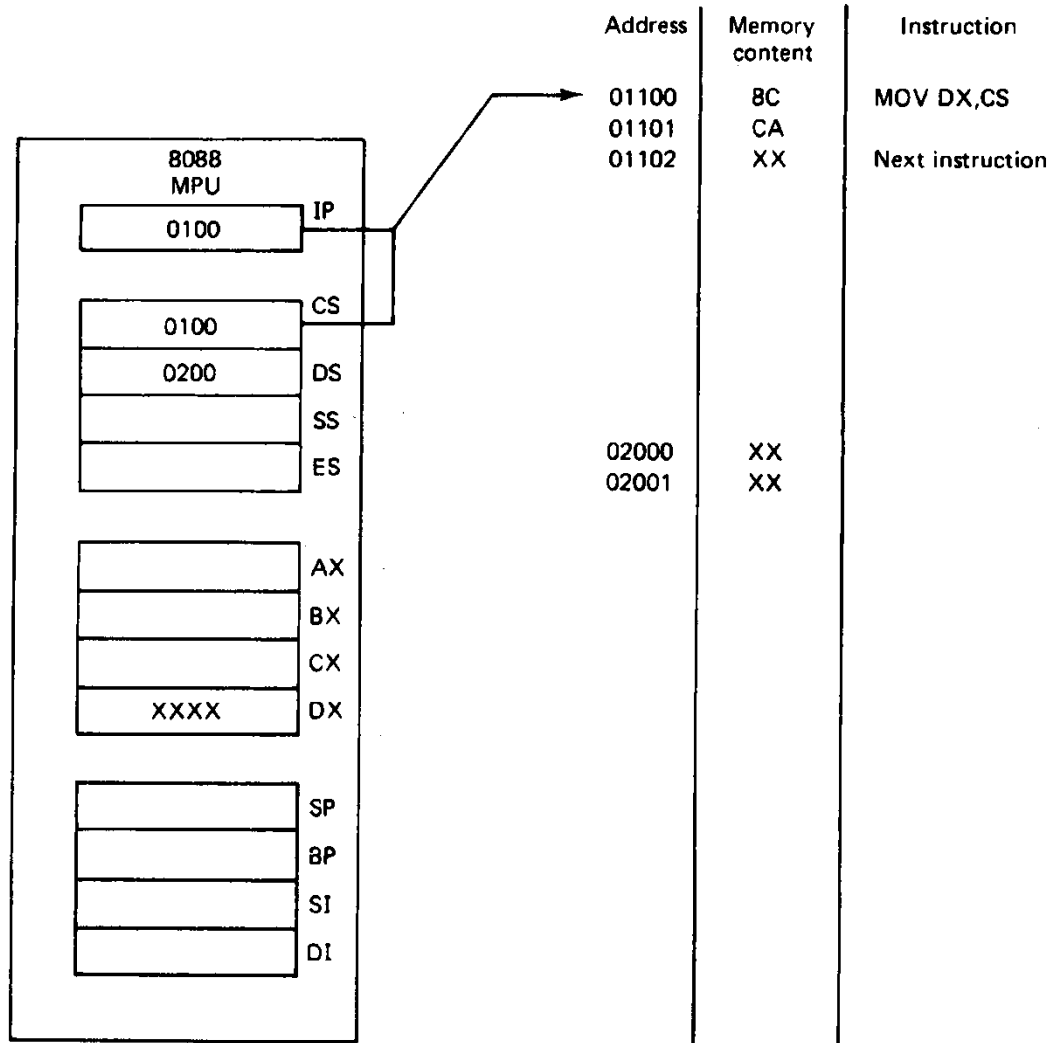
Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg-reg	Reg16
Seg-reg	Mem16
Reg16	Seg-reg
Memory	Seg-reg

(b)

Allowed operands for MOV instruction

# 3.4 The MOV Instruction

▶ MOV DX, CS



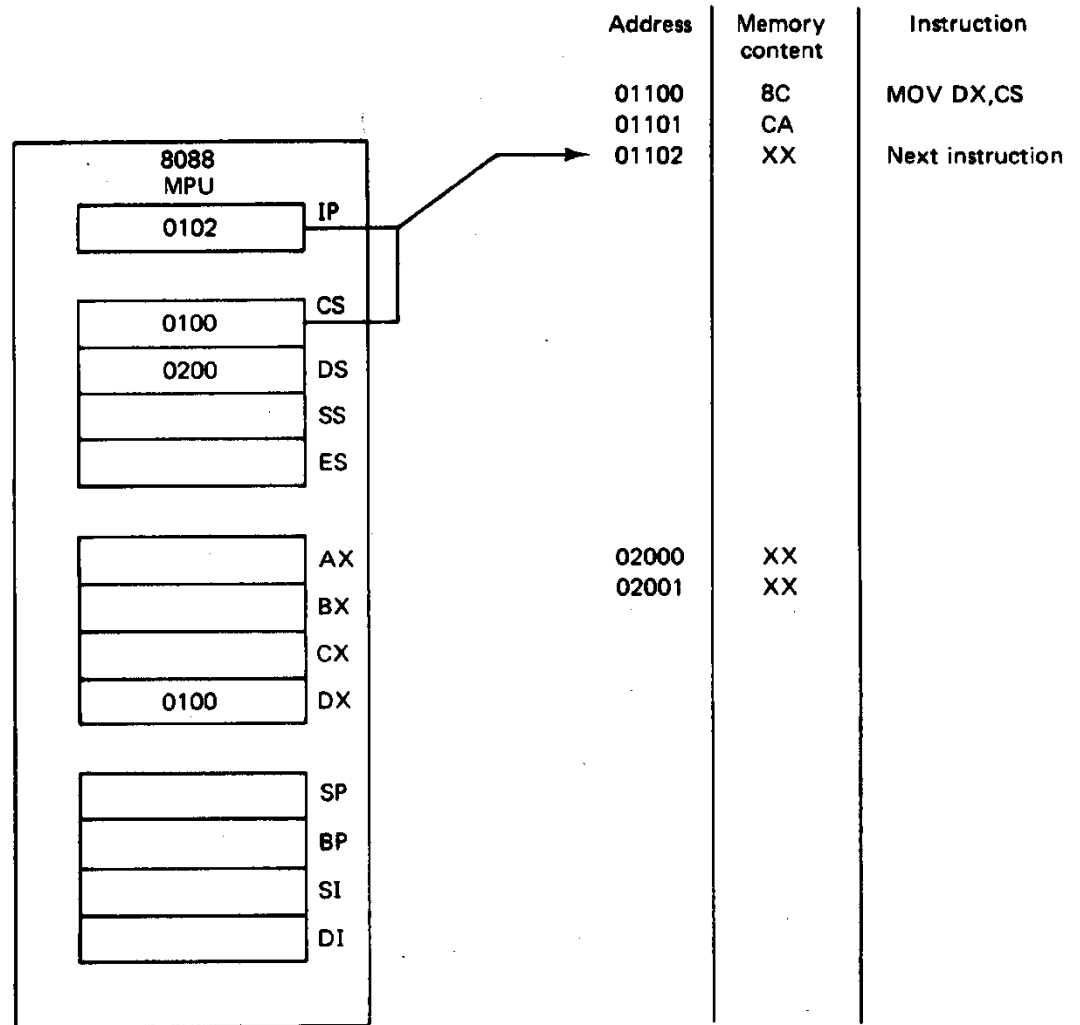
Before execution

(c)



# 3.4 The MOV Instruction

▶ MOV DX, CS



(d)

After execution

## *3.5 Addressing Modes*

- ▶ Addressing mode is a method of specifying an **operand** & categorized into three types:
  - **Register** operand addressing mode
  - **Immediate** operand addressing mode
  - **Memory** operand addressing mode
    - **Direct** addressing mode
    - **Register indirect** addressing mode
    - **Based** addressing mode
    - **Indexed** addressing mode
    - **Based-indexed** addressing mode

# 3.5 Addressing Modes

- ▶ **Register** operand addressing mode:

The operand to be accessed is specified as residing in an internal register of 8088.

e.g. **MOV AX, BX**

- ▶ Only the **data registers** can be accessed as bytes or words
  - Ex. AL,AH → bytes
  - AX → word
- ▶ **Index** and **pointer** registers as words
  - Ex. SI → word pointer
- ▶ **Segment** registers only as words
  - Ex. DS → word pointer

Register	Operand sizes	
	Byte (Reg 8)	Word (Reg 16)
Accumulator	AL, AH	AX
Base	BL, BH	BX
Count	CL, CH	CX
Data	DL, DH	DX
Stack pointer	—	SP
Base pointer	—	BP
Source index	—	SI
Destination index	—	DI
Code segment	—	CS
Data segment	—	DS
Stack segment	—	SS
Extra segment	—	ES

# 3.5 Addressing Modes

## ▶ Register operand addressing mode

### • Example

**MOV AX,BX**

Source = BX → word data

Destination = AX → word data

Operation: (BX) → (AX)

### • State before fetch and execution

CS:IP = 0100:0000 = 01000H

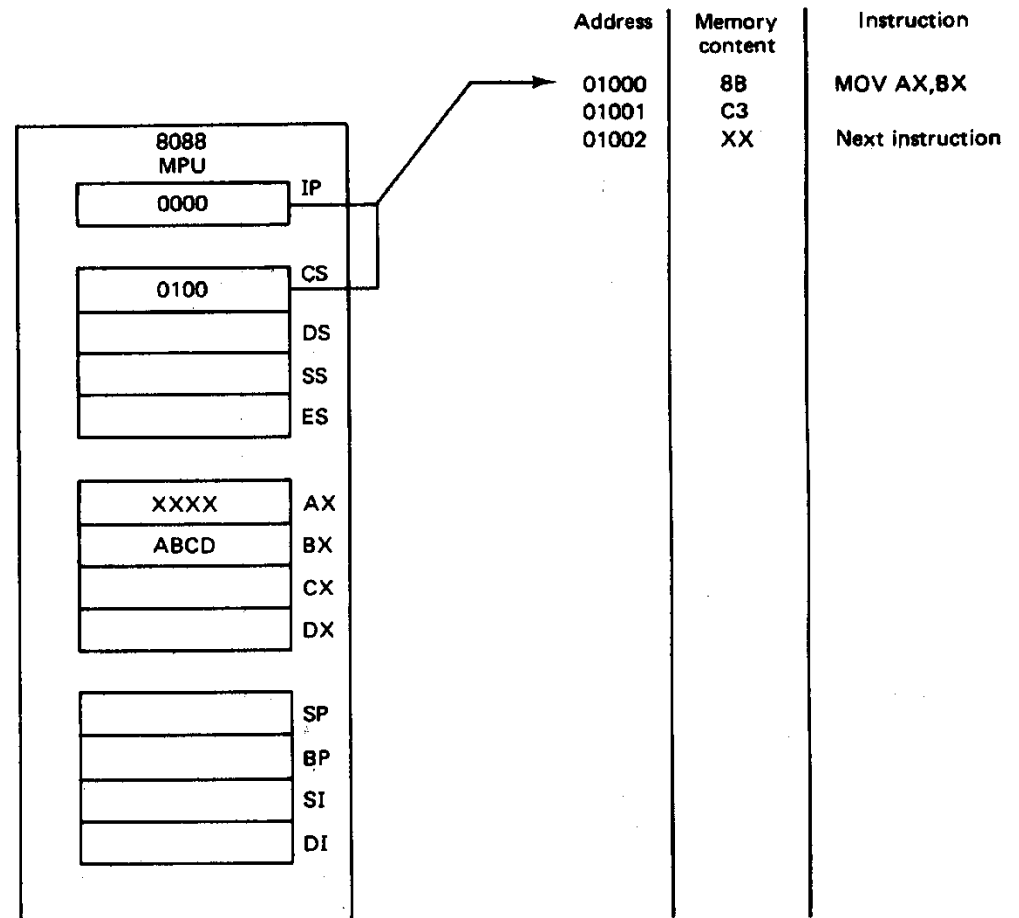
Move instruction code = 8BC3H

(01000H) = 8BH

(01001H) = C3H

(BX) = ABCDH

(AX) = XXXX → don't care state



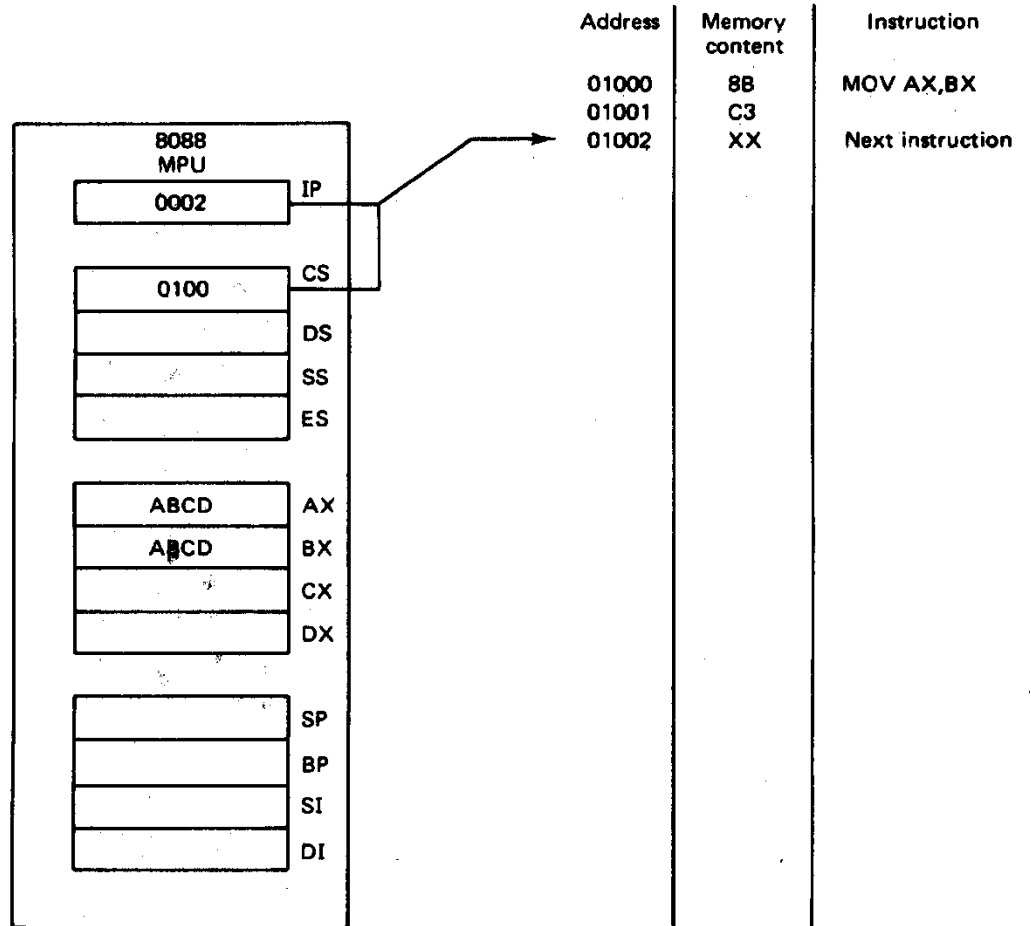
Before execution

(a)

# 3.5 Addressing Modes

## ▶ Register operand addressing mode

- Example (continued)
- State after execution  
CS:IP = 0100:0002 = 01002H  
01002H → points to next sequential instruction  
(BX) = ABCDH  
(AX) = ABCDH → Value in BX copied into AX



After execution

(b)

## 3.5 Addressing Modes

- ▶ **Immediate** operand addressing mode
  - Operand is coded as **part of the instruction**
  - Applies only to the **source** operand
  - **Destination** operand uses **register** addressing mode
- ▶ **Types**
  - **Imm8** = 8-bit immediate operand
  - **Imm16** = 16-bit immediate operand
  - General instruction structure and operation

**MOV Rx,ImmX**  
**ImmX → (Rx)**

Before execution

# 3.5 Addressing Modes

## ▶ Immediate operand addressing mode

- Example

**MOV AL,15H**

Source = Imm8 → immediate byte data

Destination = AL → Byte of data

• Operation: (Imm8) → (AL)

• State before fetch and execution

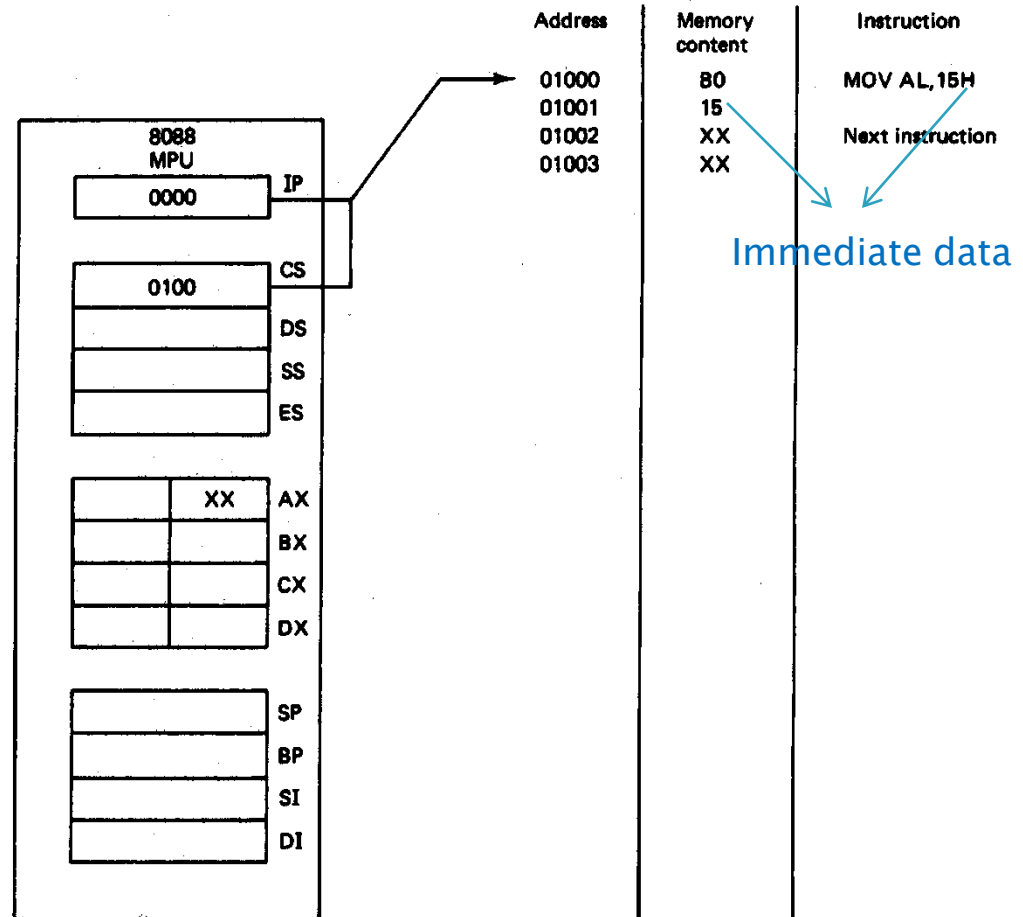
CS:IP = 0100:0000 = 01000H

Move instruction code = B015H

(01000H) = B0H

(01001H) = 15H → Immediate data

What about MOV AL,1515H ?



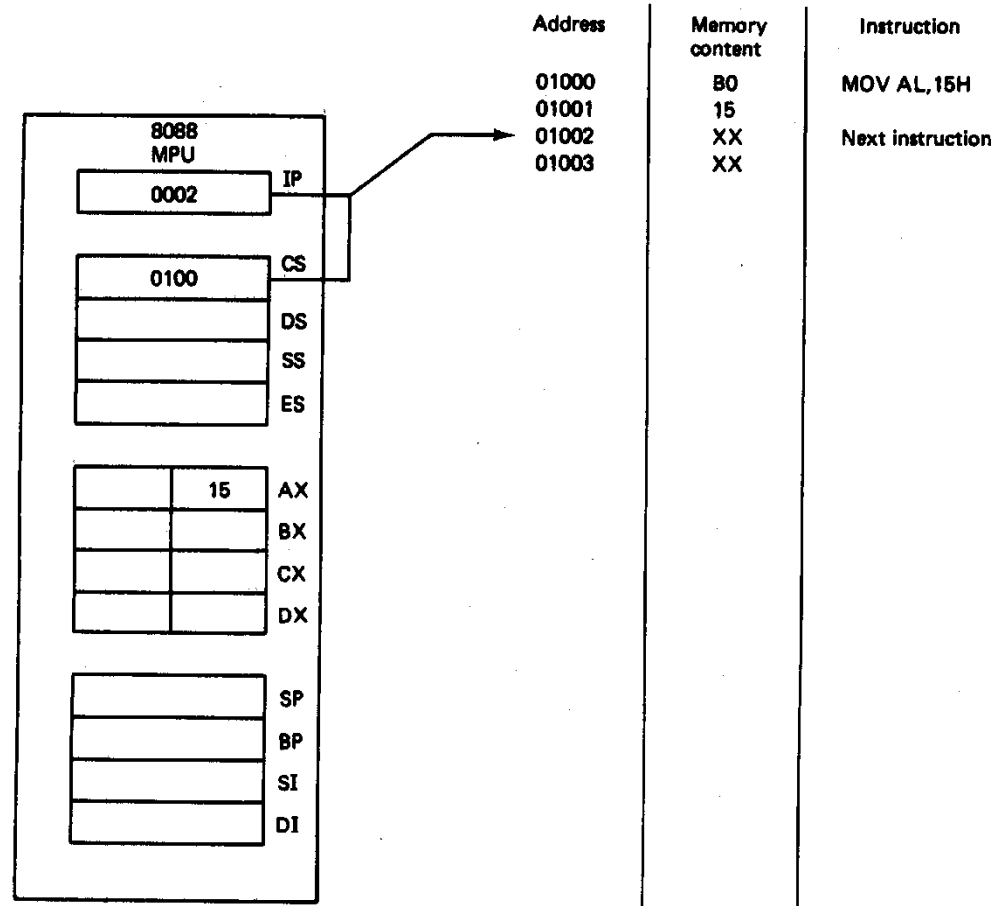
(a)

Before execution

# 3.5 Addressing Modes

## ▶ Immediate operand addressing mode

- Example (continued)
  - State after execution  
(AH) = XX → don't care state
- What about MOV AL,1515H ?



(b)

After execution



# 3.5 Addressing Modes

## ▶ Memory addressing modes

To reference an operand in memory, the 8088 must calculate the physical address (PA) of the operand and then initiate a read or write operation to this storage location.

$$\text{Physical Address (PA)} = \text{Segment Base Address (SBA)} + \text{Effective Address (EA)}$$

$$\text{PA} = \text{SBA} : \text{EA}$$

$$\text{PA} = \text{Segment base} : \text{Base} + \text{Index} + \text{Displacement}$$

$$\text{PA} = \left\{ \begin{array}{c} \text{CS} \\ \text{SS} \\ \text{DS} \\ \text{ES} \end{array} \right\} : \left\{ \begin{array}{c} \text{BX} \\ \text{BP} \end{array} \right\} + \left\{ \begin{array}{c} \text{SI} \\ \text{DI} \end{array} \right\} + \left\{ \begin{array}{c} \text{8-bit displacement} \\ \text{16-bit displacement} \end{array} \right\}$$

$$\text{EA} = \text{Base} + \text{Index} + \text{Displacement}$$

Physical and effective address computation for memory operands

### Where:

SBA = Segment base address

EA = Effective address (offset)

#### • Components of a effective address

- Base → base registers BX or BP
- Index → index register SI or DI
- Displacement → 8 or 16-bit displacement
- Not all elements are used in all computations—results in a variety of addressing modes

# Memory addressing modes

Effective address (EA)

Memory addressing mode	Example	Base	Index	Disp.
Direct	MOV CX,[1234]			✓
Indirect	MOV AX,[SI]	✓		
Based	MOV [BX]+ 1234H,AL	✓		✓
Indexed	MOV AL,[SI]+1234H		✓	✓
Based-Index	MOV AH, [BX][SI]+1234H	✓	✓	✓

$$PA = SBA + EA$$

$$EA = \text{Base} + \text{Index} + \text{Displacement}$$

# 3.5 Addressing Modes

- ▶ Memory addressing modes – **Direct addressing mode**

PA = Segment base: Direct address

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \text{Direct address} \right\}$$

The default segment register is DS  
Computation of a direct  
memory address

e.g. `MOV AX, [1234H]`

- Similar to immediate addressing in that information **coded directly** into the instruction

- Immediate information is the **effective address** called the direct address

- Physical address computation

PA = SBA:EA → 20-bit address

PA = SBA:[DA] → immediate 8-bit or 16 bit displacement

[DA]: Displacement Address

- Segment base address is **DS** by default

$$PA = DS:[DA]$$

- Segment override prefix (**SEG**) is required to enable use of another segment register

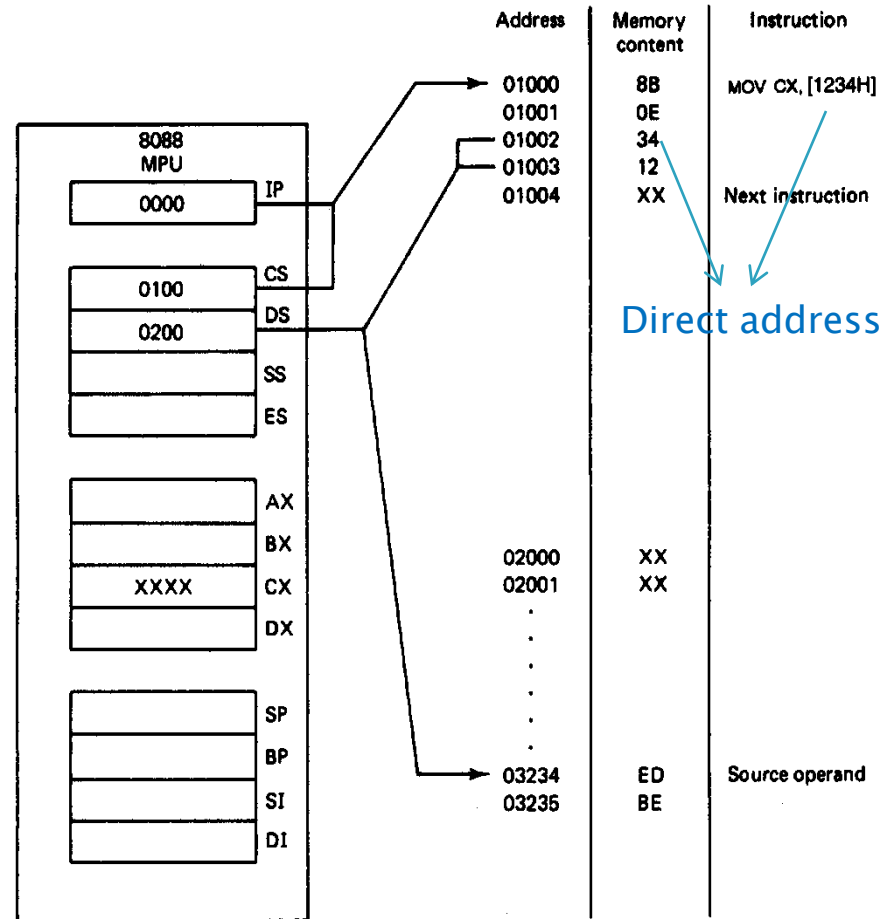
$$PA = SEG:ES:[DA]$$

# 3.5 Addressing Modes

- ▶ Memory addressing modes – Direct addressing mode
- Example

**MOV CX,[1234H]**

- State before fetch and execution
  - Instruction
- CS = 0100H  
 IP = 0000H  
 CS:IP = 0100:0000H = 01000H  
 (01000H,01001H) = Opcode = 8B0E  
 (01003H,01002) = DA = 1234H
- Source operand—direct addressing
- DS = 0200H  
 DA = 1234H  
 PA = DS:DA = 0200H:1234H  
 = 02000H+1234H = 03234H  
 (03235H,03234H) = BEEDH
- Destination operand--register addressing
- (CX) = XXXX → don't care state



Before execution (a)

# 3.5 Addressing Modes

## ▶ Memory addressing modes – Direct addressing mode

- Example (continued)
- State after execution

Instruction

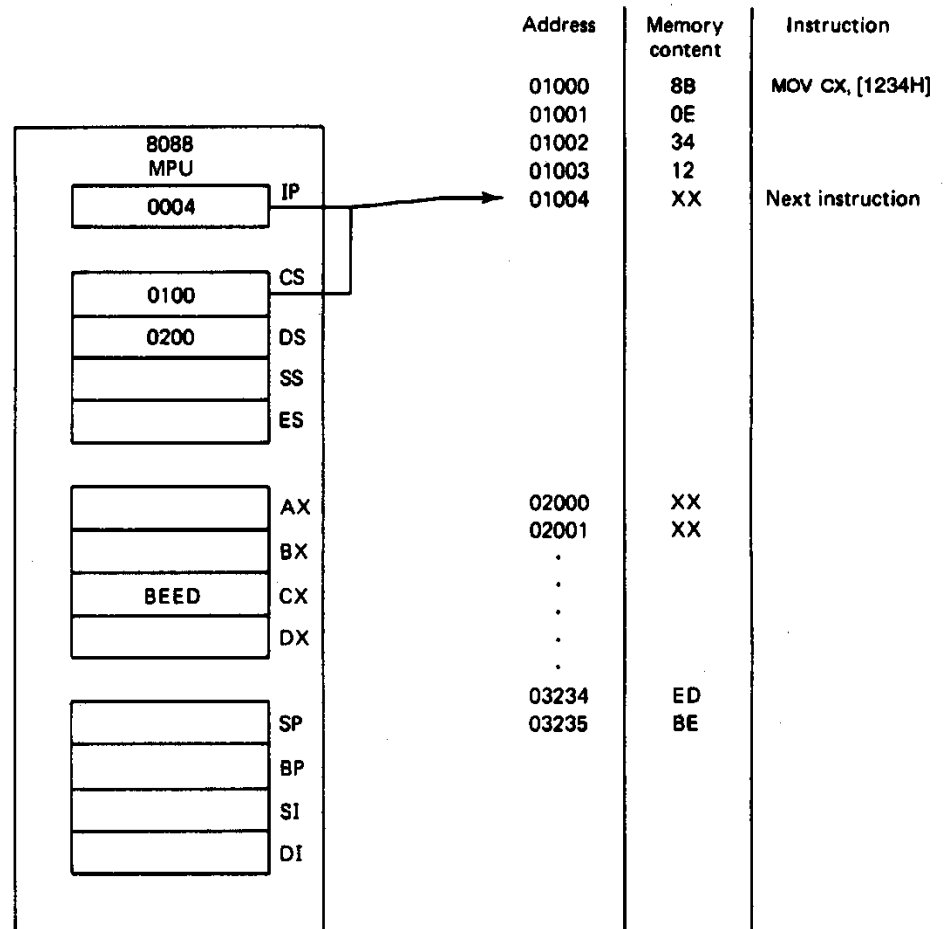
CS:IP = 0100:0004 = 01004H

01004H → points to next

Sequential instruction

• Source operand  
 (03235H,03234H) = BEEDH →  
 unchanged

• Destination operand  
 (CX) = BEEDH



(b)

After execution

# 3.5 Addressing Modes

- ▶ Memory addressing modes – **Register indirect addressing mode**

PA = Segment Base : Direct Address

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \begin{array}{c} BX \\ BP \\ SI \\ DI \end{array} \right\}$$

The default segment register is DS  
Computation of an indirect memory address

e.g. **MOV AX, [SI]**

- Similar to direct addressing in that the effective address is combined with the contents of DS to obtain the physical address

- Effective address resides in either a **base** or **index** register

- Physical address computation

PA = SBA:EA → 20-bit address

PA = SBA:[Rx] → 16-bit offset

- Segment base address is **DS** by default for BX, SI, and DI

- Segment base address is **SS** by default for BP

PA = DS:[Rx]

- Segment override prefix (SEG) is required to enable use of another segment register

PA = SEG:ES:[Rx]

# 3.5 Addressing Modes

## Memory addressing modes – Register indirect addressing mode

$$PA = 02000_{16} + 1234_{16} = 03234_{16}$$

### • Example

**MOV AX,[SI]**

#### • State before fetch and execution Instruction

CS = 0100H

IP = 0000H

CS:IP = 0100:0000H = 01000H

(01000H,01001H) = Opcode = 8B04H

#### • Source operand–register indirect addressing

DS = 0200H

SI = 1234H

PA = DS:SI = 0200H:1234H

= 02000H + 1234H

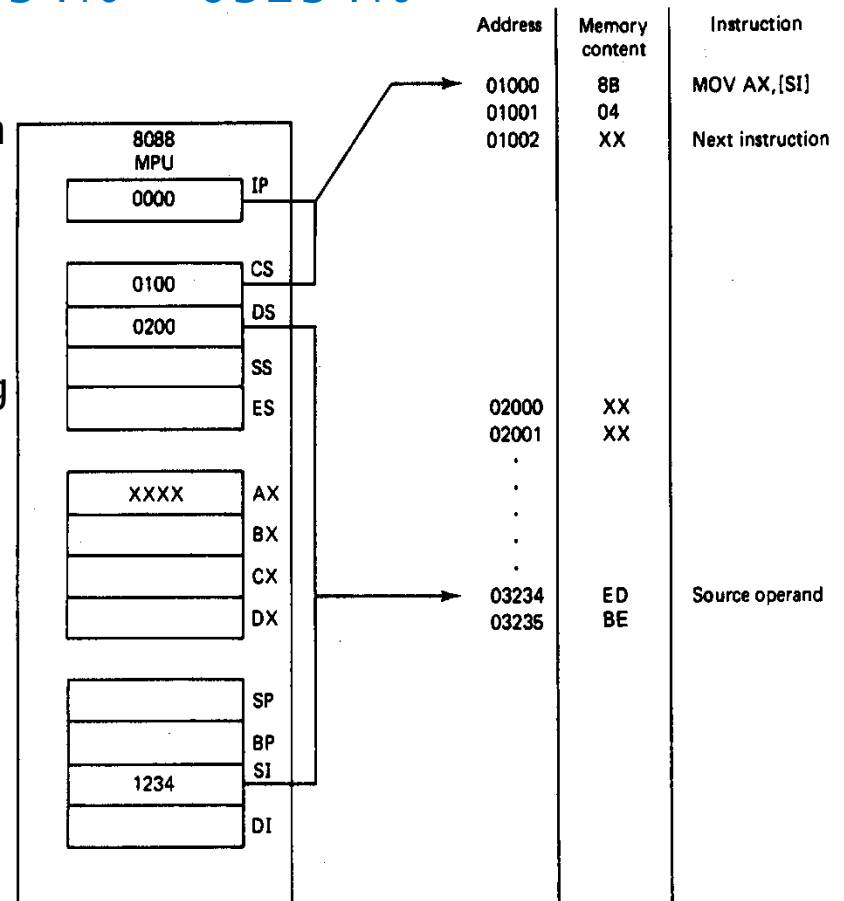
= 03234H

(03235H,03234H) = BEEDH

#### • Destination operand–register operand addressing

(AX) = XXXX

→ don't care state



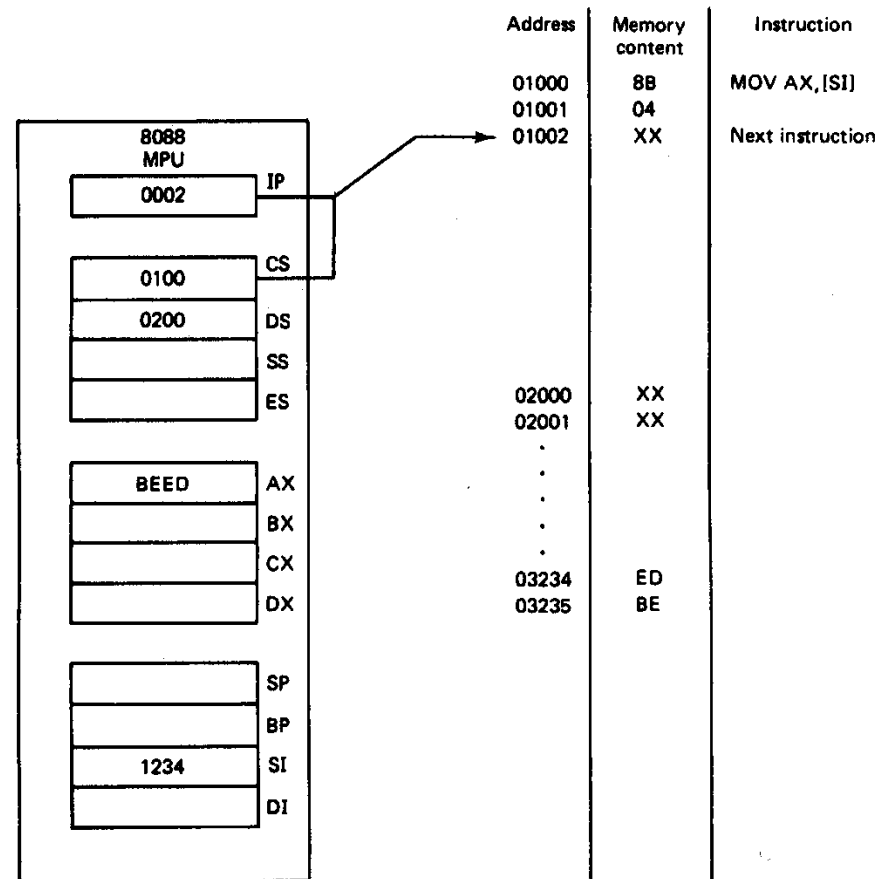
Before execution<sup>(a)</sup>  
CPE 0408330

# 3.5 Addressing Modes

- Memory addressing modes – Register indirect addressing mode

$$PA = 02000_{16} + 1234_{16} = 03234_{16}$$

- Example (continued)
  - State after execution Instruction  
 $CS:IP = 0100:0002 = 01002H$   
 $01002H \rightarrow$  points to next sequential instruction
  - Source operand  
 $(03235H, 03234H) = BEEDH \rightarrow$  unchanged
  - Destination operand  
 $(AX) = BEEDH$



(b)

After execution

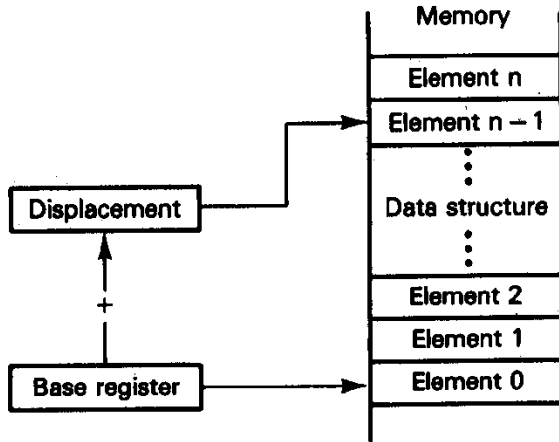


# 3.5 Addressing Modes

## ▶ Memory addressing modes – Based addressing mode

$$PA = \left\{ \begin{matrix} CS \\ DS \\ SS \\ ES \end{matrix} \right\} : \left\{ \begin{matrix} BX \\ BP \end{matrix} \right\} + \left\{ \begin{matrix} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{matrix} \right\}$$

(a)



(b)

- Effective address formed from contents of a base register and a displacement
- Base register is either BX or BP (stack)
  - Direct/indirect displacement is 8-bit or 16bit
- Physical address computation

$PA = SBA:EA \rightarrow 20\text{-bit address}$

$PA = SBA:[BX \text{ or } BP] + DA$

- Accessing a data structure
  - Based addressing makes it easy to access elements of data in an **array**
  - Address in base register points to **start** of the array
  - Displacement selects the element **within** the array
  - Value of the displacement is simply changed to access **another element** in the array
  - Program changes value in base register to select **another array**

e.g. `MOV [BX]+1234H, AL`

# 3.5 Addressing Modes

- Memory addressing modes – Based addressing mode

$$PA = 02000_{16} + 1000_{16} + 1234_{16} = 04234_{16}$$

- Example

**MOV [BX] +1234H,AL**

- State before fetch and execution Instruction

CS = 0100H, IP = 0000H

CS:IP = 0100:0000H = 01000H

(01000H,01001H) = Opcode = 8887H

(01002H,01003H) = Direct displacement = 1234H

- Destination operand—based addressing

DS = 0200H, BX = 1000H, DA = 1234H

PA = DS:DS+DA = 0200H:1000H+1234H

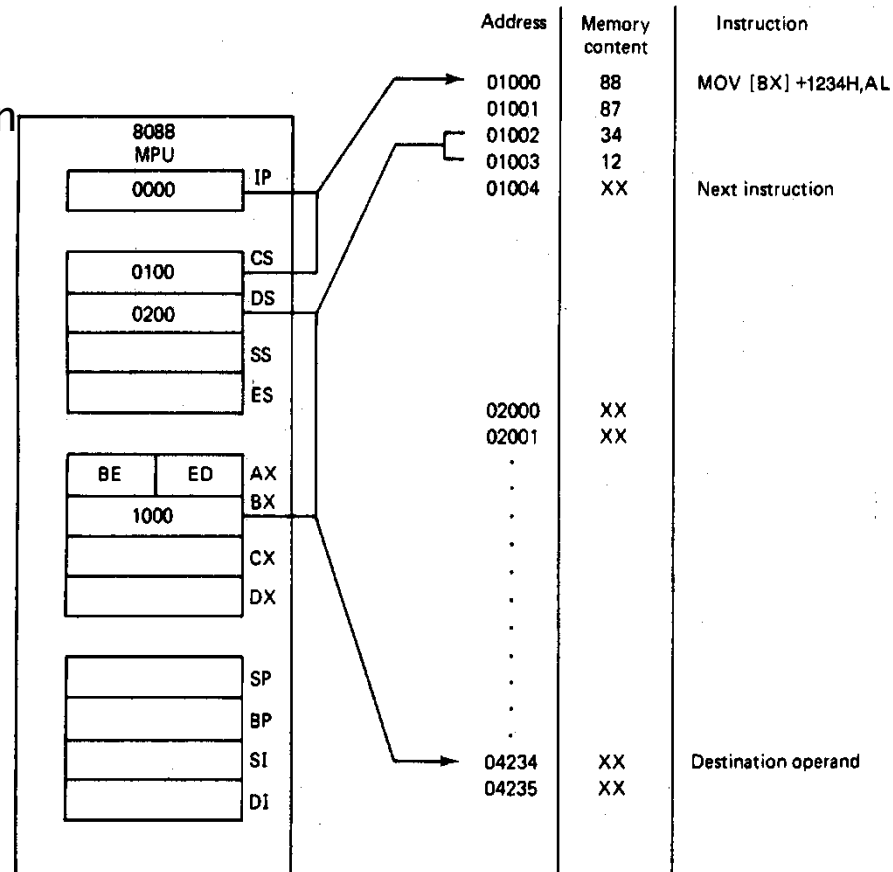
= 02000H+1000H+1234H

= 04234H

(04234H) = XXH

- Source operand—register operand addressing

(AL) = ED



Before execution (a)

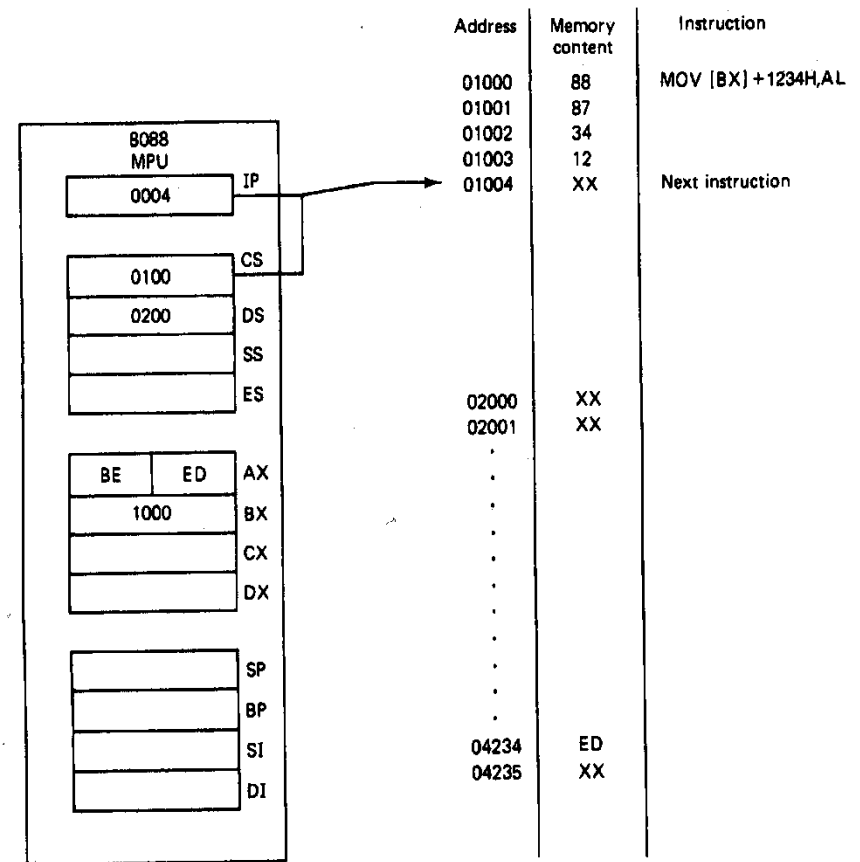
# 3.5 Addressing Modes

- ▶ Memory addressing modes – Based addressing mode

$$PA = 02000_{16} + 1000_{16} + 1234_{16} = 04234_{16}$$

- Example (continued)
- State after execution Instruction  
 $CS:IP = 0100:0004 = 01004H$   
 $01004H \rightarrow$  points to next sequential instruction
- Destination operand  
 $(04234H) = EDH$
- Source operand  
 $(AL) = EDH \rightarrow$  **unchanged**

**Note:** if BP is used instead of BX, the calculation of PA is performed using SS instead of DS.

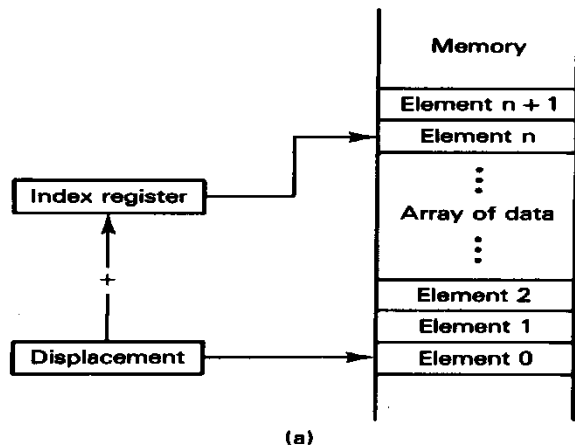


(b) After execution  
CPE 0408330

# 3.5 Addressing Modes

## Memory addressing modes – Indexed addressing mode

PA = Segment Base : Index + Displacement



PA = Segment base: Index + Displacement

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \begin{array}{c} SI \\ DI \end{array} \right\} + \left\{ \begin{array}{c} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{array} \right\}$$

(b)

Computation of an indexed address

e.g. `MOV AL, [SI]+1234H`

- Similar to based addressing, it makes accessing elements of data in an array easy
  - Displacement points to the **beginning** of array in memory
  - Index register selects element **within** the array
  - Program simply changes the value of the displacement to access another array
  - Program changes (re-computes) value in index register to select another element in the array
  - Effective address formed from direct displacement and contents of an index register
  - Direct displacement is 8-bit or 16-bit
  - Index register is either SI → source operand or DI → destination operand
  - Physical address computation
- PA = SBA:EA → 20-bit address  
 PA = SBA: DA + [SI or DI]

# 3.5 Addressing Modes

- Memory addressing modes – Indexed addressing mode

$$PA = 02000_{16} + 2000_{16} + 1234_{16} = 05234_{16}$$

## • Example

**MOV AL,[SI] +1234H**

- State before fetch and execution Instruction

CS = 0100H

IP = 0000H

CS:IP = 0100:0000H = 01000H

(01000H,01001H) = Opcode = 8A84H

(01002H,01003H) = Direct displacement = 1234H

- Source operand—indexed addressing

DS = 0200H

SI = 2000H

DA = 1234H

PA = DS:SI+DA = 0200H:2000H+1234H

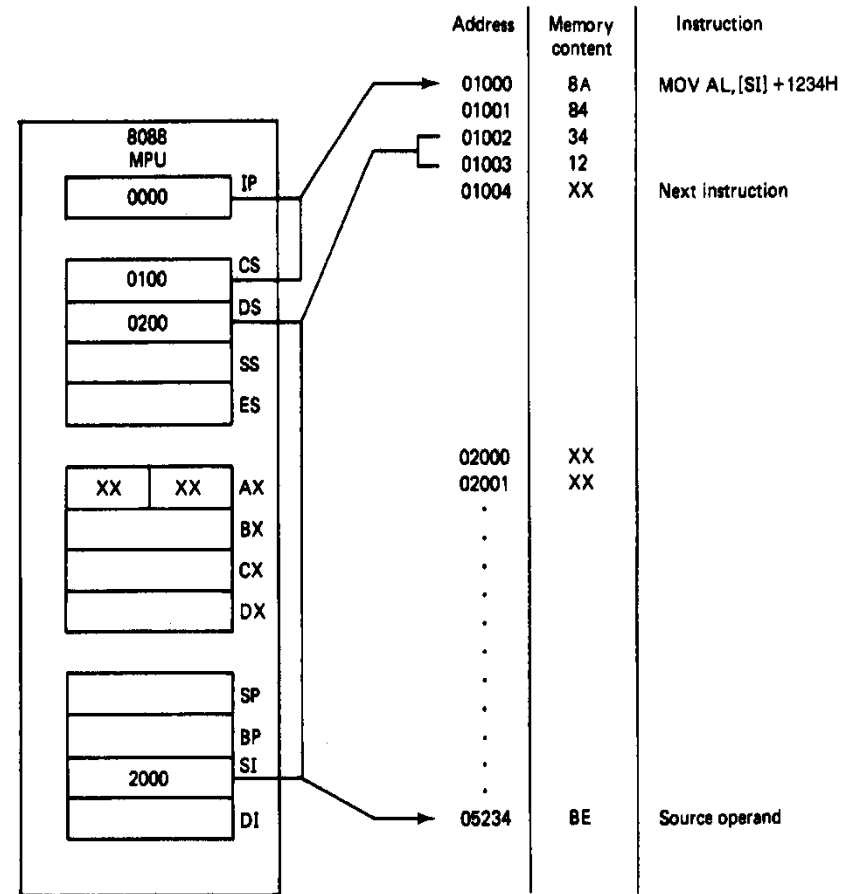
= 02000H+2000H+1234H

= 05234H

(05234H) = BEH

- Destination operand—register operand addressing

(AL) = XX → don't care state



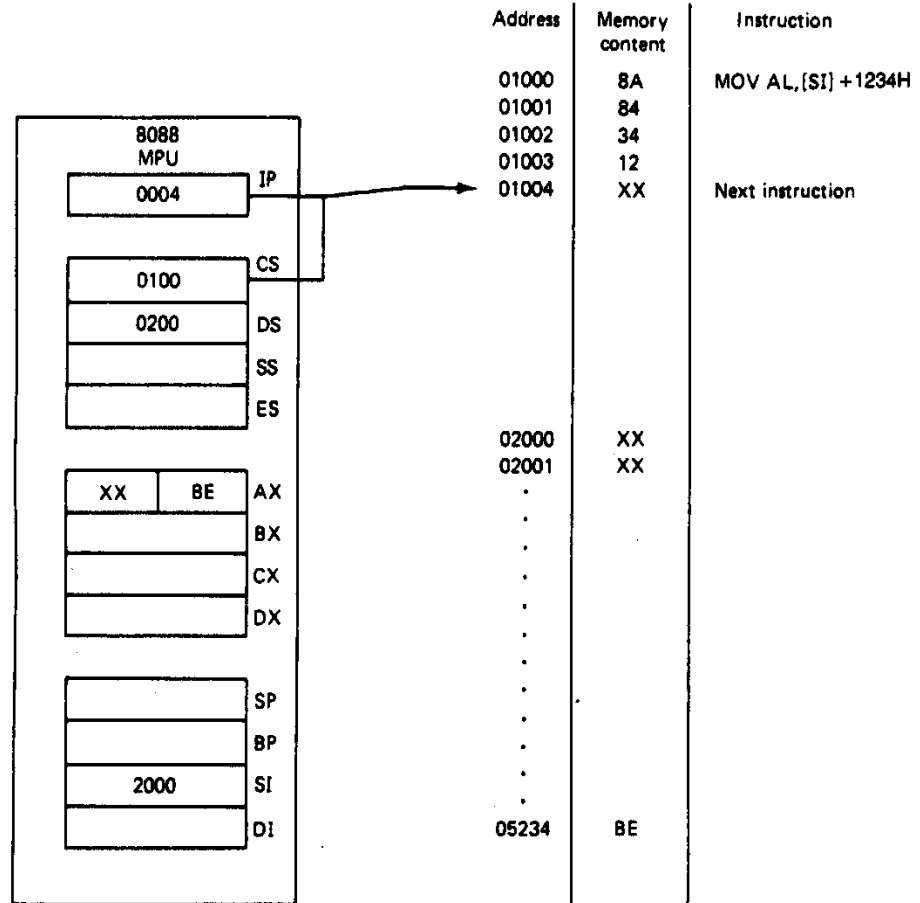
(a) Before execution

# 3.5 Addressing Modes

- Memory addressing modes – Indexed addressing mode

$$PA = 02000_{16} + 2000_{16} + 1234_{16} = 05234_{16}$$

- Example (continued)
- State after execution Instruction  
 $CS:IP = 0100:0004 = 01004H$   
 $01004H \rightarrow$  points to next sequential instruction
- Source operand  
 $(05234H) = BEH \rightarrow$  unchanged
- Destination operand  
 $(AL) = BEH$

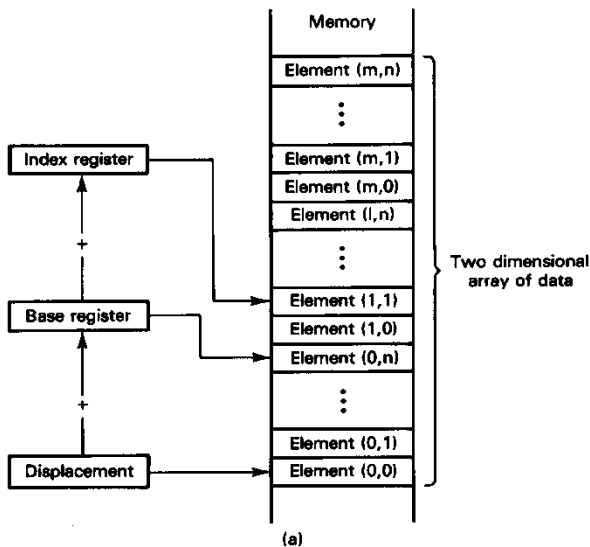


(b)

After execution

# 3.5 Addressing Modes

## ▶ Memory addressing modes – Based-indexed addressing mode



PA = Segment base: Base + Index + Displacement

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{Bmatrix}$$

### Computation of an indexed address

e.g. `MOV AH, [BX][SI]+1234H`

- Combines the functions of based and indexed addressing modes
  - Enables easy access to **two-dimensional arrays** of data
  - **Displacement** points to the **beginning** of array in memory
  - **Base** register selects the **row (m) of elements**
  - **Index** register selects element in a **column (n)**
  - Program simply changes the value of the displacement to access another array
  - Program changes (re-computes) value in base register to select another row of elements
  - Program changes (re-computes) the value of the index register to select the element in another column
  - Effective address formed from direct displacement and contents of a base register and an index register
  - Direct displacement is 8-bit or 16bit
  - Base register either BX or BP (stack)
  - Index register is either SI → source operand or DI → destination operand
  - Physical address computation
- PA = SBA:EA → 20-bit address  
 PA = SBA:DA + [BX or BP] + [SI or DI]

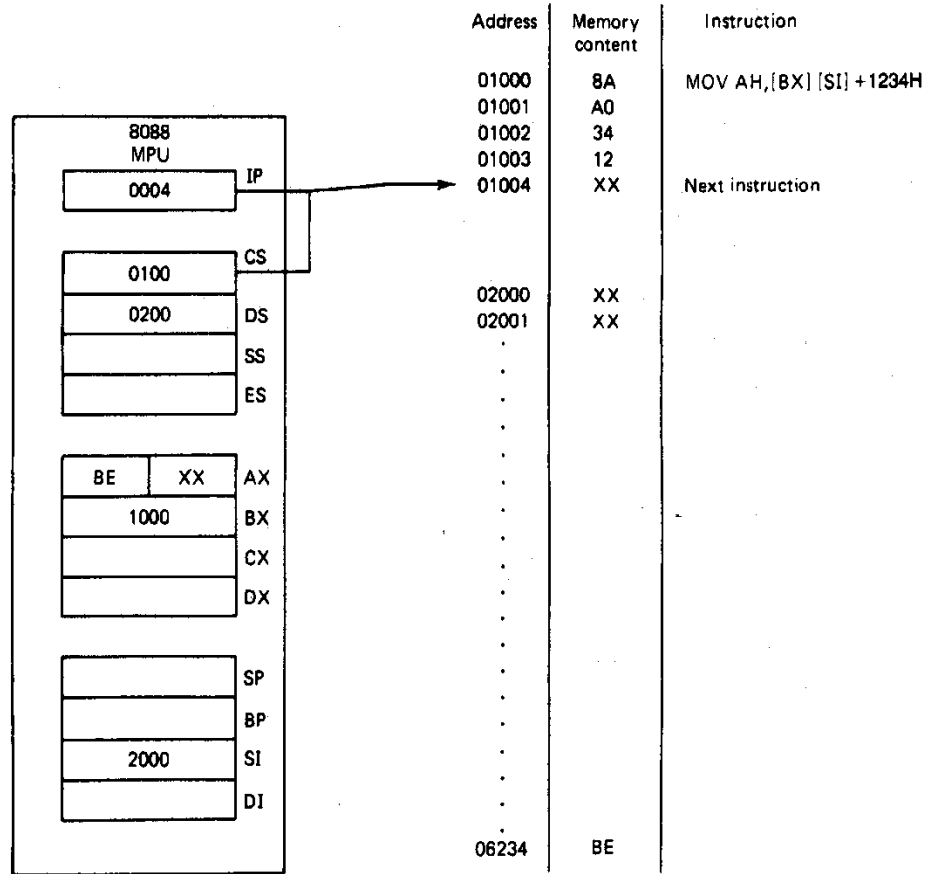




# 3.5 Addressing Modes

- ▶ Memory addressing modes – Based-indexed addressing mode  
 $PA = 02000_{16} + 1000_{16} + 2000_{16} + 1234_{16} = 06234_{16}$

- Example (continued)
- State after execution Instruction  
 $CS:IP = 0100:0004 = 01004H$   
 $01004H \rightarrow$  points to next sequential instruction
- Source operand  
 $(06234H) = BEH \rightarrow$  unchanged
- Destination operand  
 $(AH) = BEH$



(b)  
After execution

# H.W. #3

- Solve the following problems from Chapter 3 from the course textbook:  
5, 10, 23, 25, 26, 29, 31

# **CPE 408330**

## **Assembly Language and Microprocessors**

### **Chapter 4: Machine Language Coding and the DEBUG Software Development Program of the PC**

[Computer Engineering Department,  
Hashemite University]

# Lecture Outline

- ▶ 4.1 Converting Assembly Language Instructions to Machine Code
- ▶ 4.2 Encoding a Complete Program in Machine Code
- ▶ 4.3 The PC and Its DEBUG Program
- ▶ 4.4 Examining and Modifying the Contents of Memory
- ▶ 4.5 Input and Output of Data

# Lecture Outline

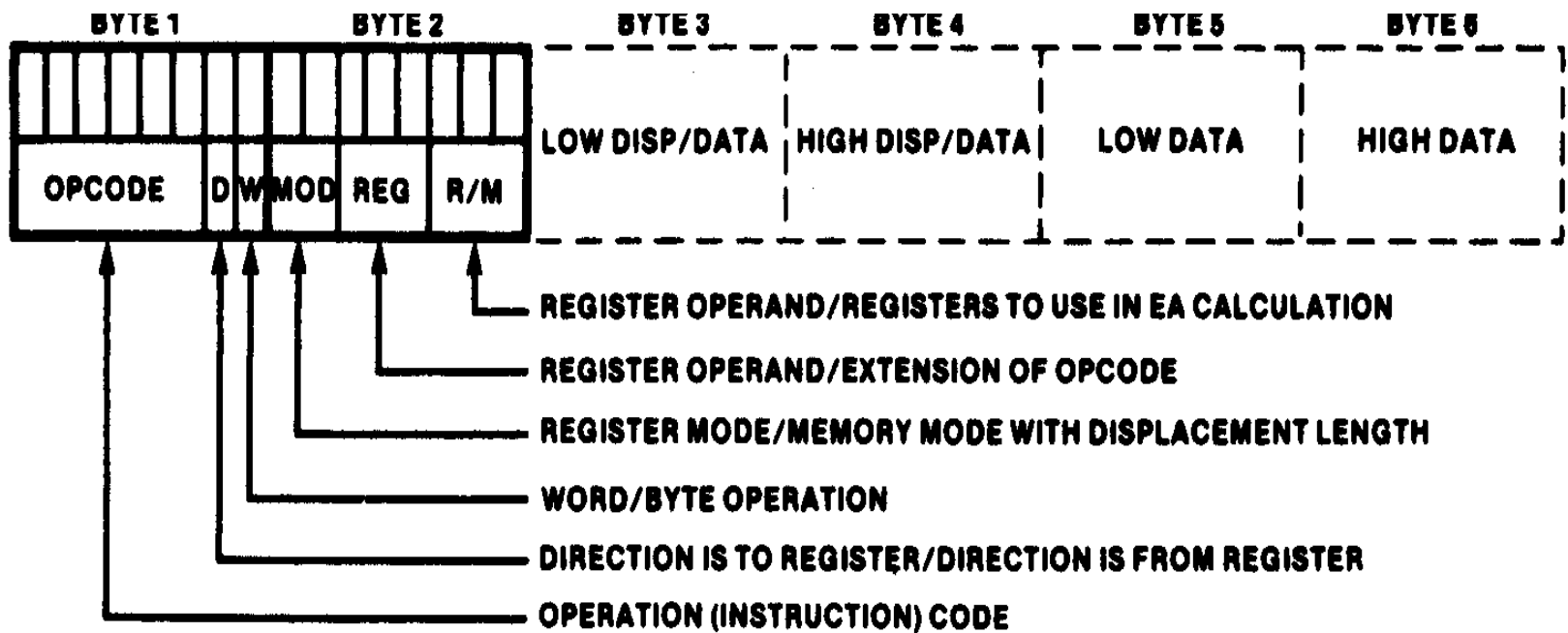
- ▶ 4.6 Hexadecimal Addition and Subtraction
- ▶ 4.7 Loading, Verifying and Saving Machine Language Program
- ▶ 4.8 Assembling Instructions with the Assemble Command
- ▶ 4.9 Executing Instructions and Programs with the TRACE and GO command
- ▶ 4.10 Debugging a Program

# 4.1 *Converting Assembly Language Instructions to Machine Code*

- ▶ Part of the 80x86 instruction set architecture (ISA)
  - What is the machine instruction length (fixed, variable, hybrid)?
  - What are the sizes of the fields—varying sizes?
  - What are the functions of the fields?
- ▶ 80x86's register-memory architectures is hybrid length
  - Multiple instruction sizes, but all have byte wide lengths—
    - 1 to 6 bytes in length for 8088/8086
    - Up to 17 bytes for 80386, 80486, and Pentium
  - Advantages of hybrid length
    - Allows for many addressing modes
    - Allows full size (32-bit) immediate data and addresses
  - Disadvantage of variable length
    - Requires more complicated decoding hardware—speed of decoding is critical in modern application
- ▶ Load-store architectures normally fixed length—PowerPC (32-bit), SPARC (32-bit), MIPS (32-bit), Itanium (128-bits, 3 instructions)

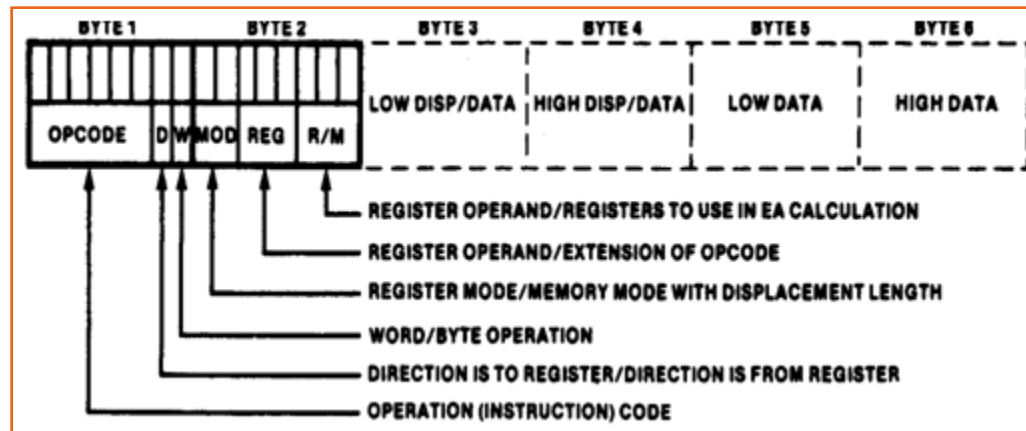
# 4.1 Converting Assembly Language Instructions to Machine Code

- ▶ **General instruction format** for machine code:



# 4.1 Converting Assembly Language Instructions to Machine Code

- ▶ **Byte 1** specification:
  - **Opcode** field (6–bits)
    - Specifies the operation to be performed
  - **Register direction** bit (D–bit)
    - 1 – the register operand is a destination operand
    - 0 – the register operand is a source operand
  - **Data size** bit (W–bit)
    - 1 – 16–bit data size
    - 0 – 8–bit data size





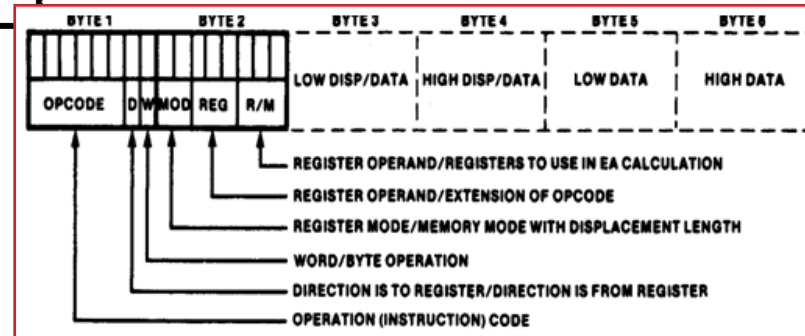
# 4.1 Converting Assembly Language Instructions to Machine Code

- ▶ **Byte 2** specification
  - Mode (MOD) field (2-bits)—specifies the type of the second operand

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

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

MOD field and R/M field encoding



# 4.1 *Converting Assembly Language Instructions to Machine Code*

- ▶ Byte 2 specification
  - Mode (**MOD**) field (2-bits):
    - Memory mode: 00, 01, 10—Register to memory move operation
      - **00** = no immediate displacement (register used for addressing)
      - **01** = 8-bit displacement (imm8) follows (8-bit offset address)
      - **10** = 16-bit displacement (imm16) follows (16-bit offset address)
    - Register mode: **11**—register to register move operation
      - **11** = register specified as the second operand

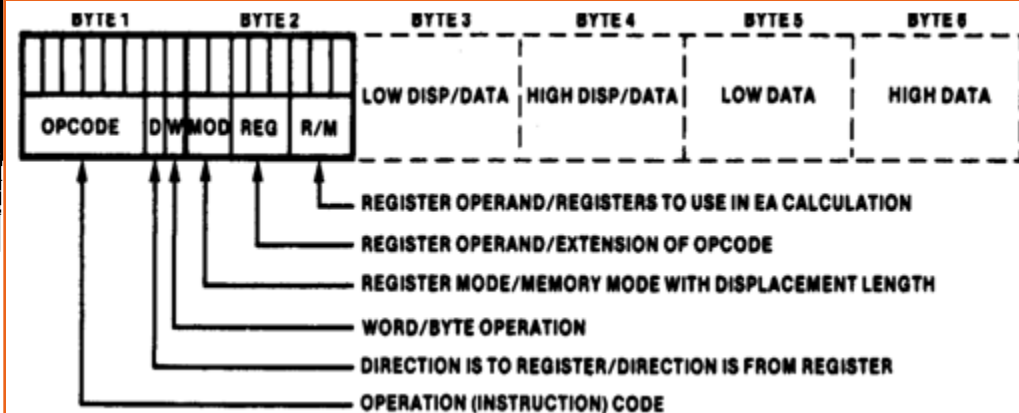
Register (REG) field encoding

# 4.1 Converting Assembly Language Instructions to Machine Code

- ▶ Byte 2 specification
  - Register (**REG**) field (3-bit)
    - Identifies the register for the first operand
    - W (1-bit)—data size word/byte for all registers
      - Byte = 0
      - Word = 1
  - Register/Memory (**R/M**) field (3-bit)

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

Register (REG) field encoding

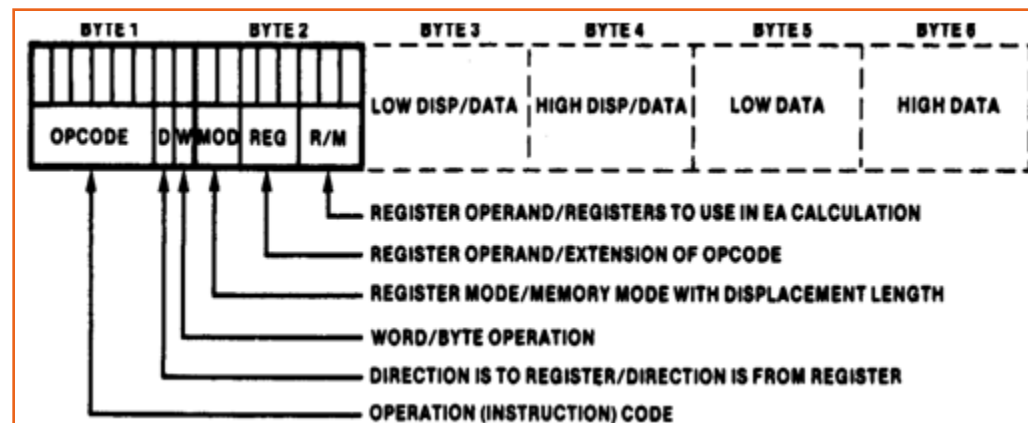


# 4.1 Converting Assembly Language Instructions to Machine Code

## ▶ Byte 2 specification

- Register/Memory (R/M) field (3-bit)—specifies the **second** operand as a register or a storage location in memory
  - Dependent on MOD field
    - **Mod = 11**, R/M selects a register:
      - R/M = 000 Accumulator register
      - R/M = 001 = Count register
      - R/M = 010 = Data Register

MOD = 11		
R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI



# 4.1 Converting Assembly Language Instructions to Machine Code

## ▶ Byte 2 specification

- MOD = 00, 10, or 11 selects an addressing mode for the second operand that is a storage location in **memory**, which may be the source or destination
  - Dependent on MOD field
  - Mod = 00, R/M:
    - R/M = 100 → effective address computed as  $EA = (SI)$
    - R/M = 000 = → effective address computed as  $EA = (BX) + (SI)$
    - R/M = 110 = → effective address is coded in the instruction as a direct address  $EA = \text{direct address}$   
= imm8 or imm16

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

# 4.1 *Converting Assembly Language Instructions to Machine Code*

## ▶ EXAMPLE

Encode the instruction in machine code  
`MOV BL, AL`

- ▶ Reg–Reg instruction has two encodings:
- ▶ Encoding (1) REG is source
  - ▶ D-bit is 0: this means REG specifies source
  - ▶ R/W specifies destination register
- ▶ Encoding (2) REG is destination
  - ▶ D-bit is 1: this means REG specifies destination
  - ▶ R/W specifies source

## 4.1 *Encoding (1):* MOV BL, AL

- ▶ EXAMPLE

Encode the instruction in machine code  
MOV BL, AL

- ▶ Solution:

Byte 1:            OPCODE = 100010 (for MOV),  
                    **D = 0 (source)**, W = 0 (8-bit)

- ▶ This leads to BYTE 1 =  $10001000_2 = 88_{16}$

- ▶ In byte 2 the source operand, specified by REG, is AL

**REG = 000**, MOD = 11, **R/M = 011**

- ▶ Therefore, BYTE 2 =  $11000011_2 = C3_{16}$

MOV BL, AL =  $88C3_{16}$

## 4.1 *Encoding (2):* MOV BL, AL

- ▶ EXAMPLE

Encode the instruction in machine code  
MOV BL, AL

- ▶ Solution:

Byte 1:            OPCODE = 100010 (for MOV),  
                    **D = 1 (destination)**, W = 0 (8-bit)

- ▶ This leads to BYTE 1 =  $10001010_2 = 8A_{16}$

- ▶ In byte 2 the source operand, specified by REG, is AL  
                    **REG = 011**, MOD = 11, **R/M = 000**

- ▶ Therefore, BYTE 2 =  $1101\ 1000_2 = C3_{16}$   
                    MOV BL, AL =  $8AD8_{16}$



# MOV BL, AL

MOD = 11		
R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Mnemonic and Description	Instruction Code			
<b>DATA TRANSFER</b>				
<b>MOV – Move:</b>	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		

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

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

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

# 4.1 Converting Assembly Language Instructions to Machine Code

## ▶ EXAMPLE

Encode the instruction in machine code

`ADD AX, [SI]`

## ▶ Solution:

OPCODE = 000000 (for ADD), D = 1 (dest.),  
W = 1 (16-bit)

▶ This leads to BYTE 1 =  $00000011_2 = 03_{16}$

▶ In byte 2 the destination operand, specified by REG, is AX

REG = 000, MOD = 00, R/M = 100

▶ Therefore, BYTE 2 =  $00000100_2 = 04_{16}$

`ADD AX, [SI]` =  $0304_{16}$

# ADD AX, [SI]

MOD = 11		
R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Mnemonic and Description	Instruction Code			
ARITHMETIC	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ADD – Add:				
Reg./Memory with Register to Either	000000dw	mod reg r/m		
Immediate to Register/Memory	100000sw	mod000r/m	data	data if s:w = 01
Immediate to Accumulator	0000010w	data	data if w = 1	
ADC – Add with Carry:				
Reg./Memory with Register to Either	000100dw	mod reg r/m		
Immediate to Register/Memory	100000sw	mod010r/m	data	data if s:w = 01
Immediate to Accumulator	0001010w	data	data if w = 1	

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

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

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX) <sub>CPE 0408330</sub>	(BX) + D8	(BX) + D16

# 4.1 Converting Assembly Language Instructions to Machine Code

- ▶ EXAMPLE

Encode the instruction in machine code

`XOR CL, [1234H]`

- ▶ Solution:

OPCODE = 001100 (for XOR), D = 1 (dest.), W = 0 (8-bit)

- ▶ This leads to BYTE 1 = 00110010<sub>2</sub> = 32<sub>16</sub>

- ▶ In byte 2 the destination operand, specified by REG, is CL

REG = 001, MOD = 00, R/M = 110

- ▶ Therefore, BYTE 2 = 00001110<sub>2</sub> = 0E<sub>16</sub>

BYTE 3 = 34<sub>16</sub>

BYTE 4 = 12<sub>16</sub>

`XOR CL, [1234H]` = 320E3412<sub>16</sub>

# XOR CL, [1234H]

MOD = 11		
R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

OR – Or:

Reg./Memory and Register to Either

Immediate to Register/Memory

Immediate to Accumulator

XOR – Exclusive or:

Reg./Memory and Register to Either

Immediate to Register/Memory

Immediate to Accumulator

000010dw	mod reg r/m		
1000000w	mod 001r/m	data	data if w = 1
0000110w		data	data if w = 1
001100dw	mod reg r/m	Disp. H	Disp. L
1000000w	mod 110r/m	data	data if w = 1
0011010w		data	data if w = 1

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

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

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

# 4.1 Converting Assembly Language Instructions to Machine Code

- ▶ EXAMPLE

Encode the instruction in machine code

`ADD [BX][DI]+1234H, AX`

- ▶ Solution:

    OPCODE = 000000 (for ADD), D = 0 (source), W = 1 (16-bit)

- ▶ This leads to BYTE 1 = 00000001<sub>2</sub> = 01<sub>16</sub>

- ▶ In byte 2 the destination operand, specified by REG, is AX

    REG = 000, MOD = 10, R/M = 001

- ▶ Therefore, BYTE 2 = 10000001<sub>2</sub> = 81<sub>16</sub>

    BYTE 3 = 34<sub>16</sub>

    BYTE 4 = 12<sub>16</sub>

`ADD [BX][DI]+1234H, AX = 0181341216`

# ADD [BX][DI]+1234H, AX

MOD = 11		
R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Mnemonic and Description	Instruction Code			
<b>ARITHMETIC</b>	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
<b>ADD – Add:</b>				
Reg./Memory with Register to Either	000000d w	mod reg r/m	Disp. H	Disp. L
Immediate to Register/Memory	100000s w	mod 000r/m	data	data if s: w = 01
Immediate to Accumulator	0000010 w	data	data if w = 1	
<b>ADC – Add with Carry:</b>				
Reg./Memory with Register to Either	000100d w	mod reg r/m	Disp. H	Disp. L
Immediate to Register/Memory	100000s w	mod 010r/m	data	data if s: w = 01
Immediate to Accumulator	0001010 w	data	data if w = 1	

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

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

MOD=11			EFFECTIVE ADDRESS CALCULATION			
R/M	W=0	W=1	R/M	MOD=00	MOD=01	MOD=10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX) <sup>CPE 0408330</sup>	(BX) + D8	(BX) + D16

# 4.1 Converting Assembly Language Instructions to Machine Code

- ▶ Additional one-bit field and their functions

Field	Value	Function
S	0	No sign extension
	1	Sign extend 8-bit immediate data to 16 bits if W=1
V	0	Shift/rotate count is one
	1	Shift/rotate count is specified in CL register
Z	0	Repeat/loop while zero flag is clear
	1	Repeat/loop while zero flag is set

Immediate to Register/Memory

1 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s: w = 01
---------------	---------------	------	-------------------

ROL – Rotate Left

1 1 0 1 0 0 v w	mod 0 0 0 r/m
-----------------	---------------

- ▶ Instructions that involve a segment register (SR-field)

Register	SR
ES	00
CS	01
SS	10
DS	11



# 4.1 Converting Assembly Language Instructions to Machine Code

## ▶ EXAMPLE

Encode the instruction in machine code

`MOV WORD PTR [BP][DI]+1 234H, 0ABCDH`

## ▶ Solution:

This example does not follow the general format

▶ From Fig. 3-6 in the text, MOV  $\rightarrow$  1100011W, and W = 1 for word-size data

▶ BYTE 1 = 11000111<sub>2</sub> = C7<sub>16</sub>

▶ BYTE 2 = (MOD)000(R/M) = 10000011<sub>2</sub> = 83<sub>16</sub>

▶ BYTE 3 = 34<sub>16</sub> BYTE 4 = 12<sub>16</sub>

▶ BYTE 5 = CD<sub>16</sub> BYTE 6 = AB<sub>16</sub>

`MOV WORD PTR [BP][DI]+1 234H, 0ABCDH`  
`= C7833412CDAB16`

# MOV WORD PTR [BP][DI]+1234H, 0ABCDH

Mnemonic and Description	Instruction Code					
<b>DATA TRANSFER</b>						
<b>MOV – Move:</b>	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m	Disp. H	Disp. L		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	Disp. H	Disp. L	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1			
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high			
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high			

MOD = 11		
R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

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

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

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

# 4.1 Converting Assembly Language Instructions to Machine Code

## ▶ EXAMPLE

Encode the instruction in machine code

`MOV [BP][DI]+1234H, DS`

## ▶ Solution:

This example does not follow the general format

▶ From Fig. 3-6 in the text, MOV  $\rightarrow$  10001100, and the instruction is 10001100(MOD)0(SR)(R/M)(DISP)

▶ From Fig. 4-5 in the text, we find that for DS, the SR = 11

▶ Therefore, the instruction is coded as

`MOV [BP][DI]+1234H, DS`  
= 100011001001101100110100000100102  
= 8C9B3412<sub>16</sub>

# MOV [BP][DI]+1 234H, DS

MOD = 11		
R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register	SR
ES	00
CS	01
SS	10
DS	11

Mnemonic and Description	Instruction Code			
<b>DATA TRANSFER</b>				
<b>MOV – Move:</b>	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 SR r/m	Disp LO	Disp HI

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

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

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

## *4.2 Encoding a Complete Program in Machine Code*

- ▶ Steps in encoding a complete assembly program:
  - Identify the general machine code format (Fig. 3–6)
  - Evaluate the bit fields (Fig. 4–2,4–3,4–4,4–5)
  - Express the binary–code instruction in hexadecimal form
- ▶ To execute the program, the machine code of the program must be stored in the **code segment** of memory.
- ▶ The **first byte** of the program is stored at the **lowest address**.

# 4.2 Encoding a Complete Program in Machine Code

## ▶ EXAMPLE

Encode the “block move” program in Fig. 4–6(a) and show how it would be stored in memory starting at address 20016.

## ▶ Solution:

MOV AX, 2000H	;LOAD AX REGISTER
MOV DS, AX	;LOAD DATA SEGMENT ADDRESS
MOV SI, 100H	;LOAD SOURCE BLOCK POINTER
MOV DI, 120H	;LOAD DESTINATION BLOCK POINTER
MOV CX, 10H	;LOAD REPEAT COUNTER
NXTPT: MOV AH, [SI]	;MOVE SOURCE BLOCK ELEMENT TO AH
MOV [DI], AH	;MOVE ELEMENT FROM AH TO DEST. BLOCK
INC SI	;INCREMENT SOURCE BLOCK POINTER
INC DI	;INCREMENT DESTINATION BLOCK POINTER
DEC CX	;DECREMENT REPEAT COUNTER
JNZ NXTPT	;JUMP TO NXTPT IF CX NOT EQUAL TO ZERO
NOP	;NO OPERATION

JNE/JNZ – Jump on Not Equal/Not Zero	0 1 1 1 0 1 0 1	IP-INC8
JNL/JGE – Jump on Not Less/Greater or Equal	0 1 1 1 1 1 0 1	disp
JNLE/JG – Jump on Not Less or Equal/Greater	0 1 1 1 1 1 1 1	disp

# 4.2 Encoding a Complete Program in Machine Code

Instruction	Type of instruction	Machine code
MOV AX,2000H	Move immediate data to register	$101110000000000000000000_2 = B80020_{16}$
MOV DS,AX	Move register to segment register	$1000111011011000_2 = 8ED8_{16}$
MOV SI,100H	Move immediate data to register	$10111110000000000000000001_2 = BE0001_{16}$
MOV DI,120H	Move immediate data to register	$10111111001000000000000001_2 = BF2001_{16}$
MOV CX,10H	Move immediate data to register	$10111001000100000000000000_2 = B91000_{16}$
MOV AH,[SI]	Move memory data to register	$1000101000100100_2 = 8A24_{16}$
MOV [DI],AH	Move register data to memory	$1000100000100101_2 = 8825_{16}$
INC SI	Increment register	$01000110_2 = 46_{16}$
INC DI	Increment register	$01000111_2 = 47_{16}$
DEC CX	Decrement register	$01001001_2 = 49_{16}$
JNZ NXTPT	Jump on not equal to zero	$0111010111110111_2 = 75F7_{16}$
NOP	No operation	$1001000_2 = 90_{16}$

# 4.2 Encoding a Complete Program in Machine Code

Memory address	Contents	Instruction
200H	B8H	MOV AX,2000H
201H	00H	
202H	20H	
203H	8EH	MOV DS,AX
204H	D8H	
205H	BEH	MOV SI,100H
206H	00H	
207H	01H	
208H	BFH	MOV DI,120H
209H	20H	
20AH	01H	
20BH	B9H	MOV CX,10H
20CH	10H	
20DH	00H	
20EH	8AH	MOV AH,[SI]
20FH	24H	
210H	88H	MOV [DI],AH
211H	25H	
212H	46H	INC SI
213H	47H	INC DI
214H	49H	DEC CX
215H	75H	JNZ \$-9
216H	F7H	
217H	90H	NOP



## 4.3 The PC and Its DEBUG Program

- ▶ Using DEBUG, the programmer can issue commands to the microcomputer.
- ▶ Loading the **DEBUG** program  
C:\DEBUG
- ▶ Six kinds of information are entered as part of a **command**:
  - A command letter
  - An address
  - A register name
  - A file name
  - A drive name
  - Data

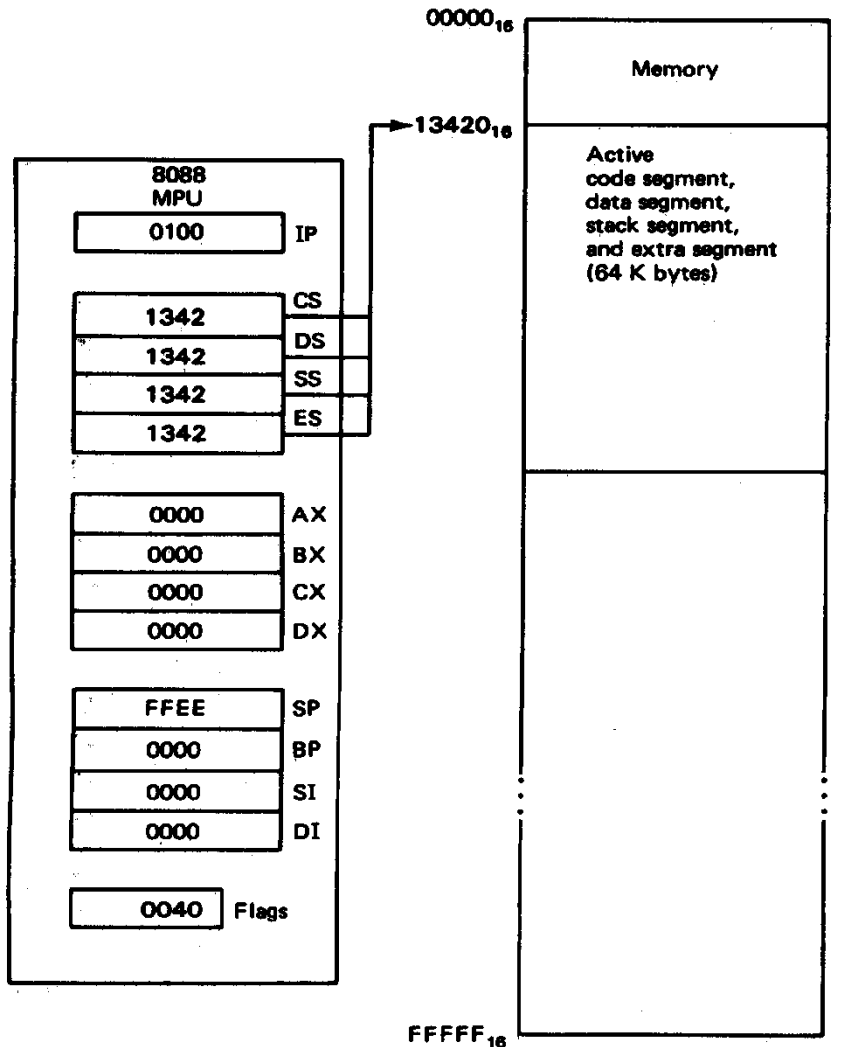
```
-R AX  
-D DS:100  
-N A:BLK.1  
-F 100 11F 22
```

## 4.3 The PC and Its DEBUG Program

- ▶ The DEBUG program **command set**:
  - Register: R [register name]
  - Quit: Q
  - Dump: D[address]
  - Enter: E address [list]
  - Fill: F st. address end address list
  - Move: M st. addr. end addr. dest. Addr.
  - Compare: C st. addr. end addr. dest. Addr.
  - Search: S st. address end address list
  - Input: I address
  - Output: O address, byte
  - Hex Add/Subtract: H num1,num2
  - Assemble: A [starting address]
  - Unassemble: U [starting address ending address]
  - **Name: N file name]**
  - **Write: W [st. addr. [drive st. sector no. of sectors]]**
  - **Load: L [st. addr. [drive st. sector no. of sectors]]**
  - Trace: T [=address] [number]
  - Go: G [= starting address [breakpoint address ...]]

# 4.3 The PC and Its DEBUG Program

- ▶ An **initial state** when with the loading of DEBUG



Status register (Flags) =  
0000 0000 0**1**00 0000  
Interrupt flag = 1 →  
**interrupt enable**

## 4.3 The PC and Its DEBUG Program

- ▶ Syntax for the REGISTER (R) command

R [REGISTER NAME]

- ▶ e.g.

```
-R AX (↵)
```

```
AX 0000  
:-
```

```
:00FF (↵)
```

;This alter the content of AX

```
-
```

If [REGISTER NAME] is empty → display the values of all registers

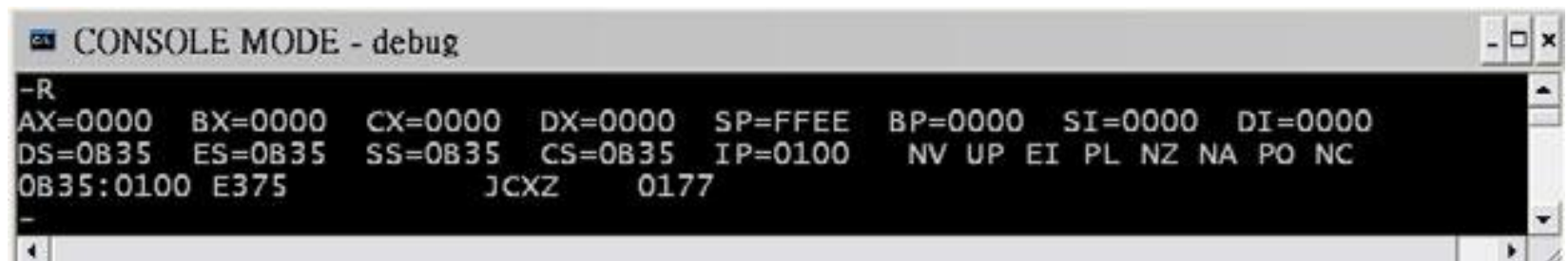
## 4.3 The PC and Its DEBUG Program

- ▶ EXAMPLE

Verify the initialized state of the 8088 by examining the contents of its registers with the Register command.

- ▶ Solution:

-R (↵)



```
CONSOLE MODE - debug
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0100  NV UP EI PL NZ NA PO NC
0B35:0100 E375          JCXZ    0177
-
```

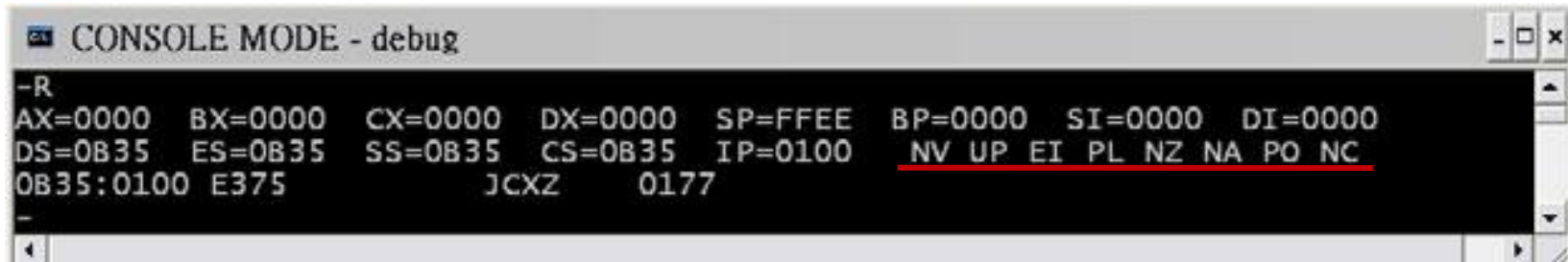
## 4.3 The PC and Its DEBUG Program

- ▶ Register mnemonics for the R command

Symbol	Register
AX	Accumulator register
BX	Base register
CX	Count register
DX	Data register
SI	Source index register
DI	Destination index register
SP	Stack pointer register
BP	Base pointer register
CS	Code segment register
DS	Data segment register
SS	Stack segment register
ES	Extra segment register
F	Flag register
IP	Instruction pointer

# 4.3 The PC and Its DEBUG Program

- ▶ Status flag notations



```
CONSOLE MODE - debug
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0100 NV UP EI PL NZ NA PO NC
0B35:0100 E375 JCXZ 0177
-
```

Flag	Meaning	Set	Reset
OF	Overflow	OV	NV
DF	Direction	DN	UP
IF	Interrupt	EI	DI
SF	Sign	NG	PL
ZF	Zero	ZR	NZ
AF	Auxiliary carry	AC	NA
PF	Parity	PE	PO
CF	Carry	CY	NC

## 4.3 The PC and Its DEBUG Program

- ▶ EXAMPLE

Issue commands to the DEBUG program on the PC that causes the value in BX to be modified to FF00<sub>16</sub> and then verify that this new value is loaded into BX.

- ▶ Solution:

```
-R BX (↵)
BX 0000
:FF00 (↵)
-R BX (↵)
BX FF00
:_ (↵)
-
```



## 4.3 The PC and Its DEBUG Program

- ▶ EXAMPLE

Use the Register command to set the parity flag to even parity. Verify that the flag has been changed.

- ▶ Solution:

```
-R F (↵)
NV UP EI PL NZ NA PO NC -PE (↵)
-R F (↵)
NV UP EI PL NZ NA PE NC - (↵)
```

## *4.4 Examining and Modifying the Contents of Memory*

- ▶ The commands provided for use in examining and modifying the **memory**:
  - DUMP
  - ENTER
  - FILL
  - MOVE
  - COMPARE
  - SEARCH

# 4.4 Examining and Modifying the Contents of Memory

- ▶ DUMP Command (D)

The DUMP command allows us to **examine** the contents of a memory location or a **block of consecutive** memory location.

D [ADDRESS]

- ▶ e.g.

```
-D (↵)
-D 1342:100 (↵)
-D DS:100 (↵)
-D 100 (↵)
```

Command	Meaning
- D	Display 128 bytes starting from DS:0100
- D 1373:200	Display 128 bytes starting from 1373:200
- D 1F0	Display 128 bytes starting from DS:1F0
- D 200 300	Display memory locations from DS:200 to DS:300
- D CS:200 212	Display memory locations from CS:200 to CS:212

# 4.4 Examining and Modifying the Contents of Memory

- ▶ DUMP Command (D)

```
CONSOLE MODE - debug
-D
0B35:0100 E3 75 18 50 A0 D3 96 04-41 F8 2F 01 80 3A F8 29 .u P . . . A . . . : )
0B35:0110 01 58 89 3E F3 99 C6 06-F5 99 00 E8 34 00 24 0B .X.>.....4.$
0B35:0120 E8 17 01 AC EB 78 80 3E-B1 98 01 75 03 E8 8D E0 .....x.>...u...
0B35:0130 3C 2E 75 09 FE 06 F6 99-C6 06 F5 99 FF 3C 3F 75 <.u.....<?u
0B35:0140 03 80 CF 02 3C 2A 75 30-80 CF 02 80 3E 2F 9A 00 ....<#u0.....>/..
0B35:0150 75 04 EB 24 EB 78 B4 07-80 3E F6 99 00 74 02 B4 u..$.x.>...t..
0B35:0160 02 80 3F 2A 26 F5 99 72-EB 86 E1 E3 09 86 E1 E8 ..?*&...r.....
0B35:0170 C8 00 86 E1 E2 F7 86 E1-EB D4 E2 75 21 80 CF 04 .....u!...
```

Address of the first byte of data

ASCII version of the memory data

16 bytes of data per line,  
128 bytes per dump

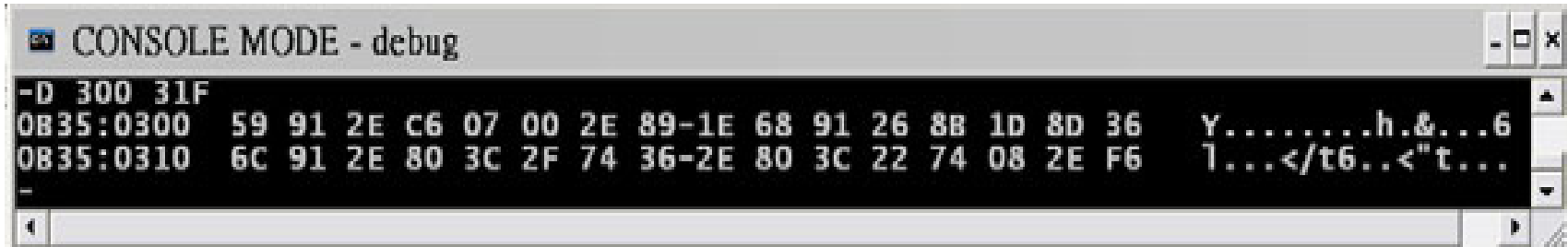
# 4.4 Examining and Modifying the Contents of Memory

- ▶ EXAMPLE

Issue a dump command to display the contents of the 32 bytes of memory located at offset 0300<sub>16</sub> through 031F<sub>16</sub> in the current data segment.

- ▶ Solution:

```
-D 300 31F (↵)
```



# 4.4 Examining and Modifying the Contents of Memory

- ▶ EXAMPLE

Use the Dump command to examine the 16 bytes of memory just below the top of the stack.

- ▶ Solution:

-D SS:FFEE FFFD (↵)



```
CONSOLE MODE - debug
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0100 NV UP EI PL NZ NA PO NC
0B35:0100 E375 JCXZ 0177
-
-D SS:FFEE FFFD
0B35:FFE0 00 00
0B35:FFF0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 ..
```

# 4.4 Examining and Modifying the Contents of Memory

- ▶ **ENTER** Command (E)

E ADDRESS [LIST]

e.g.

```
-E DS:100 FF FF FF FF FF (↵)
```

Used to browse group of locations and enter a values for specific one

```
-E DS:100 (↵)  
-1342:0100 FF. _ (↵) (Return to end)
```

```
-E DS:100 (↵)  
-1342:0100 FF. _ (Space bar to continue)  
-1342:0100 FF. FF._
```

# 4.4 Examining and Modifying the Contents of Memory

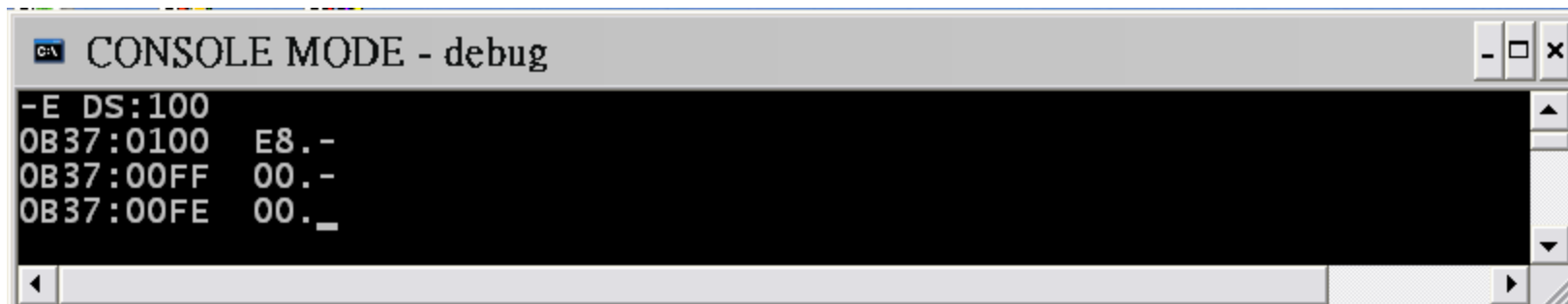
## ▶ EXAMPLE

Start a data entry sequence by examining the contents of address DS:100 and then, without entering new data, depress the “-” key. What happens?

## ▶ Solution:

```
-E DS:100 (↵)
1342:0100 FF. _
```

Entering “-” causes the display of **previous** byte storage location.



```
CONSOLE MODE - debug
-E DS:100
0B37:0100  E8.-
0B37:00FF  00.-
0B37:00FE  00._
```



# 4.4 Examining and Modifying the Contents of Memory

- ▶ EXAMPLE

Enter ASCII data to the memory.

- ▶ Solution:

```
-E DS:200 "ASCII" (↵)  
or  
-E DS:200 'ASCII' (↵)
```



The screenshot shows a debugger console window titled "CONSOLE MODE - debug". The window contains the following text:

```
-E DS:200 "ASCII"  
-D DS:200 204  
0B37:0200 41 53 43 49 49  
-  
_
```

The word "ASCII" is displayed on the right side of the console window.

## 4.4 Examining and Modifying the Contents of Memory

- ▶ **FILL** Command (F)

The FILL command fills a block of consecutive memory locations all with the same data.

**F STARTING\_ADDRESS ENDING\_ADDRESS LIST**

- ▶ e.g.

```
-F 100 11F 22 (←↓)
```

Issue two fill commands to fill memory locations with different values

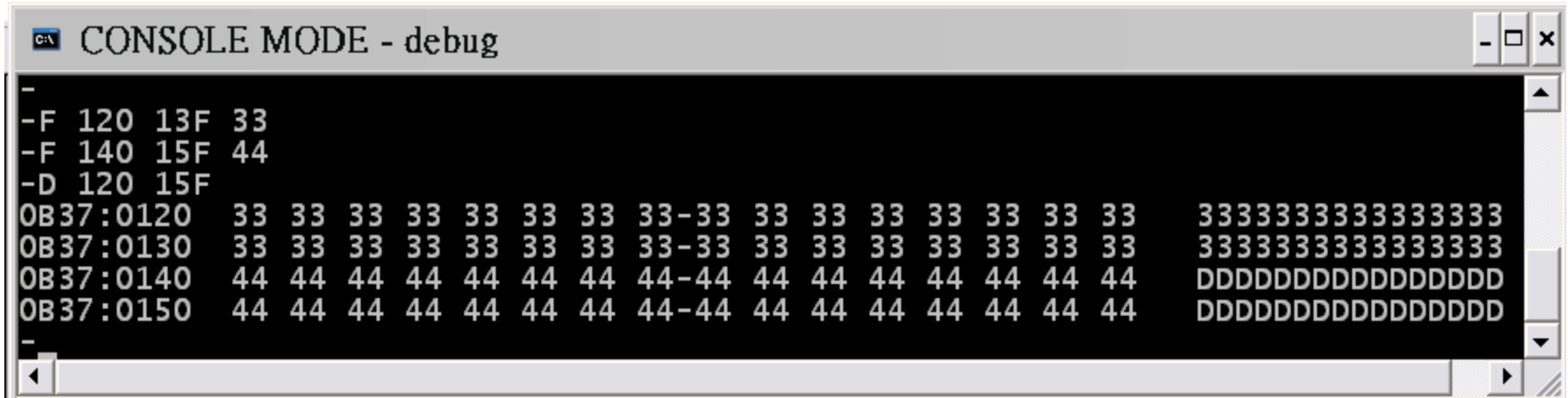
# 4.4 Examining and Modifying the Contents of Memory

- ▶ EXAMPLE

Initialize all storage locations in the block of memory from DS:120 through DS:13F with the value 33<sub>16</sub> and the block of storage locations from DS:140 to DS:15F with the value 44<sub>16</sub>.

- ▶ Solution:

```
-F 120 13F 33 (↵)
-F 140 15F 44 (↵)
```



# 4.4 Examining and Modifying the Contents of Memory

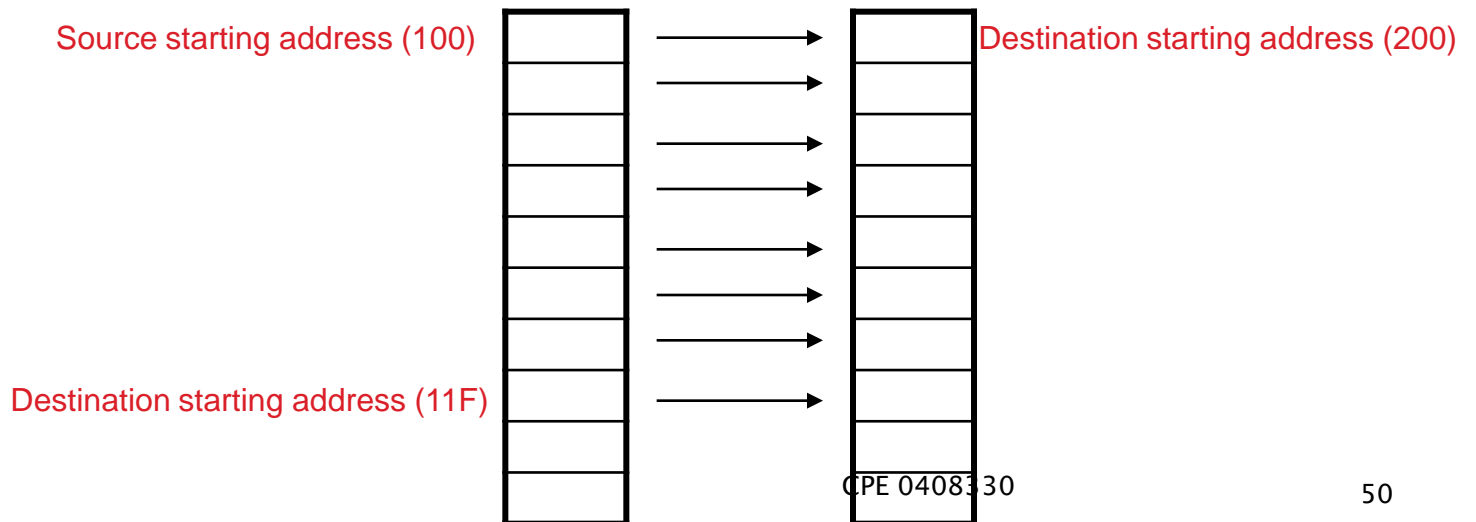
- ▶ **MOVE** Command (M)

The MOVE command allows us to copy a block of data from one part of memory to another part. Note that the source locations is not affected

M START\_ADDRESS END\_ADDRESS DEST\_ADDRESS

- ▶ e.g.

```
-M 100 11F 200 (←)
```



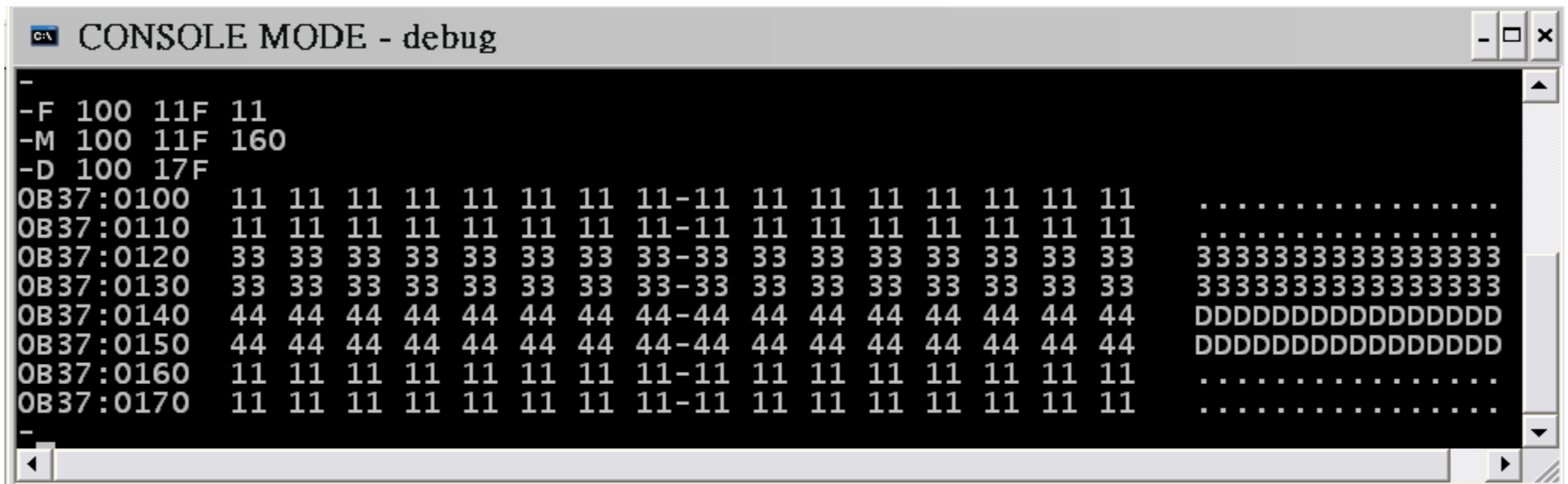
# 4.4 Examining and Modifying the Contents of Memory

- ▶ EXAMPLE

Fill each storage location in the block of memory from address DS:100 through DS:11F with the value 11 16. Then copy this block of data to a destination block starting at DS:160.

- ▶ Solution:

```
-F 100 11F 11 (↵)
-M 100 11F 160 (↵)
```



## 4.4 Examining and Modifying the Contents of Memory

- ▶ **COMPARE** Command (C)

The COMPARE command allows us to compare the contents of two blocks of data to determine if they are the same or not.

`C START_ADDRESS END_ADDRESS DEST_ADDRESS`

- ▶ e.g.

```
-C 100 10F 120 (↵)
```

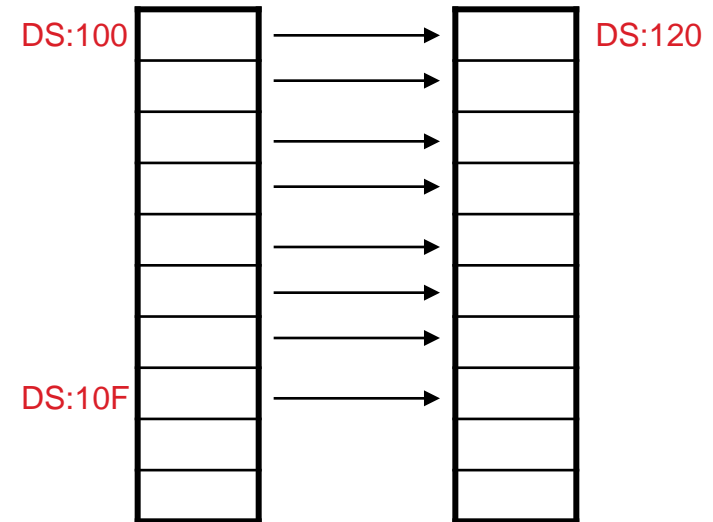
If the two locations are equal → don't display anything.

If the two locations are different → display each location with its content

# 4.4 Examining and Modifying the Contents of Memory

- ▶ COMPARE Command (C)

```
CONSOLE MODE - debug
-
-C 100 10F 120
0B37:0100 11 33 0B37:0120
0B37:0101 11 33 0B37:0121
0B37:0102 11 33 0B37:0122
0B37:0103 11 33 0B37:0123
0B37:0104 11 33 0B37:0124
0B37:0105 11 33 0B37:0125
0B37:0106 11 33 0B37:0126
0B37:0107 11 33 0B37:0127
0B37:0108 11 33 0B37:0128
0B37:0109 11 33 0B37:0129
0B37:010A 11 33 0B37:012A
0B37:010B 11 33 0B37:012B
0B37:010C 11 33 0B37:012C
0B37:010D 11 33 0B37:012D
0B37:010E 11 33 0B37:012E
0B37:010F 11 33 0B37:012F
```



Results produced when unequal data are found with a COMPARE command

## 4.4 Examining and Modifying the Contents of Memory

- ▶ **SEARCH** Command (S)

The SEARCH command can be used to scan through a block of data in memory to determine whether or not it contains specific data.

**S START\_ADDRESS END\_ADDRESS LIST**

- ▶ e.g.

```
-S 100 17F 33 (↵)
```



# 4.4 Examining and Modifying the Contents of Memory

- ▶ SEARCH Command (S)

```
C:\> CONSOLE MODE - debug
-
-D 100
OB37:0100  22 22 22 22 22 22 22 22-22 22 22 22 22 22 22 22  """"""""""
OB37:0110  22 22 22 22 22 22 22 22-22 22 22 22 22 22 22 22  """"""""""
OB37:0120  33 33 33 33 33 33 33 33-33 33 33 33 33 33 33 33  3333333333333333
OB37:0130  33 33 33 33 33 33 33 33-33 33 33 33 33 33 33 33  3333333333333333
OB37:0140  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDDD
OB37:0150  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDDD
OB37:0160  80 3E 2F 9A 00 74 05 F6-C7 02 75 48 89 3E F3 99  .>/..t....uH.>..
OB37:0170  FF 06 F3 99 C6 06 F5 99-FF C6 06 F6 99 00 E8 99  .....
-S 118 127 33
OB37:0120
OB37:0121
OB37:0122
OB37:0123
OB37:0124
OB37:0125
OB37:0126
OB37:0127
-
```

## 4.5 Input and Output of Data

- ▶ **INPUT** Command (I)

The INPUT command **read** data from an **input** port of the 64K byte-wide ports of 8088 I/O.

I ADDRESS

- ▶ e.g.

-I 61 (←↓)
4D

- ▶ The contents of the port at I/O address  $0061_{16}$  are  $4D_{16}$

## 4.5 Input and Output of Data

- ▶ **OUTPUT** Command (O)

The OUTPUT command **write** data to an **output** port of the 64K byte-wide ports of 8088 I/O.

### O ADDRESS BYTE

- ▶ e.g.

```
-O 61 4F (↵)
```

- ▶ This command causes the value  $4F_{16}$  to be written into the byte-wide output port at address  $0061_{16}$

# 4.6 Hexadecimal Addition and Subtraction

- ▶ **HEXADECIMAL** Command (H)

The HEXADECIMAL command provides the ability to add and subtract hexadecimal numbers.

H NUM1 NUM2

- ▶ e.g.

```
-H ABC0 0FFF (←)
BBBF 9BC1
```

```
-H BBBF A (←)
BBC9 BBB5
```

\*Both number and results are limited to **four** hexadecimal digits.

# 4.6 Hexadecimal Addition and Subtraction

- ▶ EXAMPLE

Use the H command to find the negative of the number  $0009_{16}$ .

- ▶ Solution:

```
-H 0 9 (↵)
0009 FFF7
```

- ▶  $FFF7_{16}$  is the negative of  $9_{16}$  expressed in 2's complement form.

# 4.6 Hexadecimal Addition and Subtraction

## ▶ EXAMPLE

If a byte of data is located at physical address  $02A34_{16}$  and the data segment register contains  $0150_{16}$ , what value must be loaded into the source index register such that DS:SI points to the byte storage location?

## ▶ Solution:

```
-H 2A34 1500 (↙)
3F34 1534
```

▶ This shows that SI must be loaded with the value  $1534_{16}$ .

# 4.7 Loading, Verifying and Saving Machine Language Program

- ▶ An example to load an instruction

`MOV BL, AL`

- ▶ The machine code is `88C316`

```
-E CS:100 88 C3 (←↓)
-D CS:100 101 (←↓)
1342:0100 88 C3
```

# 4.7 Loading, Verifying and Saving Machine Language Program

- ▶ **UNASSEMBLE** Command (U)

The UNASSEMBLE command converts machine code instructions to their equivalent assembly language source statement.

U [STARTING\_ADDRESS [ENDING\_ADDRESS] ]

- ▶ e.g.

```
-U CS:100 101 (↵)
1342:0100 88C3 MOV BL, AL
```

Why CS not DS ?

Hexadecimal

Instruction



# 4.7 Loading, Verifying and Saving Machine Language Program

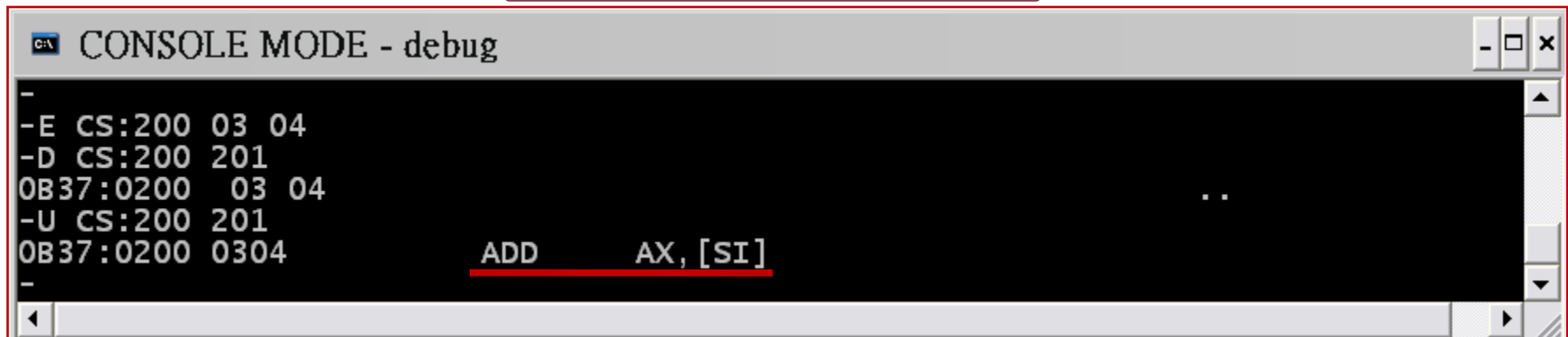
## ▶ EXAMPLE

Use a sequence of commands to load, verify loading, and unassemble the machine code instruction 0304H. Load the instruction at address CS:200.

## ▶ Solution:

Why CS not DS ?

```
-E CS:200 03 04 (↵)
-D CS:200 201 (↵)
-U CS:200 201 (↵)
ADD AX, [SI]
```



```
CONSOLE MODE - debug
-
-E CS:200 03 04
-D CS:200 201
0B37:0200 03 04
-U CS:200 201
0B37:0200 0304      ADD   AX, [SI]
```

# 4.7 Loading, Verifying and Saving Machine Language Program

- ▶ **WRITE** Command (W)

The WRITE command gives the ability to save data stored in memory on a diskette.

W [START\_ADDRESS [DRIVE START\_SECTOR NUM\_SECTOR] ]

- ▶ e.g.

```
-W CS:200 1 10 1 (↵)
-W 200 1 10 1 (↵)
```

Drive B

1 Sector = 512 Byte

\* Be caution in saving program in a disk, especially the hard drive.

# 4.7 Loading, Verifying and Saving Machine Language Program

- ▶ **LOAD** Command (L)

The LOAD command gives the ability to reload memory from a diskette.

L [START\_ADDRESS [DRIVE START\_SECTOR NUM\_SECTOR] ]

- ▶ e.g.

```
-L CS:300 1 10 1 (↵)
```

- The reloading of the instruction can be verified by U command

- ▶ e.g.

```
-U CS:300 301 (↵)  
1342:300 301 ADD AX, [SI]
```

# 4.7 Loading, Verifying and Saving Machine Language Program

## ▶ EXAMPLE

Enter the machine code of the block move program. The program is to be loaded into memory starting at address CS:100. Verify, unassemble, and save the code.

## ▶ Solution:

```
-E CS:100 B8 00 20 8E D8 BE  
00 01 BF 20 01 B9 10  
00 8A 24 88 25 46 (↵)  
-D CS:100 117(↵)  
-U CS:100 117(↵)  
-W CS:100 1 100 1 (↵)
```

# 4.7 Loading, Verifying and Saving Machine Language Program

```
C:\> CONSOLE MODE - debug
-
-E CS:100 B8 00 20 8E D8 BE 00 01 BF 20 01 B9 10 00 8A 24
-E CS:110 88 25 46 47 49 75 F7 90
-D CS:100 117
OB37:0100 B8 00 20 8E D8 BE 00 01-BF 20 01 B9 10 00 8A 24 .. . . . . . . . . . $
OB37:0110 88 25 46 47 49 75 F7 90 .%FGIU..
-U CS:100 117
OB37:0100 B80020      MOV      AX,2000
OB37:0103 8ED8      MOV      DS,AX
OB37:0105 BE0001      MOV      SI,0100
OB37:0108 BF2001      MOV      DI,0120
OB37:010B B91000      MOV      CX,0010
OB37:010E 8A24      MOV      AH,[SI]
OB37:0110 8825      MOV      [DI],AH
OB37:0112 46      INC      SI
OB37:0113 47      INC      DI
OB37:0114 49      DEC      CX
OB37:0115 75F7      JNZ      010E
OB37:0117 90      NOP
-W CS:100 0 100 1
-
```

# 4.7 Loading, Verifying and Saving Machine Language Program

## ▶ **NAME** Command (N)

The NAME command, along with the WRITE command, gives the ability to save a program on the diskette under a file name.

**N FILE NAME**

- The BX, CX registers must be updated to identify the size of the program that is to be saved in the file.

**(BX CX) = number of bytes**

**Because of programs are small → set BX = 0000H**

- After BX, CX registers have been initialized, the write command is used to saved the program.
- To reload the program, the command sequence is

**N FILE NAME**

**L [STARTING ADDRESS]**

# 4.7 Loading, Verifying and Saving Machine Language Program

## ▶ EXAMPLE

Save a machine code program into a file.

## ▶ Solution:

```
-N A:BLK.1 (↵) ; Give a file name in disk A
-R CX (↵) ; Give a program size of 1816 bytes
CX XXXX
:18
-R BX (↵)
BX XXXX
:0 (↵)
W CS:100 (↵) ; Save the program in disk A
```

# 4.7 Loading, Verifying and Saving Machine Language Program

## ▶ EXAMPLE

Reload a program into memory.

## ▶ Solution:

```
-N A:BLK.1 (↵) ; Give a file name in disk A  
-L CS:100 (↵) ; Load the program name BLK.1 in disk A
```

```
C:\DOS>REN A:BLK.1 BLK.EXE (↵) ; Rename the file
```

```
C:\DOS>DEBUG A:BLK.EXE (↵) ; Load the program directly into  
memory
```

```
C:\DOS>A:BLK.EXE (↵) ; Run the program
```



## 4.8 Assembling Instructions with the Assemble Command

- ▶ **ASSEMBLE** Command (A)

The ASSEMBLE command let us automatically assemble the instructions of a program.

A [STARTING\_ADDRESS]

- ▶ e.g.

→ The program will be saved in memory starting from this location

```
-A CS:100 (↵)
1342:0100 _
1342:0100 ADD [BX+SI+1234], AX
(↵)
1342:0104 _
-D CS:100 103 (↵)
```

← Your instruction

# 4.8 Assembling Instructions with the Assemble Command

- ▶ EXAMPLE

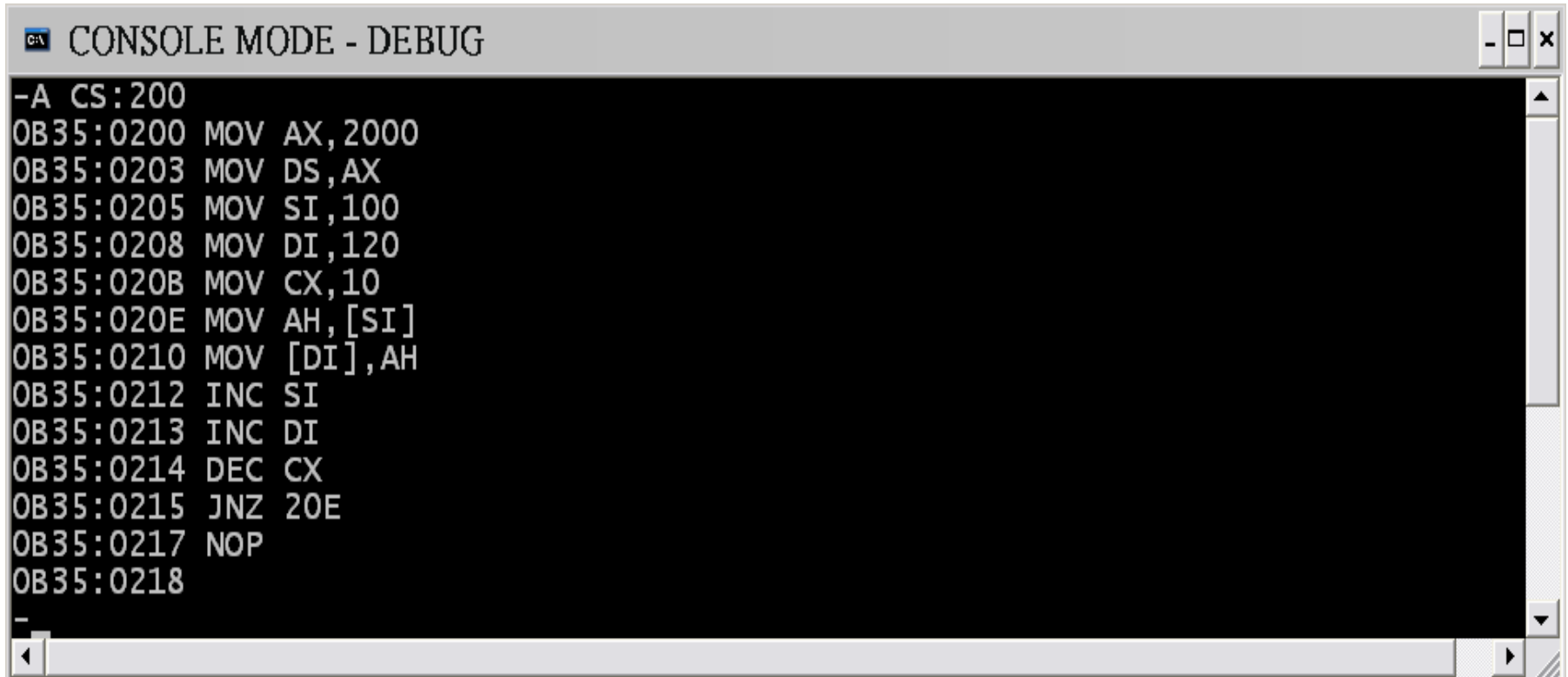
Assemble a complete program with the ASSEMBLE command.

- ▶ Solution:

```
-A CS:200 (↵)
0B35:0200 MOV AX, 2000 (↵)
0B35:0203 MOV DS, AX (↵)
0B35:0205 MOV SI, 100 (↵)
. . . .
. . . .
0B35:0217 NOP (↵)
0B35:0218 (↵)
```

## 4.8 Assembling Instructions with the Assemble Command

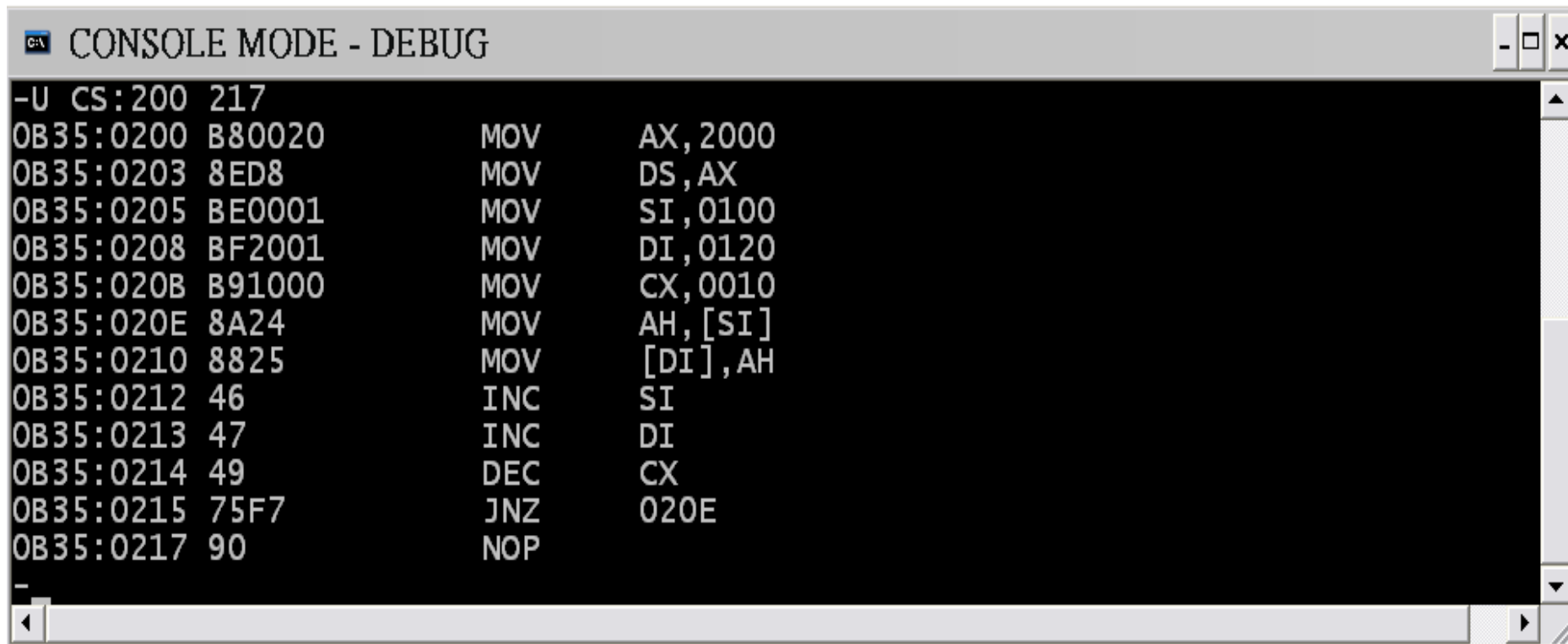
- ▶ Assemble a program with ASSEMBLE command

A screenshot of a debugger console window titled "CONSOLE MODE - DEBUG". The window contains a list of assembly instructions with their addresses and hex values. The instructions are: -A CS:200, 0B35:0200 MOV AX,2000, 0B35:0203 MOV DS,AX, 0B35:0205 MOV SI,100, 0B35:0208 MOV DI,120, 0B35:020B MOV CX,10, 0B35:020E MOV AH,[SI], 0B35:0210 MOV [DI],AH, 0B35:0212 INC SI, 0B35:0213 INC DI, 0B35:0214 DEC CX, 0B35:0215 JNZ 20E, 0B35:0217 NOP, and 0B35:0218. The window has a standard Windows-style title bar with minimize, maximize, and close buttons.

```
CONSOLE MODE - DEBUG
-A CS:200
0B35:0200 MOV AX,2000
0B35:0203 MOV DS,AX
0B35:0205 MOV SI,100
0B35:0208 MOV DI,120
0B35:020B MOV CX,10
0B35:020E MOV AH,[SI]
0B35:0210 MOV [DI],AH
0B35:0212 INC SI
0B35:0213 INC DI
0B35:0214 DEC CX
0B35:0215 JNZ 20E
0B35:0217 NOP
0B35:0218
```

# 4.8 Assembling Instructions with the Assemble Command

- ▶ Unassemble a program with **UNASSEMBLE** command (the reverse operation of assemble)



```
C:\> UNASSEMBLE CS:200 217
OB35:0200 B80020      MOV     AX,2000
OB35:0203 8ED8          MOV     DS,AX
OB35:0205 BE0001      MOV     SI,0100
OB35:0208 BF2001      MOV     DI,0120
OB35:020B B91000      MOV     CX,0010
OB35:020E 8A24          MOV     AH,[SI]
OB35:0210 8825          MOV     [DI],AH
OB35:0212 46           INC     SI
OB35:0213 47           INC     DI
OB35:0214 49           DEC     CX
OB35:0215 75F7          JNZ     020E
OB35:0217 90           NOP
```

## 4.9 Executing Instructions and Programs with the TRACE and GO command

### ▶ TRACE Command (T)

The TRACE command provides the programmer with the ability to execute the program **one instruction** at a time.

T [=STARTING\_ADDRESS] [NUMBER]

### ▶ e.g.

Important

The address of the first instruction

```
-T = CS:100 (←) // executes one instruction
```

```
-T (←) // executes one instruction starting at  
CS:IP
```

```
-T = CS:100 3 (←) // executes three instructions
```

## 4.9 Executing Instructions and Programs with the TRACE and GO command

- ▶ EXAMPLE

Load and trace a program.

- ▶ Solution:

```
-L CS:100 1 10 1 (↵) // or -A CS:100 (↵)
-U 100 101 (↵)
-R AX (↵)
AX 0000
:1111 (↵)
-R SI (↵)
SI 0000
:1234 (↵)
-E DS:1234 22 22 (↵)
-T =CS:100 (↵)
```

# 4.9 Executing Instructions and Programs with the TRACE and GO command

```
CONSOLE MODE - DEBUG
-
-L CS:100 0 10 1
-U 100 101
OB35:0100 0304          ADD     AX,[SI]
-R AX
AX 0000
:1111
-R SI
SI 0000
:1234
-E DS:1234 22 22
R
AX=1111  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=1234  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0100  NV UP EI PL NZ NA PO NC
OB35:0100 0304          ADD     AX,[SI]          DS:1234=222
-D DS:1234 1235
OB35:1230          22 22          ""
-T =CS:100
AX=3333  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=1234  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0102  NV UP EI PL NZ NA PE NC
OB35:0102 0000          ADD     [BX+SI],AL      DS:1234=22
-
```

## 4.9 Executing Instructions and Programs with the TRACE and GO command

### ▶ GO Command (G)

The GO command is typically used to run programs that are already working or to execute programs in the later stages or debugging.

`G [=STARTING_ADDRESS [BREAKPOINT ADDRESS LIST] ]`

▶ e.g.

```
-G =CS:200 217 (↵)
```

```
-G =CS:100 (↵)
```

```
-G (↵) // start execution from CS:IP
```

The list of addresses that the execution will stop on them and display internal registers information (maximum 10 breakpoints )



## 4.9 Executing Instructions and Programs with the TRACE and GO command

### ▶ EXAMPLE

Use GO command to execute a program and examine the result.

### ▶ Solution:

```
-N A:BLK.EXE (↵) ; Define the program file to be loaded
-L CS:200 (↵) ; Load the program at CS:200
-R DS (↵)
DS 1342
:2000 (↵) ; Define the data segment address
-F DS:100 10F FF (↵) ; Fill memory with FF
-F DS:120 12F 00 (↵) ; Fill memory with 00
-R DS (↵)
DS 2000
:1342 ; Store data segment with 134216
```

## *4.9 Executing Instructions and Programs with the TRACE and GO command*

### ▶ Solution (continued) :

```
-R (↵) ; Show data register status
-U CS:200 217 (↵) ; Unassemble the program
-G =CS:200 20E (↵) ; Execute the program to CS:20E
-G =CS:20E 215 (↵) ; Execute the program to CS:215
-D DS:100 10F (↵) ; Display memory at DS:100
-D DS:120 12F (↵) ; Display memory at DS:120
-G =CS:215 217 (↵) ; Execute the program to CS:217
-D DS:100 10F (↵) ; Display memory at DS:100
-D DS:120 12F (↵) ; Display memory at DS:120
```

# 4.9 Executing Instructions and Programs with the TRACE and GO command

```
C:\ CONSOLE MODE - DEBUG
-
-N A:BLK.EXE
-L CS:100
-R DS
DS 0B99
:2000
-F DS:100 10F FF
-F DS:120 12F 00
-D DS:100 10F
2000:0100  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
-D DS:120 12F
2000:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
-R DS
DS 2000
:1342
-R
AX=0000  BX=FFFF  CX=FE18  DX=0000  SP=FFFE  BP=0000  SI=1234  DI=0000
DS=1342  ES=0B99  SS=0B99  CS=0B99  IP=0100  NV UP EI PL NZ NA PE NC
0B99:0100  B80020          MOV     AX,2000
-
```

# 4.9 Executing Instructions and Programs with the TRACE and GO command

```
CONSOLE MODE - DEBUG
-U CS:100 117
0B99:0100 B80020      MOV     AX,2000
0B99:0103 8ED8        MOV     DS,AX
0B99:0105 BE0001      MOV     SI,0100
0B99:0108 BF2001      MOV     DI,0120
0B99:010B B91000      MOV     CX,0010
0B99:010E 8A24        MOV     AH,[SI]
0B99:0110 8825        MOV     [DI],AH
0B99:0112 46          INC     SI
0B99:0113 47          INC     DI
0B99:0114 49          DEC     CX
0B99:0115 75F7        JNZ     010E
0B99:0117 90          NOP
-G =CS:100 10E

AX=2000  BX=FFFF  CX=0010  DX=0000  SP=FFFE  BP=0000  SI=0100  DI=0120
DS=2000  ES=0B99  SS=0B99  CS=0B99  IP=010E  NV UP EI PL NZ NA PE NC
0B99:010E 8A24        MOV     AH,[SI]                      DS:0100=FF
-G =CS:10E 115

AX=FF00  BX=FFFF  CX=000F  DX=0000  SP=FFFE  BP=0000  SI=0101  DI=0121
DS=2000  ES=0B99  SS=0B99  CS=0B99  IP=0115  NV UP EI PL NZ AC PE NC
0B99:0115 75F7        JNZ     010E
-
```

# 4.9 Executing Instructions and Programs with the TRACE and GO command

```
CONSOLE MODE - DEBUG
-
-D DS:100 10F
2000:0100  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
-D DS:120 12F
2000:0120  FF 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
-G =CS:115 117

AX=FF00  BX=FFFF  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0110  DI=0130
DS=2000  ES=0B99  SS=0B99  CS=0B99  IP=0117  NV UP EI PL ZR NA PE NC
0B99:0117 90          NOP
-D DS:100 10F
2000:0100  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
-D DS:120 12F
2000:0120  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
```

## 4.10 Debugging a Program

- ▶ Errors in a program are also referred to as *bugs*; *the* process of removing them is called *debugging*.
- ▶ Two types of errors
  - Syntax error
  - Execution error
- ▶ A syntax error is an error caused by not following the rules for coding or entering an instruction. These types of errors are typically identified by the microcomputer and signalled to user with an error message
- ▶ In the DEBUG environment, the TRACE command is usually used to debug execution errors.

# 4.10 Debugging a Program

- ▶ Review of the DEBUG commands
  - Register: R [register name]
  - Quit: Q
  - Dump: D[address]
  - Enter: E address [list]
  - Fill: F st. address end address list
  - Move: M st. addr. end addr. dest. Addr.
  - Compare: C st. addr. end addr. dest. Addr.
  - Search: S st. address end address list
  - Input: I address
  - Output: O address, byte
  - Hex Add/Subtract: H num1,num2
  - Assemble: A [starting address]
  - Unassemble: U [starting address ending address]
  - Name: N file name
  - Write: W [st. addr. [drive st. sector no. of sectors]]
  - Load: L [st. addr. [drive st. sector no. of sectors]]
  - Trace: T [=address] [number]
  - Go: G [= starting address [breakpoint address ...]]

# H.W. #4

- Solve the following problems from Chapter 4 from the course textbook:

2, 5, 10, 15, 20, 23, 24, 26, 28, 30



# **CPE 408330**

## **Assembly Language and Microprocessors**

### **Chapter 5: 8088/8086 Microprocessor Programming – Integer Instructions and Computations**

[Computer Engineering Department,  
Hashemite University]

# Lecture Outline

- ▶ 5.1 **Data-Transfer** Instructions
- ▶ 5.2 **Arithmetic** Instructions
- ▶ 5.3 **Logic** Instructions
- ▶ 5.4 **Shift** Instructions
- ▶ 5.5 **Rotate** Instructions

# 5.1 Data-Transfer Instructions

- ▶ The data-transfer functions provide the ability to move data either between its **internal registers** or between an internal register and a storage location in **memory**.
- ▶ The data-transfer functions include
  - MOV (Move byte or word)
  - XCHG (Exchange byte or word)
  - XLAT (Translate byte)
  - LEA (Load effective address)
  - LDS (Load data segment)
  - LES (Load extra segment)

# 5.1 Data-Transfer Instructions - Move Instruction

## ▶ The MOVE Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
MOV	Move	MOV D,S	(S) → (D)	None

(a)

Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg-reg	Reg16
Seg-reg	Mem16
Reg16	Seg-reg
Memory	Seg-reg

(b)

Allowed operands for MOV instruction

- Used to move (copy) data between:
  - Registers
  - Register and memory
  - Immediate operand to a register or memory

- General format:

**MOV D,S**

- Operation: Copies the content of the source to the destination

(S) → (D)

- Source contents unchanged
- Flags unaffected
- Allowed operands

Register

Memory

Accumulator (AH,AL,AX)

Immediate operand (Source only)

Segment register (Seg-reg)

- Examples:

**MOV [SUM],AX**

(AL) → (address SUM)

(AH) → (address SUM+1)

# *5.1 Data-Transfer Instructions -*

## **Move Instruction**

- ▶ The MOVE Instruction

e.g. `MOV DX, CS`  
`MOV [SUM], AX`

- ▶ Note that the MOV instruction cannot transfer data directly between external memory.

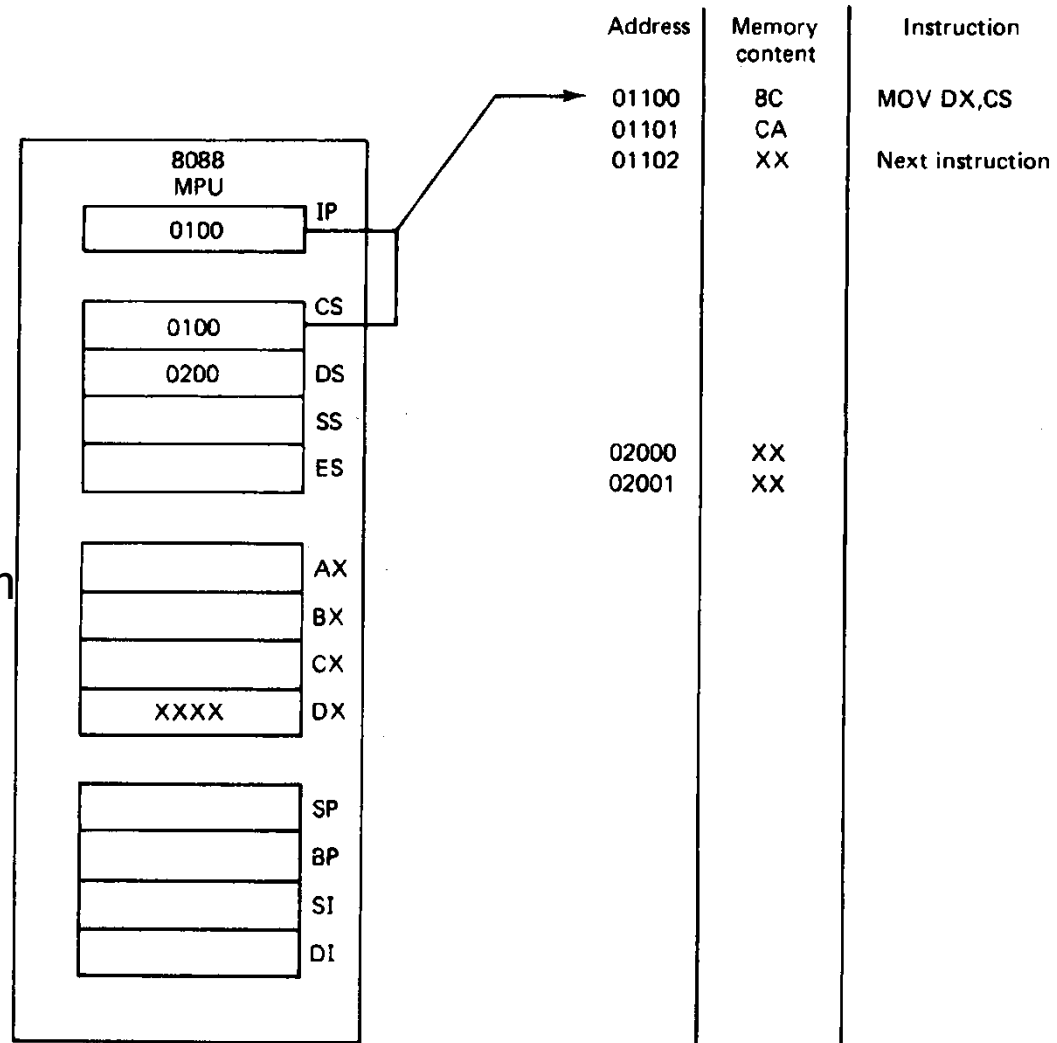
# 5.1 Data-Transfer Instructions - Move Instruction

## MOV DX, CS

Source = CS → word data  
 Destination = DX → word data  
 Operation: (CS) → (DX)

- State before fetch and execution

CS:IP = 0100:0100 = 01100H  
 Move instruction code = 8CCA  
 (01100H) = 8CH  
 (01101H) = CAH  
 (CS) = 0100H  
 (DX) = XXXX → don't care state



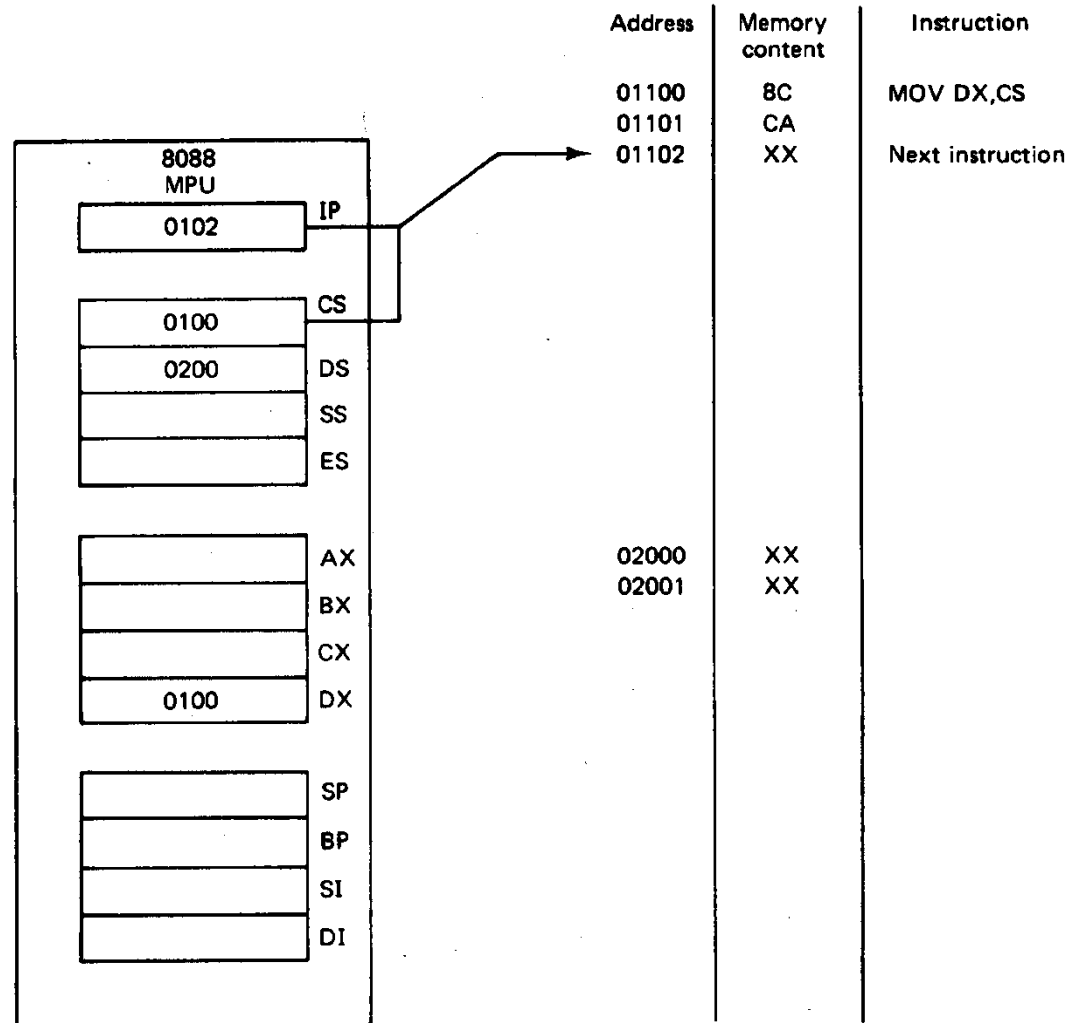
Before execution

(c)

# 5.1 Data-Transfer Instructions - Move Instruction

MOV DX, CS

- State after execution  
 $CS:IP = 0100:0102 = 01102H$   
 $01002H \rightarrow$  points to next sequential instruction  
 $(CS) = 0100H$   
 $(DX) = 0100H \rightarrow$  Value in CS copied into DX  
 Value in CS unchanged



(d)

After execution

# 5.1 Data-Transfer Instructions - Move Instruction

## ▶ EXAMPLE

What is the effect of executing the instruction  
`MOV CX, [SOURCE_MEM]`

Where `SOURCE_MEM` equal to  $20_{16}$  is a memory location offset relative to the current data segment starting at  $1A00_{16}$ .

## ▶ Solution:

$((DS)0 + 20_{16}) \rightarrow (CL)$

$((DS)0 + 20_{16} + 1_{16}) \rightarrow (CH)$

Therefore CL is loaded with the contents held at memory address

$$1A000_{16} + 20_{16} = 1A020_{16}$$

and CH is loaded with the contents of memory address

$$1A000_{16} + 20_{16} + 1_{16} = 1A021_{16}$$



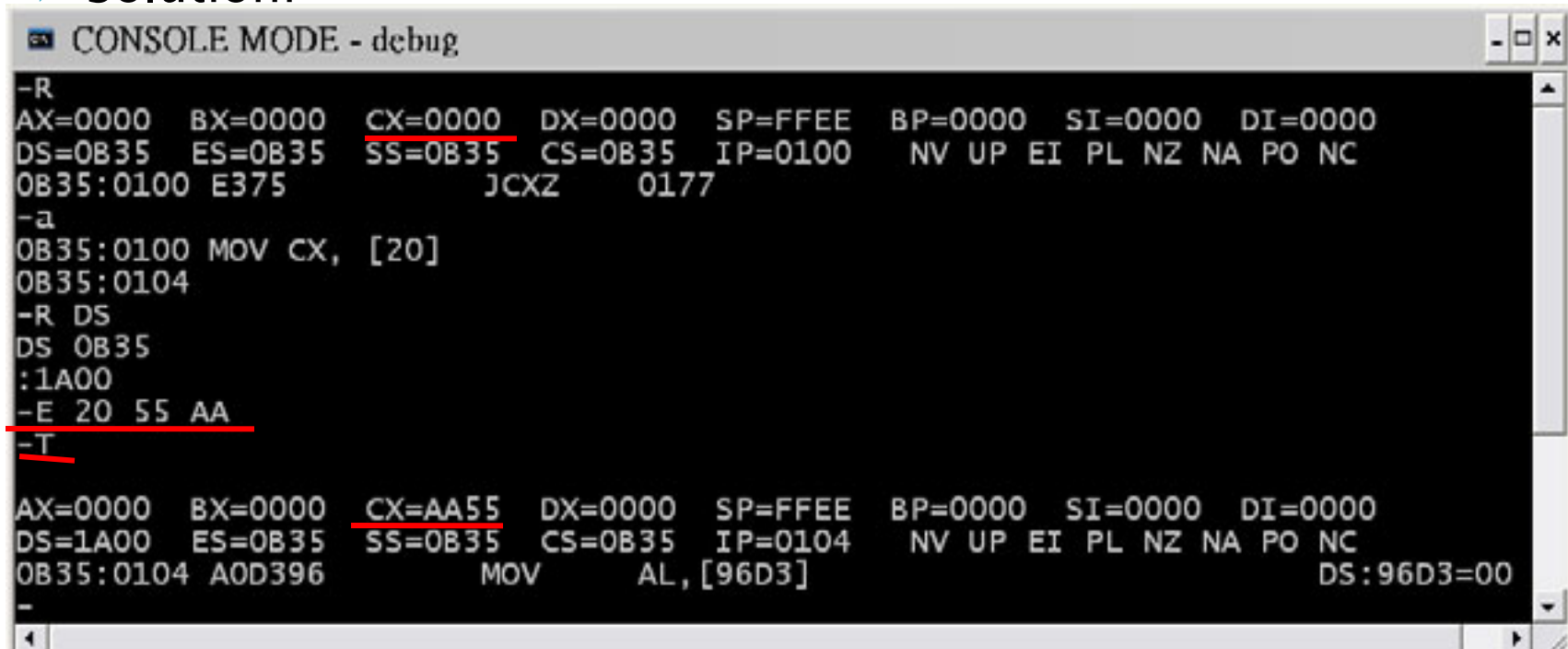
# 5.1 Data-Transfer Instructions - Move Instruction

## ▶ EXAMPLE

Use the DEBUG to verify

MOV CX,[20]  
DS = 1A00, (DS:20) = AA55H  
(1A00:20) → (CX)

## ▶ Solution:



```
CONSOLE MODE - debug
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0100  NV UP EI PL NZ NA PO NC
0B35:0100 E375          JCXZ    0177
-a
0B35:0100 MOV CX, [20]
0B35:0104
-R DS
DS 0B35
:1A00
-E 20 55 AA
-T
AX=0000  BX=0000  CX=AA55  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A00  ES=0B35  SS=0B35  CS=0B35  IP=0104  NV UP EI PL NZ NA PO NC
0B35:0104 A0D396          MOV    AL, [96D3]  DS:96D3=00
-
```

# 5.1 Data-Transfer Instructions - Move Instruction

- Example—Initialization of internal registers with immediate data and address information

- **DS**, **ES**, and **SS** registers initialized from immediate data via **AX**

IMM16 → (AX)

(AX) → (DS) & (ES) = 2000H

IMM16 → (AX)

(AX) → (SS) = 3000H

- Data registers initialized

IMM16 → (AX) = 0000H

(AX) → (BX) = 0000H

IMM16 → (CX) = 000AH and (DX) = 0100H

- Index register initialized from immediate operations

IMM16 → (SI) = 0200H and (DI) = 0300H

DS,ES to 2000H

SS to 3000H

AX, BX to 0H

CX to 0AH

SI to 200

DI to 300

**MOV AX,2000H**

**MOV DS,AX**

**MOV ES,AX**

**MOV AX,3000H**

**MOV SS,AX**

**MOV AX,0H**

**MOV BX,AX**

**MOV CX,0AH**

**MOV DX,100H**

**MOV SI,200H**

**MOV DI,300H**

# 5.1 Data-Transfer Instructions - Exchange Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
XCHG	Exchange	XCHG D,S	(D) ↔ (S)	None

(a)

Destination	Source
Accumulator	Reg16
Memory	Register
Register	Register
Register	Memory

(b)

Allowed operands for XCHG instruction

- Used to exchange the data between two data registers or a data register and memory

- General format:

XCHG D,S

- Operation: **Swaps** the content of the source and destination

- Both source and destination change

(S) → (D)

(D) → (S)

- Flags unaffected

- Special accumulator destination version executes faster

- Examples:

XCHG AX,DX

(Original value in AX) → (DX)

(Original value in DX) → (AX)

# 5.1 Data-Transfer Instructions - Exchange Instruction

XCHG [SUM],BX

Note: SUM = 1234

Source = BX → word data

Destination = memory offset

SUM → word data

Operation: (SUM) → (BX)

(BX) → (SUM)

What is the general logical address of the destination operand?

- State before fetch and execution

CS:IP = 1100:0101 = 11101H

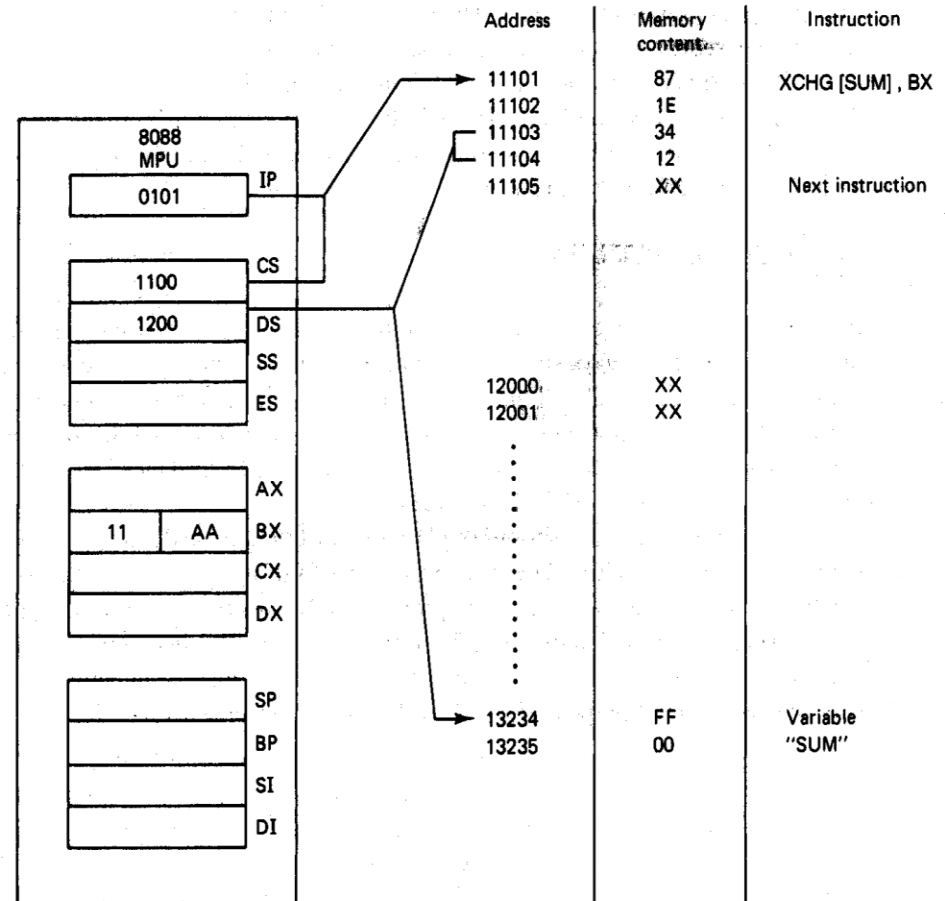
Move instruction code = 871E3412H

(01104H,01103H) = 1234H = SUM

(DS) = 1200H

(BX) = 11AA

(DS:SUM) = (1200:1234) = 00FFH



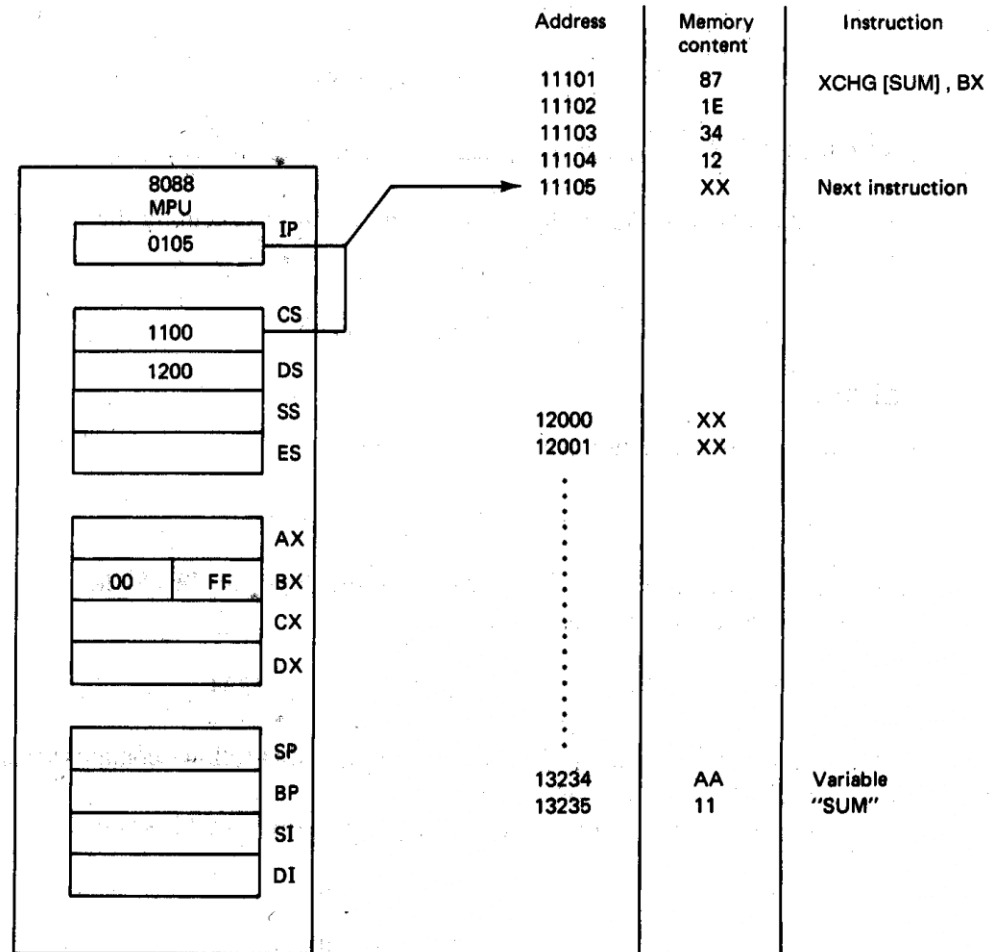
(c)

Before execution

# 5.1 Data-Transfer Instructions - Exchange Instruction

## XCHG [SUM],BX

- State after execution  
 $CS:IP = 1100:0105 = 11105H$   
 $11005H \rightarrow$  points to next sequential instruction
- Register updated  
 $(BX) = 00FFH$
- Memory updated  
 $(1200:1234) = AAH$   
 $(1200:1235) = 11H$



(d)

After execution

# 5.1 Data-Transfer Instructions - Exchange Instruction

- ▶ EXAMPLE

Use the DEBUG to verify the previous example.

- ▶ Solution:

```
CONSOLE MODE - DEBUG
-
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0100  NV UP EI PL NZ NA PO NC
OB35:0100 8B0E2000      MOV      CX,[0020]          DS:0020=FFF
-A 1100:101
1100:0101 XCHG [1234], BX
1100:0105
-R BX
BX 0000
:11AA
-R DS
DS 0B35
:1200
-R CS
CS 0B35
:1100
-R IP
IP 0100
:101
-
```

# 5.1 Data-Transfer Instructions - Exchange Instruction

- ▶ Solution (cont'd):

```
C:\ CONSOLE MODE
-
-R
AX=0000 BX=11AA CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1200 ES=0B35 SS=0B35 CS=1100 IP=0101 NV UP EI PL NZ NA PO NC
1100:0101 871E3412 XCHG BX,[1234] DS:1234=11AA
-E 1234 FF 00
-U 101 104
1100:0101 871E3412 XCHG BX,[1234]
-T
AX=0000 BX=00FF CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1200 ES=0B35 SS=0B35 CS=1100 IP=0105 NV UP EI PL NZ NA PO NC
1100:0105 0000 ADD [BX+SI],AL DS:00FF=00
-D 1234 1235
1200:1230 AA 11 ..
-Q
C:\>
```

# 5.1 Data-Transfer Instructions - Translate Instruction

## ▶ The XLAT Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
XLAT	Translate	XLAT	$((AL)+(BX)+(DS)0) \rightarrow (AL)$	None

- Translate instruction
  - Used to look up a byte-wide value in a table in memory and copy that value in the AL register
  - General format:

XLAT

- Operation: Copies the content of the element pointed to in the source table in memory to the AL register

$((AL)+(BX) + (DS)0) \rightarrow (AL)$

Where:

(DS)0 = Points to the active data segment

(BX) = Offset to the first element in the table

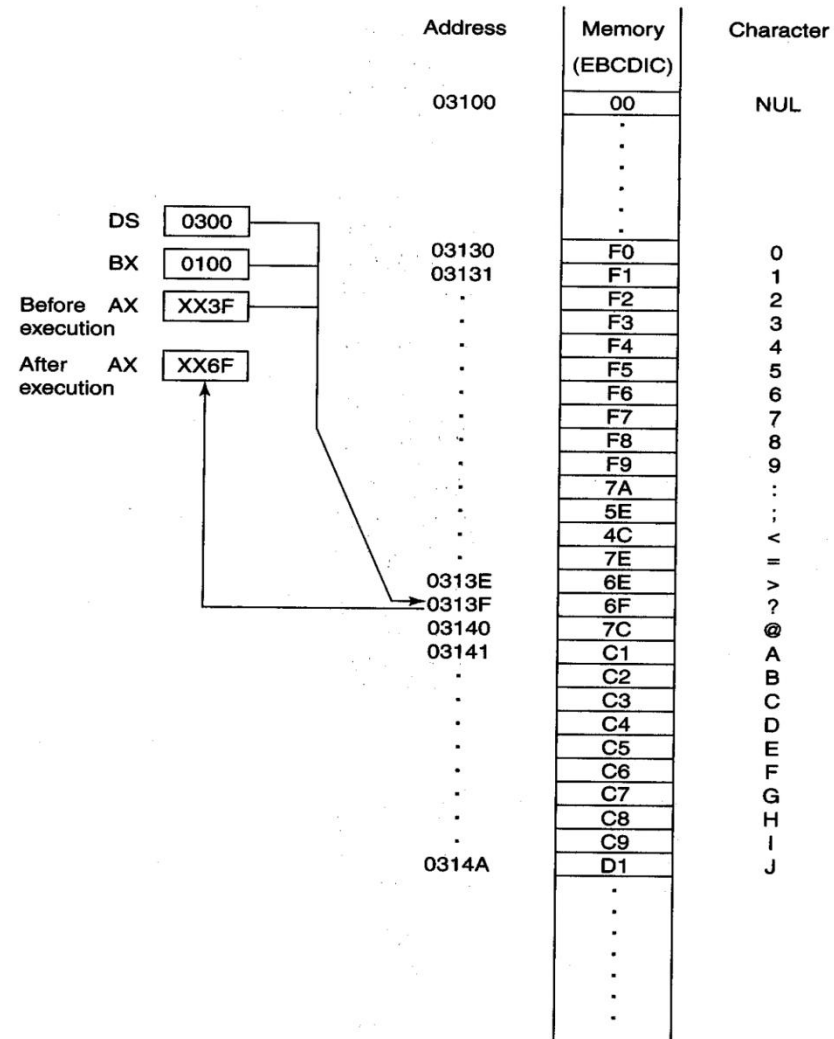
(AL) = Displacement to the element of the table that is to be accessed\*

\*8-bit value limits table size to 256 elements



# 5.1 Data-Transfer Instructions - Translate Instruction

- Application: ASCII to EBCDIC Translation
  - Fixed EBCDIC table coded into memory starting at offset in BX
  - Individual EBCDIC codes placed in table at displacement (AL) equal to the value of their equivalent ASCII character
  - A = 41H in ASCII, A = C1H in EBCDIC
  - Place the value C1H in memory at address  $(41H + (BX) + (DS)0)$ , etc.
  - Example
- XLAT  
 (DS) = 0300H  
 (BX) = 0100H  
 (AL) = 3FH → 6FH = ? (Question mark)



# 5.1 Data-Transfer Instructions - Load Effective Address and Load Full Pointer Instructions

- The LEA, LDS, and LES Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
LEA	Load effective address	LEA Reg16,EA	EA → (Reg16)	None
LDS	Load register and DS	LDS Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (DS)	None
LES	Load register and ES	LES Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (ES)	None

(a)

- Load effective address instruction
  - Used to load an **address** pointer **offset** from memory into a register.
  - General format:  
**LEA Reg16,EA**
  - Operation:  
EA → (Reg16)
  - Source unaffected:
  - Flags unaffected

# 5.1 Data-Transfer Instructions - Load Effective Address and Load Full Pointer Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
LEA	Load effective address	LEA Reg16,EA	EA → (Reg16)	None
LDS	Load register and DS	LDS Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (DS)	None
LES	Load register and ES	LES Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (ES)	None

(a)

- **Load full pointer**
  - Used to load a full address pointer from memory into a segment register and a register
  - Segment base address
  - Offset
  - General format and operation for LDS
    - **LDS** Reg16,EA
    - (EA) → (Reg16)
    - (EA+2) → (DS)
  - **LES** operates the same, except initializes **ES**

# 5.1 Data-Transfer Instructions - Load Effective Address and Load Full Pointer Instructions

- Example

LDS SI,[200H]

Source = pointer to DS:200H → 32 bits

Destination = SI → word pointer offset

DS → word pointer SBA

Operation: (DS:200H) → (SI)

(DS:202H) → (DS)

- State before fetch and execution

CS:IP = 1100:0100 = 11100H

LDS instruction code = C5360002H

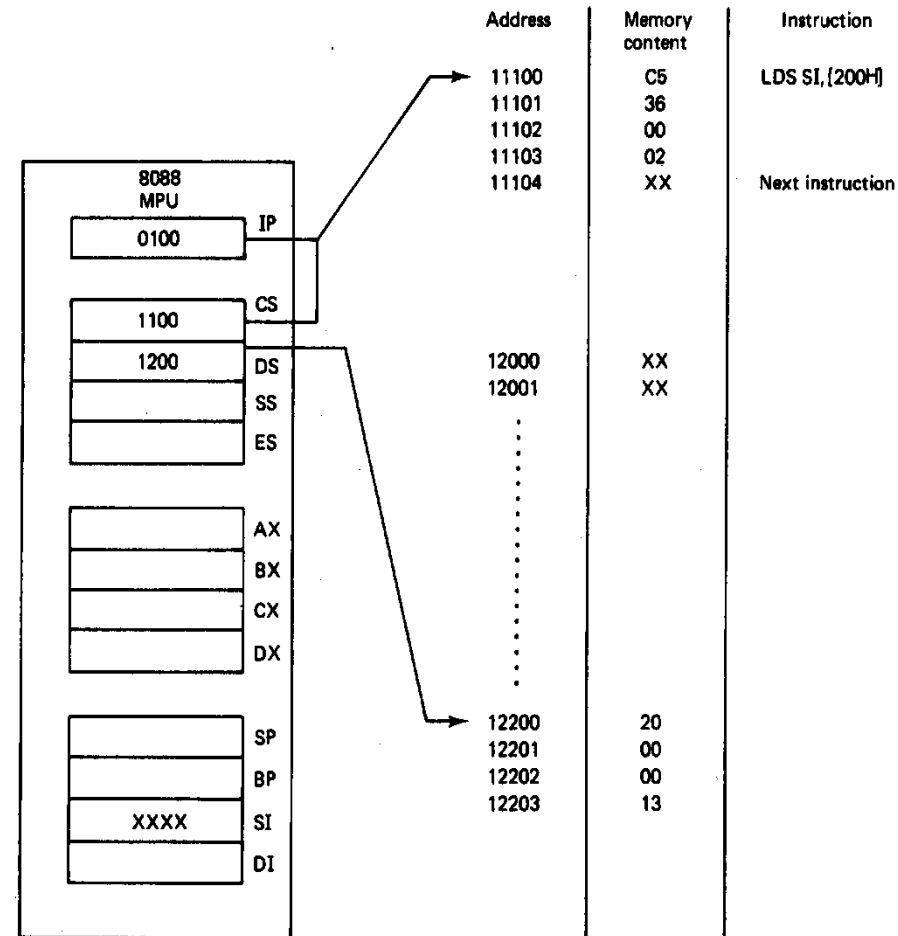
(11102H,11103H) = (EA) = 0200H

(DS) = 1200H

(SI) = XXXX → don't care state

(DS:EA) = 12200H = 0020H = Offset

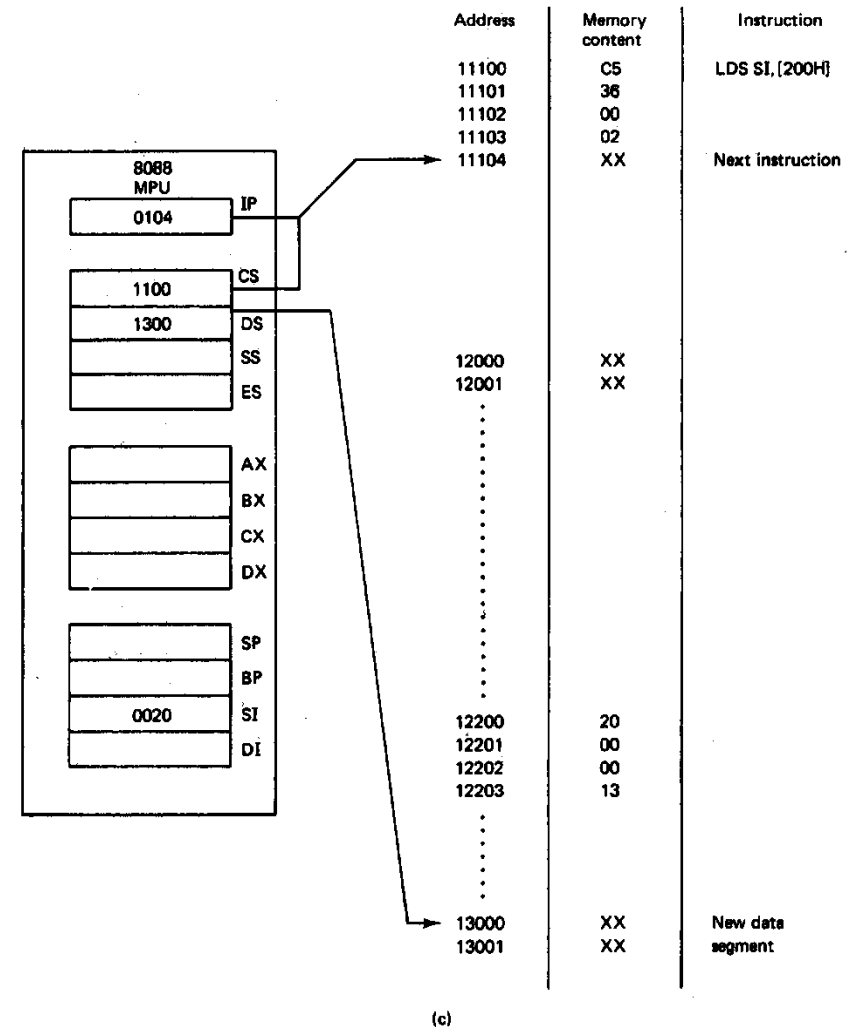
(DS:EA+2) = 12202H = 1300H = SBA



Before execution (b)

# 5.1 Data-Transfer Instructions - Load Effective Address and Load Full Pointer Instructions

- Example
  - State after execution
- CS:IP = 1100:0104 = 11104H  
 01004H → points to next sequential instruction  
 (DS) = 1300H → defines a new data segment  
 (SI) = 0020H → defines new offset into DS



After execution

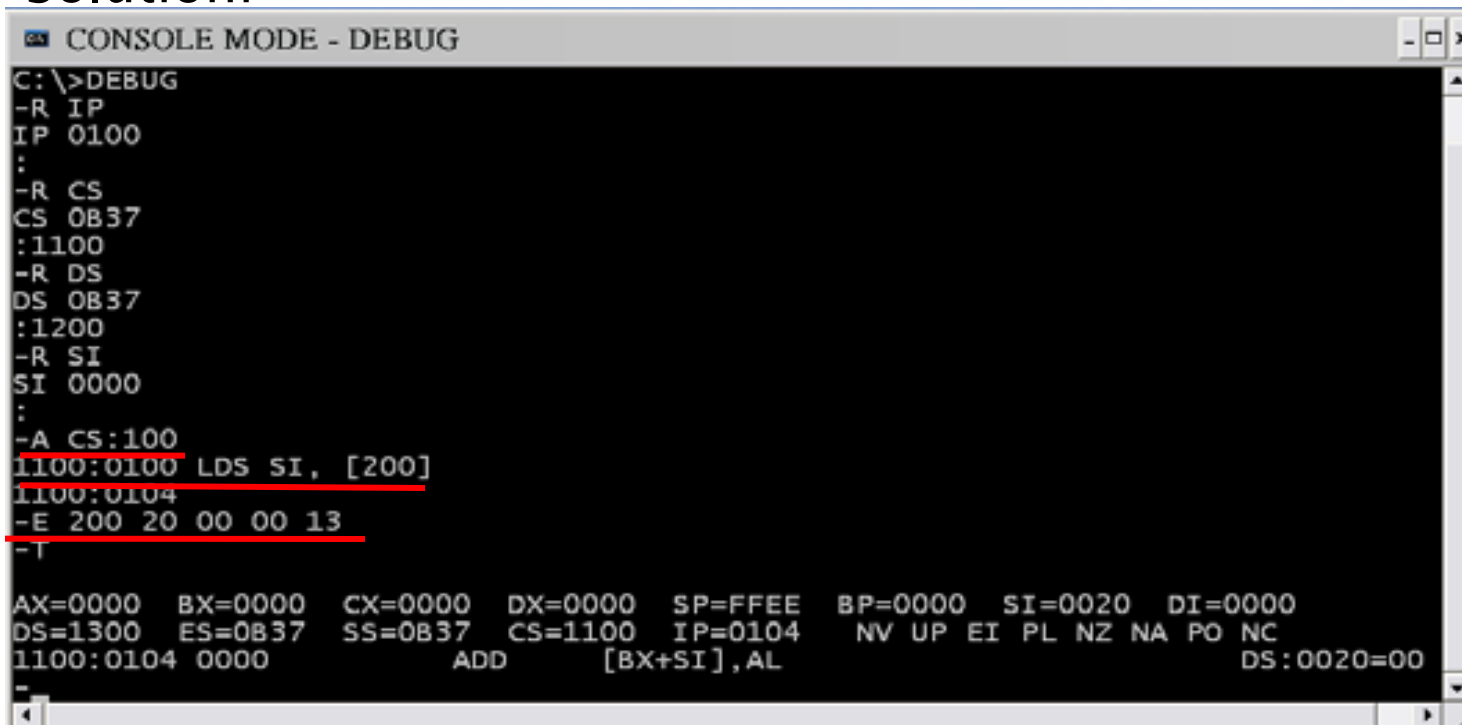
# 5.1 Data-Transfer Instructions - Load Effective Address and Load Full Pointer Instructions

## ▶ EXAMPLE

Verify the following instruction using DEBUG program.

LDS SI, [200H]

## ▶ Solution:



```
CONSOLE MODE - DEBUG
C:\>DEBUG
-R IP
IP 0100
:
-R CS
CS 0B37
:1100
-R DS
DS 0B37
:1200
-R SI
SI 0000
:
-A CS:100
1100:0100 LDS SI, [200]
1100:0104
-E 200 20 00 00 13
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0020 DI=0000
DS=1300 ES=0B37 SS=0B37 CS=1100 IP=0104 NV UP EI PL NZ NA PO NC
1100:0104 0000 ADD [BX+SI],AL DS:0020=00
```

# 5.1 Data-Transfer Instructions - Load Effective Address and Load Full Pointer Instructions

## ▶ EXAMPLE

Initializing the internal registers of the 8088 from a table

in memory.

### ▶ Solution:

- DS loaded via AX with **immediate** value using **move** instructions

DATA\_SEG\_ADDR → (AX) → (DS)

- Index register SI loaded with **move** from table

(INIT\_TABLE, INIT\_TABLE+1) → SI

- DI and ES are loaded with load **full pointer** instruction

(INIT\_TABLE+2, INIT\_TABLE+3) → DI

(INIT\_TABLE+4, INIT\_TABLE+5) → ES

- SS loaded from table via AX using **move** instructions

(INIT\_TABLE+6, INIT\_TABLE+7) → AX → (SS)

- Data registers loaded from table with **move** instructions

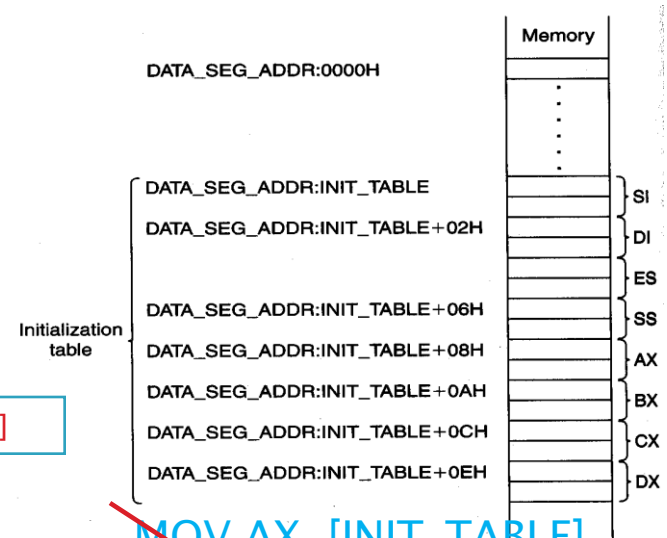
(INIT\_TABLE+8, INIT\_TABLE+9) → AX

(INIT\_TABLE+A, INIT\_TABLE+B) → BX

(INIT\_TABLE+C, INIT\_TABLE+D) → CX

(INIT\_TABLE+E, INIT\_TABLE+F) → DX

LES DI, [INIT\_TABLE+2]



```
MOV AX, [INIT_TABLE]
MOV SS, AX
LDS SI, [INIT_TABLE+02H]
LES DI, [INIT_TABLE+06H]
MOV AX, [INIT_TABLE+0AH]
MOV BX, [INIT_TABLE+0CH]
MOV CX, [INIT_TABLE+0EH]
MOV DX, [INIT_TABLE+10H]
```

## *5.2 Arithmetic Instructions*

- ▶ The arithmetic instructions include
  - Addition
  - Subtraction
  - Multiplication
  - Division
- ▶ Data formats
  - Unsigned binary bytes
  - Signed binary bytes
  - Unsigned binary words
  - Signed binary words
  - Unpacked decimal bytes
  - Packed decimal bytes
  - ASCII numbers

**BCD and ASCII Arithmetic: The microprocessor allows arithmetic manipulation of both BCD (Binary Coded Decimal) and ASCII data.**



# ASCII and BCD

## Processing ASCII Numbers

---

- 8086 provides four instructions
  - aaa** – ASCII adjust after addition
  - aas** – ASCII adjust after subtraction
  - aam** – ASCII adjust after multiplication
  - aad** – ASCII adjust before division
- \* These instructions do not take any operands
  - » Operand is assumed to be in AL

### ASCII representation

- \* Numbers are stored as a string of ASCII characters
  - » Example: 1234 is stored as 31 32 33 34H
  - ASCII Code for 0,1, ...,9: 30H, 31H, ..., 39H

### BCD representation

- \* Unpacked BCD
  - » Example: 1234 is stored as 01 02 03 04H
  - Additional byte is used for sign
    - Sign byte: 00H for + and 80H for –
- \* Packed BCD
  - » Saves space by packing two digits into a byte
  - Example: 1234 is stored as 12 34H

## Processing Packed BCD Numbers

---

- Two instructions to process packed BCD numbers
  - daa** – Decimal adjust after addition
    - Used after **add** or **adc** instruction
  - das** – Decimal adjust after subtraction
    - Used after **sub** or **sbb** instruction
- \* No support for multiplication or division
  - » For these operations
    - Unpack the numbers
    - Perform the operation
    - Repack them

Note: ASCII = Unpacked + 30H

# ASCCI Adjustment Instructions

- ▶ Arithmetic operations are performed on numbers expressed in ASCII format , but we want the final result in decimal. (this saves conversions!)
  - Result must be in AL
- ▶ After the arithmetic operation, an adjustment must be performed on the result to convert it to the equivalent decimal result.
- ▶ This is main principle for all ASCII adjust operations.

# 5.2 Arithmetic Instructions – Addition

## Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
ADD	Addition	ADD D, S	(S) + (D) → (D) Carry → (CF)	OF, SF, ZF, AF, PF, CF
ADC	Add with carry	ADC D, S	(S) + (D) + (CF) → (D) Carry → (CF)	OF, SF, ZF, AF, PF, CF
INC	Increment by 1	INC D	(D) + 1 → (D)	OF, SF, ZF, AF, PF
AAA	ASCII adjust for addition	AAA		AF, CF OF, SF, ZF, PF undefined
DAA	Decimal adjust for addition	DAA		SF, ZF, AF, PF, CF, OF, undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

(a) Addition Instructions. (b) Allowed operands for ADD and ADC (c) Allowed operands for INC

- ▶ Variety of arithmetic instruction provided to support integer addition—core instructions are

- ADD → Addition
- ADC → Add with carry
- INC → Increment

- ▶ Addition Instruction—**ADD**

- ADD format and operation:

**ADD D,S**

**(S) + (D) → (D)**

- Add values in two registers

**ADD AX,BX**

**(AX) + (BX) → (AX)**

- Add a value in memory and a value in a register

**ADD [DI],AX**

**(DS:DI) + (AX) → (DS:DI)**

- Add an immediate operand to a value in a register or memory

**ADD AX,100H**

**(AX) + IMM16 → (AX)**

- ▶ **Flags** updated based on result

- CF, OF, SF, ZF, AF, PF

## 5.2 Arithmetic Instructions – Addition Instructions

### ▶ EXAMPLE

Assume that the AX and BX registers contain  $1100_{16}$  and  $0ABC_{16}$ , respectively. What is the result of executing the instruction `ADD AX, BX`?

### ▶ Solution:

$$(BX) + (AX) = 0ABC_{16} + 1100_{16} = 1BBC_{16}$$

The sum ends up in destination register AX.

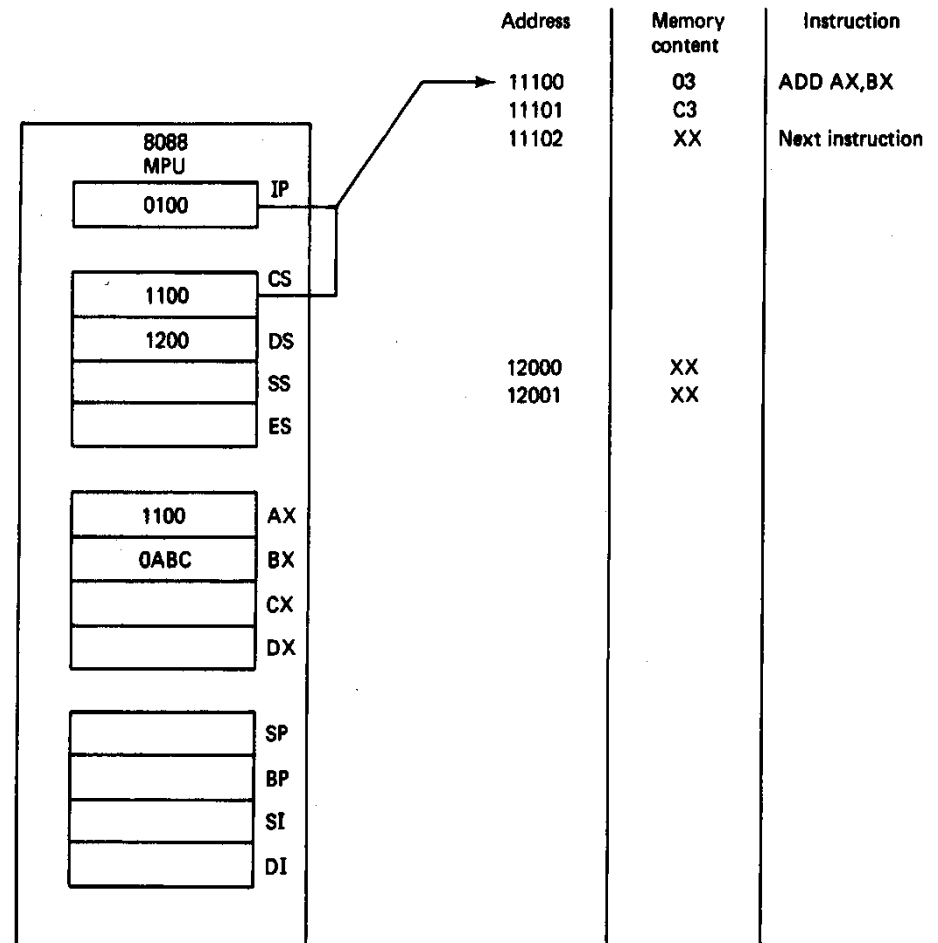
That is

$$(AX) = 1BBC_{16}$$
$$CF = 0$$

# 5.2 Arithmetic Instructions – Addition

## Instructions

- State before fetch and execution
- CS:IP = 1100:0100 = 11100H  
 ADD machine code = 03C3H  
 (AX) = 1100H  
 (BX) = 0ABCH  
 (DS) = 1200H  
 (1200:0000) = 12000H = XXXX



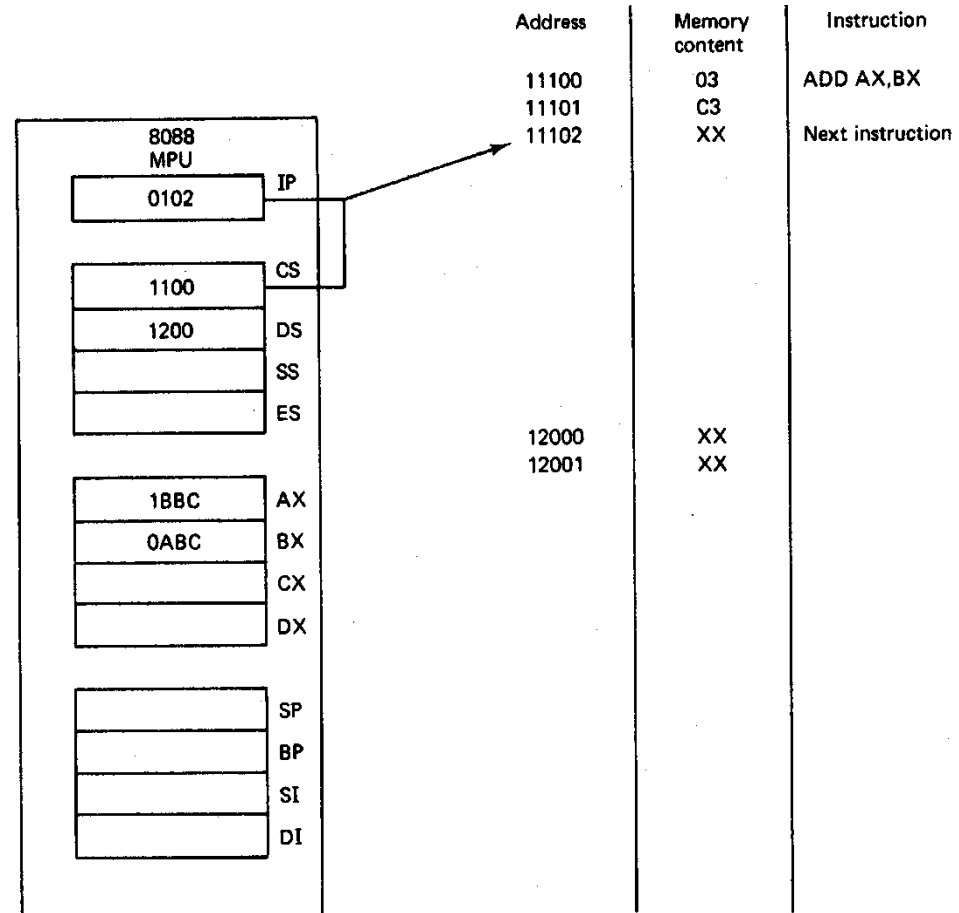
(a)

Before execution

# 5.2 Arithmetic Instructions – Addition

## Instructions

- State after execution
- CS:IP = 1100:0102 = 11102H  
11102H → points to next sequential instruction
- Operation performed
- (AX) + (BX) → (AX)  
(1100H) + (0ABCH) → 1BBCH  
(AX) = 1BBCH  
= 0001101110111100<sub>2</sub>  
(BX) = unchanged
  - Impact on flags
  - CF = 0 (no carry resulted)
  - ZF = 0 (not zero)
  - SF = 0 (positive)
  - PF = 0 (odd parity)—**parity flag** is only based on the bits of the **least significant byte**



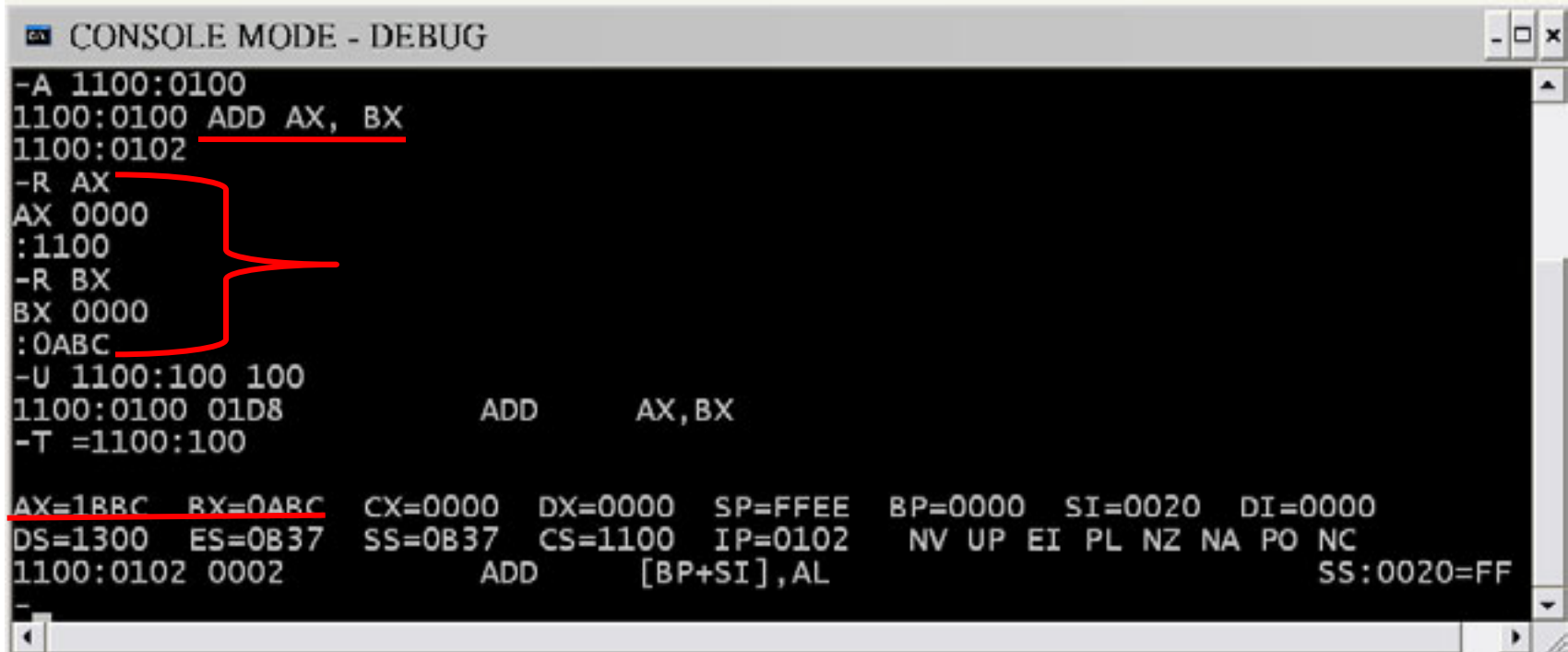
After execution

# 5.2 Arithmetic Instructions – Addition Instructions

## ▶ EXAMPLE

Verify the previous example using DEBUG program.

Solution:



```
CONSOLE MODE - DEBUG
-A 1100:0100
1100:0100 ADD AX, BX
1100:0102
-R AX
AX 0000
:1100
-R BX
BX 0000
:0ABC
-U 1100:100 100
1100:0100 01D8          ADD      AX,BX
-T =1100:100

AX=1BBC  BX=0ABC  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0020  DI=0000
DS=1300  ES=0B37  SS=0B37  CS=1100  IP=0102  NV UP EI PL NZ NA PO NC
1100:0102 0002          ADD      [BP+SI],AL      SS:0020=FF
```

# 5.2 Arithmetic Instructions – Addition Instructions

## ▶ EXAMPLE

The original contents of AX, BL, word-size memory location SUM, and carry flag (CF) are  $1234_{16}$ ,  $AB_{16}$ ,  $00CD_{16}$ , and  $0_{16}$ , respectively. Describe the results of executing the following sequence of instruction?

ADD AX, [SUM]

ADC BL, 05H

INC WORD PTR [SUM]

## ▶ Solution:

$$(AX) \leftarrow (AX) + (SUM) = 1234_{16} + 00CD_{16} = 1301_{16}$$

$$(BL) \leftarrow (BL) + \text{imm8} + (CF) = AB_{16} + 5_{16} + 0_{16} = B0_{16}$$

$$(SUM) \leftarrow (SUM) + 1_{16} = 00CD_{16} + 1_{16} = 00CE_{16}$$



# 5.2 Arithmetic Instructions – Addition

## Instructions

### ▶ EXAMPLE

What is the result of executing the following instruction sequence?

```
ADD AL, BL  
AAA
```

Assuming that AL contains  $32_{16}$  (ASCII code for 2) and BL contains  $34_{16}$  (ASCII code 4), and that AH has been cleared.

### ▶ Solution:

$$(AL) \leftarrow (AL) + (BL) = 32_{16} + 34_{16} = 66_{16}$$

The result after the AAA instruction is

$$(AL) = 06_{16}$$

$$(AH) = 00_{16}$$

with both AF and CF remain cleared

**Important:** Any adjustment operation will be performed on AL therefore the result must be always placed in AL before executing the adjustment operation

# ASCII Adjustment after ADD

## AAA Examples

Code	Registers
MOV AL, '2'	AL = 32H
MOV BL, '3'	BL = 33H
ADD AL, BL	AL = 65H
AAA	AL = 05

If low nibble of **AL**  $\leq 9$

- Clear the high nibble of AL
- AF = 0
- CF = 0

Code	Registers
MOV AL, '5'	AL = 35H
MOV BL, '6'	BL = 36H
ADD AL, BL	AL = 6BH
AAA	AX = 01H    AL=01H

If low nibble of **AL**  $> 9$  or **AF** = 1

**Clear the high nibble of AL**

- AL = AL + 6
- AH = AH + 1
- AF = 1
- CF = 1

Code	Registers
MOV AL, '9'	AL = 39H
MOV BL, '9'	BL = 39H
ADD AL, BL	AL = 72H
AAA	AX = 01H    AL=08H

If low nibble of **AL**  $> 9$  or **AF** = 1

**Clear the high nibble of AL**

- AL = AL + 6
- AH = AH + 1
- AF = 1
- CF = 1

# 5.2 Arithmetic Instructions – Addition

## Instructions

- ▶ EXAMPLE

Perform a 32-bit binary add operation on the contents of the processor's register.

- ▶ Solution:

$$(DX,CX) \leftarrow (DX,CX) + (BX,AX)$$

$$(DX,CX) = FEDCBA98_{16}$$

$$(BX,AX) = 01234567_{16}$$

```
MOV DX, FEDCH
MOV CX, BA98H
MOV BX, 0123H
MOV AX, 4567H
ADD CX, AX
ADC DX, BX           ; Add with carry
```

# 5.2 Arithmetic Instructions - Subtraction Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
SUB	Subtract	SUB D,S	$(D) - (S) \rightarrow (D)$ Borrow $\rightarrow$ (CF)	OF, SF, ZF, AF, PF, CF
SBB	Subtract with borrow	SBB D,S	$(D) - (S) - (CF) \rightarrow (D)$	OF, SF, ZF, AF, PF, CF
DEC	Decrement by 1	DEC D	$(D) - 1 \rightarrow (D)$	OF, SF, ZF, AF, PF
NEG	Negate	NEG D	$0 - (D) \rightarrow (D)$ $1 \rightarrow$ (CF)	OF, SF, ZF, AF, PF, CF
DAS	Decimal adjust for subtraction	DAS		SF, ZF, AF, PF, CF OF undefined
AAS	ASCII adjust for subtraction	AAS		AF, CF OF, SF, ZF, PF undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Accumulator	Immediate
Register	Immediate
Memory	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

Destination
Register
Memory

(d)

(a) Subtraction Instructions. (b) Allowed operands for SUB and SBB (c) Allowed operands for DEC (d) Allowed operands for NEG

▶ Variety of arithmetic instruction provided to support integer subtraction—core instructions are

- SUB  $\rightarrow$  Subtract
- SBB  $\rightarrow$  Subtract with borrow
- DEC  $\rightarrow$  Decrement
- NEG  $\rightarrow$  Negative

# 5.2 Arithmetic Instructions - Subtraction Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
SUB	Subtract	SUB D,S	$(D) - (S) \rightarrow (D)$ Borrow $\rightarrow$ (CF)	OF, SF, ZF, AF, PF, CF
SBB	Subtract with borrow	SBB D,S	$(D) - (S) - (CF) \rightarrow (D)$	OF, SF, ZF, AF, PF, CF
DEC	Decrement by 1	DEC D	$(D) - 1 \rightarrow (D)$	OF, SF, ZF, AF, PF
NEG	Negate	NEG D	$0 - (D) \rightarrow (D)$ $1 \rightarrow (CF)$	OF, SF, ZF, AF, PF, CF
DAS	Decimal adjust for subtraction	DAS		SF, ZF, AF, PF, CF OF undefined
AAS	ASCII adjust for subtraction	AAS		AF, CF OF, SF, ZF, PF undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Accumulator	Immediate
Register	Immediate
Memory	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

Destination
Register
Memory

(d)

(a) Subtraction Instructions. (b) Allowed operands for SUB and SBB (c) Allowed operands for DEC (d) Allowed operands for NEG

## ▶ Subtract Instruction—SUB

- SUB format and operation:

SUB D,S

$(D) - (S) \rightarrow (D)$

- Subtract values in two registers

SUB AX,BX

$(AX) - (BX) \rightarrow (AX)$

- Subtract a value in memory and a value in a register

SUB [DI],AX

$(DS:DI) - (AX) \rightarrow (DS:DI)$

- Subtract an immediate operand from a value in a register or memory

SUB AX,100H

$(AX) - IMM16 \rightarrow (AX)$

## ▶ Flags updated based on result

- CF, OF, SF, ZF, AF, PF

# 5.2 Arithmetic Instructions - Subtraction Instructions

```
CONSOLE MODE - DEBUG
-R BX
BX 0000
:1234
-R CX
CX 0000
:0123
-R F
NV UP EI PL NZ NA PO NC -
-A
0B37:0100 SBB BX,CX
0B37:0102
-R
AX=0000 BX=1234 CX=0123 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0100 NV UP EI PL NZ NA PO NC
0B37:0100 19CB SBB BX,CX
-T
AX=0000 BX=1111 CX=0123 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0102 NV UP EI PL NZ NA PE NC
0B37:0102 01ACEB78 ADD [SI+78EB],BP DS:78EB=0000
```

## ▶ Subtract with borrow instruction—**SBB**

- SBB format and operation:  
**SBB D,S**  
 $(D) - (S) - (CF) \rightarrow (D)$
- Used for extended subtractions
- Subtracts two registers and carry (borrow)

**SBB AX,BX**

- Example:

**SBB BX,CX**

**(BX) = 1234H**

**(CX) = 0123H**

**(CF) = 0**

**(BX) - (CX) - (CF)  $\rightarrow$  (BX)**

**1234H - 0123H - 0H =**

**1111H**

**(BX) = 1111H**

- **What about CF? CF=0**

If we execute instead:

**SBB CX, BX**

The result will be:

**CX= EEEF**

**CF= 1**

**SF=1**

**PF=0**



# 5.2 Arithmetic Instructions -

## Subtraction Instructions

- ▶ Negate instruction—**NEG** (2's complement)

- NEG format and operation

NEG D

$(0) - (D) \rightarrow (D)$

$(1) \rightarrow (CF)$

- Example:

NEG BX

$(BX) = 003AH$

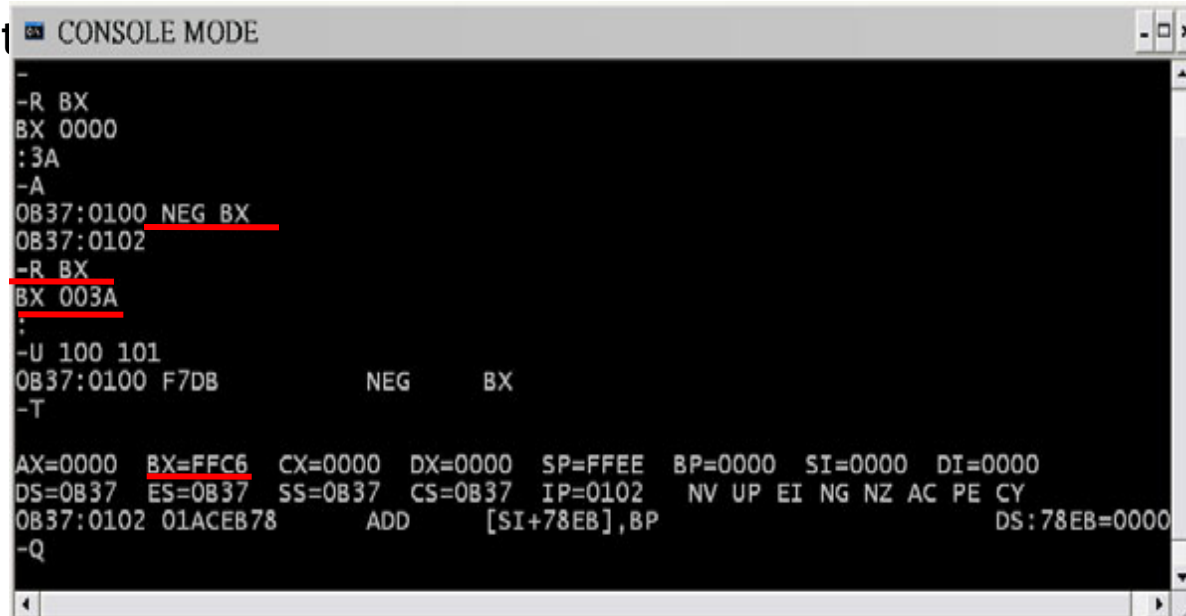
$(0) - (BX) \rightarrow (BX)$

$0000H - 003AH =$

$0000H + FFC6H$  (2's complement) =  $FFC6H$

$(BX) = FFC6H$  ;  $CF = 1$

- ▶ Since no carry is generated in this add operation, the carry flag is complemented to give  $CF = 1$ .



```
CONSOLE MODE
-R BX
BX 0000
:3A
-A
0B37:0100 NEG BX
0B37:0102
-R BX
BX 003A
-U 100 101
0B37:0100 F7DB      NEG      BX
-T
AX=0000 BX=FFC6  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B37 ES=0B37  SS=0B37  CS=0B37  IP=0102  NV UP EI NG NZ AC PE CY
0B37:0102 01ACEB78  ADD     [SI+78EB],BP  DS:78EB=0000
-Q
```

# 5.2 Arithmetic Instructions - Subtraction Instructions

```
C:\DOS>DEBUG
-R BX
BX 0000
:3A
-A
1342:0100 NEG BX
1342:0102
-R BX
BX 003A
:
-U 100 101
1342:0100 F7DB          NEG     BX
-T
AX=0000  BX=FFC6  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1342  ES=1342  SS=1342  CS=1342  IP=0102  NV UP EI NG NZ AC PE CY
1342:0102 B98AFF          MOV     CX,FF8A
-Q
C:\DOS>
```

- ▶ Decrement instruction—**DEC**
  - DEC format and operation  
**DEC D**  
 $(D) - 1 \rightarrow (D)$
  - Used to decrement pointer—addresses
    - Example  
**DEC SI**  
 $(SI) = 0FFFH$   
 $(SI) - 1 \rightarrow SI$   
 $0FFFH - 1 = 0FFEH$   
 $(SI) = 0FFEH$



# ASCII Adjustment after SUB: AAS Examples

Code	Registers
MOV AL, '3'	AL = 33H
MOV BL, '2'	BL = 32H
SUB AL, BL	AL = 01H
AAS	AL = 01

If low nibble of **AL**  $\leq 9$

- Clear the high nibble of AL (no need)
- AF = 0
- CF = 0

Code	Registers
MOV AL, '1'	AL = 31H
MOV BL, '9'	BL = 39H
SUB AL, BL	AH = 0      AL = F8H
AAS	AH = FFH    AL = 02H Answer = $-(10) + 2 = -8$

if low nibble of **AL**  $> 9$  or **AF** = 1

**Clear the high nibble of AL**

- AL = AL - 6
- AH = AH - 1
- AF = 1
- CF = 1

Code	Registers
MOV AL, '2'	AL = 32H
MOV BL, '3'	BL = 33H
SUB AL, BL	AH = 0      AL = FFH
AAS	AH = FFH    AL = 09H Answer = $-(10) + 9 = -1$

if low nibble of **AL**  $> 9$  or **AF** = 1

**Clear the high nibble of AL**

- AL = AL - 6
- AH = AH - 1
- AF = 1
- CF = 1

# 5.2 Arithmetic Instructions - Subtraction Instructions

- ▶ EXAMPLE

Perform a 32-bit binary subtraction for variable X and Y.

- ▶ Solution:

```
MOV SI, 200H ; Initialize pointer for X
MOV DI, 100H ; Initialize pointer for Y
MOV AX, [SI] ; Subtract LS words
SUB AX, [DI]
MOV [SI],AX ; Save the LS word of result
MOV AX, [SI]+2 ; Subtract MS words
SBB AX, [DI]+2
MOV [SI]+2, AX ; Save the MS word of result
```

# 5.2 Arithmetic Instructions -

## Multiplication Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
MUL	Multiply (unsigned)	MUL S	$(AL) \cdot (S8) \rightarrow (AX)$ $(AX) \cdot (S16) \rightarrow (DX), (AX)$	OF, CF SF, ZF, AF, PF undefined
DIV	Division (unsigned)	DIV S	(1) $Q((AX)/(S8)) \rightarrow (AL)$ $R((AX)/(S8)) \rightarrow (AH)$  (2) $Q((DX,AX)/(S16)) \rightarrow (AX)$ $R((DX,AX)/(S16)) \rightarrow (DX)$ If Q is $FF_{16}$ in case (1) or $FFFF_{16}$ in case (2), then type 0 interrupt occurs	OF, SF, ZF, AF, PF, CF undefined
IMUL	Integer multiply (signed)	IMUL S	$(AL) \cdot (S8) \rightarrow (AX)$ $(AX) \cdot (S16) \rightarrow (DX), (AX)$	OF, CF SF, ZF, AF, PF undefined
IDIV	Integer divide (signed)	IDIV S	(1) $Q((AX)/(S8)) \rightarrow (AL)$ $R((AX)/(S8)) \rightarrow (AH)$  (2) $Q((DX,AX)/(S16)) \rightarrow (AX)$ $R((DX,AX)/(S16)) \rightarrow (DX)$ If Q is positive and exceeds $7FFF_{16}$ or if Q is negative and becomes less than $8001_{16}$ , then type 0 interrupt occurs	OF, SF, ZF, AF, PF, CF undefined
AAM	Adjust AL for multiplication	AAM	$Q((AL)/10) \rightarrow (AH)$ $R((AL)/10) \rightarrow (AL)$	SF, ZF, PF OF, AF, CF undefined
AAD	Adjust AX for division	AAD	$(AH) \cdot 10 + (AL) \rightarrow (AL)$ $00 \rightarrow (AH)$	SF, ZF, PF OF, AF, CF undefined
CBW	Convert byte to word	CBW	(MSB of AL) $\rightarrow$ (All bits of AH)	None
CWD	Convert word to double word	CWD	(MSB of AX) $\rightarrow$ (All bits of DX)	None

(a)

Source
Reg8
Reg16
Mem8
Mem16

(b)

(a) Multiplication and Division Instructions. (b) Allowed operands

- ▶ Integer multiply instructions—**MUL** and **IMUL**
  - Multiply two unsigned or signed byte or word operands
- ▶ General format and operation
  - **MUL S** = Unsigned integer multiply
  - **IMUL S** = Signed integer multiply

$(AL) \times (S8) \rightarrow (AX)$

product gives 16 bit result

$(AX) \times (S16) \rightarrow (DX), (AX)$

16-bit product gives 32 bit result
- Source operand (S) can be an 8-bit or 16-bit value in a register or memory
- **AX** assumed to be destination for 16 bit result
- **DX, AX** assumed destination for 32 bit result
- Only CF and OF flags updated; other undefined

# 5.2 Arithmetic Instructions - Multiplication Instructions

## ▶ EXAMPLE

The 2's-complement signed data contents of AL are -1 and that of CL are -2. What result is produced in AX by executing the following instruction?

MUL CL

and

IMUL CL

## ▶ Solution:

$$(AL) = -1 \text{ (as 2's complement)} = 11111111_2 = FF_{16}$$

$$(CL) = -2 \text{ (as 2's complement)} = 11111110_2 = FE_{16}$$

Executing the MUL instruction gives

$$(AX) =$$

$$11111111_2 \times 11111110_2 = 1111110100000010_2 = FD02_{16}$$

Executing the IMUL instruction gives

$$(AX) = -1_{16} \times -2_{16} = 2_{16} = 0002_{16}$$

If the operation is MUL CX → multiply CX by AX and store the higher order word of the result in DX and the low order word of the result in AX

# 5.2 Arithmetic Instructions - Multiplication Instructions

## ▶ In general:

1 – The multiplication may take one of two forms

- Multiply AL by 8-bit operand → result will be 16-bit saved in AX.
- Multiply AX by 16-bit operand → result will be 32 bit saved in DX,AX.

2 – To perform unsigned multiplication convert the two numbers into binary and perform the multiplication.

3 – To perform signed multiplication

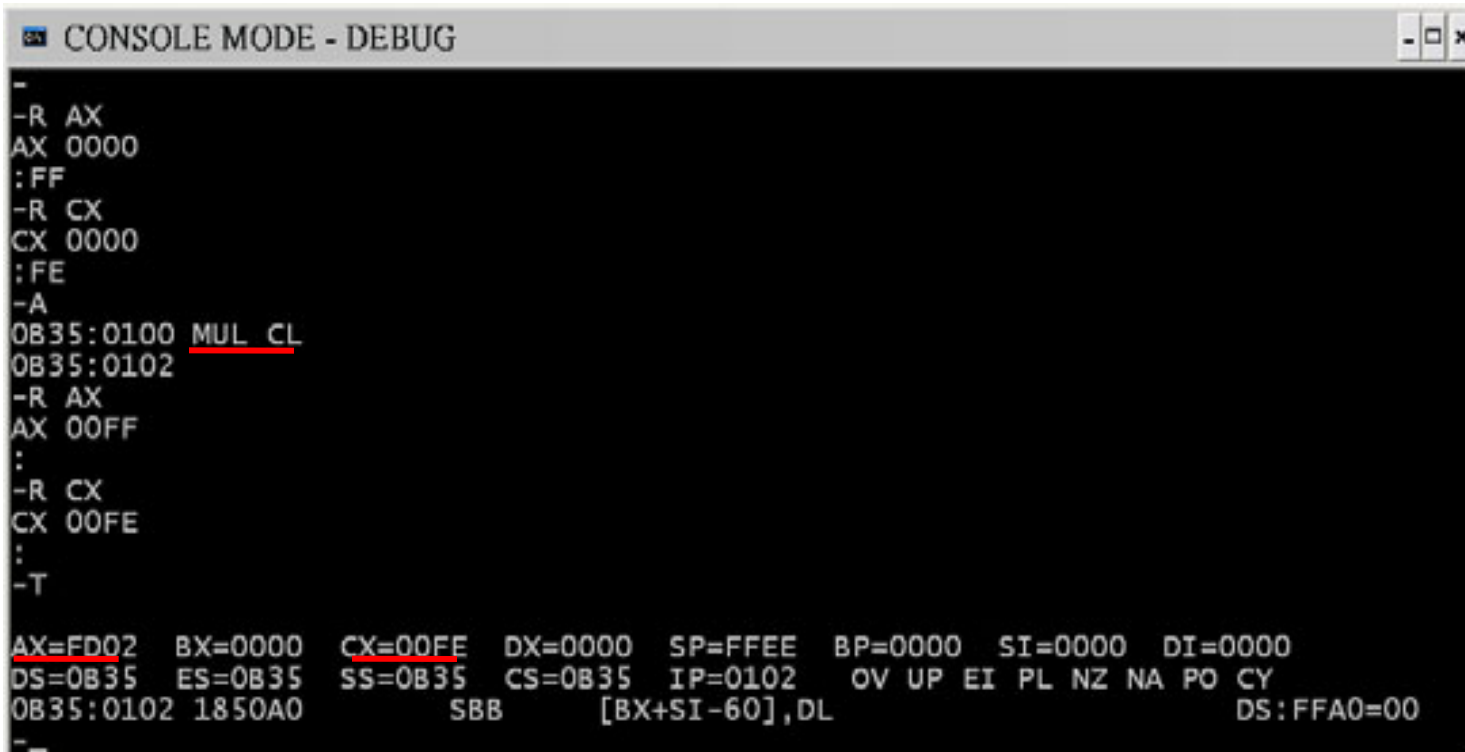
- If both operands are positive or both are negative → ignore the sign and multiply the numbers normally
- If one operand is positive and the other is negative → multiply the numbers and perform 2's complement for the result

# 5.2 Arithmetic Instructions - Multiplication Instructions

## ▶ EXAMPLE

Verify the previous example using DEBUG program.

Solution:



```
CONSOLE MODE - DEBUG
-
-R AX
AX 0000
:FF
-R CX
CX 0000
:FE
-A
0B35:0100 MUL CL
0B35:0102
-R AX
AX 00FF
:
-R CX
CX 00FE
:
-T
AX=FD02  BX=0000  CX=00FE  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0102  OV UP EI PL NZ NA PO CY
0B35:0102 1850A0          SBB      [BX+SI-60],DL          DS:FFA0=00
-
```

# 5.2 Arithmetic Instructions – Division

## Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
MUL	Multiply (unsigned)	MUL S	$(AL) \cdot (S8) \rightarrow (AX)$ $(AX) \cdot (S16) \rightarrow (DX), (AX)$	OF, CF SF, ZF, AF, PF undefined
DIV	Division (unsigned)	DIV S	(1) $Q((AX)/(S8)) \rightarrow (AL)$ $R((AX)/(S8)) \rightarrow (AH)$  (2) $Q((DX,AX)/(S16)) \rightarrow (AX)$ $R((DX,AX)/(S16)) \rightarrow (DX)$ If Q is $FF_{16}$ in case (1) or $FFFF_{16}$ in case (2), then type 0 interrupt occurs	OF, SF, ZF, AF, PF, CF undefined
IMUL	Integer multiply (signed)	IMUL S	$(AL) \cdot (S8) \rightarrow (AX)$ $(AX) \cdot (S16) \rightarrow (DX), (AX)$	OF, CF SF, ZF, AF, PF undefined
IDIV	Integer divide (signed)	IDIV S	(1) $Q((AX)/(S8)) \rightarrow (AL)$ $R((AX)/(S8)) \rightarrow (AH)$  (2) $Q((DX,AX)/(S16)) \rightarrow (AX)$ $R((DX,AX)/(S16)) \rightarrow (DX)$ If Q is positive and exceeds $7FFF_{16}$ or if Q is negative and becomes less than $8001_{16}$ , then type 0 interrupt occurs	OF, SF, ZF, AF, PF, CF undefined
AAM	Adjust AL for multiplication	AAM	$Q((AL)/10) \rightarrow (AH)$ $R((AL)/10) \rightarrow (AL)$	SF, ZF, PF OF, AF, CF undefined
AAD	Adjust AX for division	AAD	$(AH) \cdot 10 + (AL) \rightarrow (AL)$ $00 \rightarrow (AH)$	SF, ZF, PF OF, AF, CF undefined
CBW	Convert byte to word	CBW	(MSB of AL) $\rightarrow$ (All bits of AH)	None
CWD	Convert word to double word	CWD	(MSB of AX) $\rightarrow$ (All bits of DX)	None

(a)

Source
Reg8
Reg16
Mem8
Mem16

(b)

(a) Multiplication and Division Instructions. (b) Allowed operands

### Integer divide instructions—DIV and IDIV

- Divide unsigned— DIV S
- Operations:

$$(AX) / (S8) \rightarrow (AL) = \text{quotient}$$

$$(AH) = \text{remainder}$$

- 16 bit dividend in AX divided by 8-bit divisor in a register or memory,
- Quotient of result produced in AL
- Remainder of result produced in AH

$$(DX,AX) / (S16) \rightarrow (AX) = \text{quotient}$$

$$(DX) = \text{remainder}$$

- 32 bit dividend in DX,AX divided by 16-bit divisor in a register or memory
- Quotient of result produced in AX
- Remainder of result produced in DX
- Divide error** (Type 0) interrupt may occur.

# Division Examples

UNSIGNED DIV	Registers (in Hex)
MOV AX, 14	AH = 00 , AL = 0E
MOV BL, 3	BL = 03
DIV BL	AH = 02 , AL = 04
MOV AX, 14	AH = 00 , AL = 0E
MOV BL, -3	BL = FD
DIV BL	Operation = 14/253 AH = 0E, AL = 00
MOV AX, -14	AH = FF , AL = F2
MOV BL, 3	BL = 03
DIV BL	Operation = 65522/3 ERROR : INT0 (Divide overflow)
MOV AX, -14	AH = FF , AL = F2
MOV BL, -3	BL = FD
DIV BL	ERROR : INT0 (Divide overflow)

SIGNED DIV	Registers (in Hex)
MOV AX, 14	AH = 00 , AL = 0E
MOV BL, 3	BL = 03
IDIV BL	AH = 02 , AL = 04
MOV AX, 14	AH = 00 , AL = 0E
MOV BL, -3	BL = FD
IDIV BL	AH = 02 , AL = FC
MOV AX, -14	AH = FF , AL = F2
MOV BL, 3	BL = 03
IDIV BL	AH = FE , AL = FC
MOV AX, -14	AH = FF , AL = F2
MOV BL, -3	BL = FD
IDIV BL	AH = FE , AL = 04

## SIGNED DIV:

The sign for the remainder == sign of the dividend



# 5.2 Arithmetic Instructions - Convert Instructions

- ▶ Used to sign extension signed numbers for division
- ▶ Operations
  - CBW = convert byte to word  
(MSB of AL) → (all bits of AH)
  - CWD = convert word to double word  
(MSB of AX) → (all bits of DX)
- ▶ Application:
  - To divide two signed 8-bit numbers, the value of the dividend must be sign extended in AX
    - Load into AL
    - Use CBW to sign extend to 16 bits

# 5.2 Arithmetic Instructions – Division

## Instructions

### ▶ In general:

1 – The division may take one of two forms

- Divide AX by 8-bit operand → The division is performed between AX/ 8-bit operand. AL will contain the quotient of the result and AH will contain the remainder of the result. IF quotient is FF then interrupt occurs.
- Divide DX,AX by 16-bit operand → The division is performed between DX,AX/ 16-bit operand. AX will contain the quotient of the result and DX will contain the remainder of the result. IF quotient is FFFF then interrupt occurs.

2 – The way in which you perform either a signed or unsigned division is similar to the mechanism used in the multiplication instruction

3 – The sign for the remainder is always similar to the sign of the dividend  
ex.  $-26 / 8 \rightarrow$  Quotient =  $-3$  and Remainder =  $-2$

# 5.2 Arithmetic Instructions - Convert Instructions

## ▶ EXAMPLE

What is the result of executing the following instructions?

```
MOV AL, 0A1H
CBW
CWD
```

## ▶ Solution:

$(AL) = A1_{16} = 10100001_2$

Executing the CBW instruction extends the MSB of AL

$(AH) = 11111111_2 = FF_{16}$  or

$(AX) = 1111111110100001_2$

Executing CWD instruction, we get

$(DX) = 1111111111111111_2 = FFFF_{16}$

That is,

$(AX) = FFA1_{16}$   $(DX) = FFFF_{16}$

```
C:\DOS>DEBUG A:EX520.EXE
-U 0 9
0D03:0000 1E          PUSH   DS
0D03:0001 B80000      MOV    AX,0000
0D03:0004 50          PUSH   AX
0D03:0005 B0A1      MOV    AL,A1
0D03:0007 98          CBW
0D03:0008 99          CWD
0D03:0009 CB          RETF
-G 5

AX=0000 BX=0000 CX=0000 DX=0000 SP=003C BP=0000 SI=0000 DI=0000
DS=0CF3 ES=0CF3 SS=0D04 CS=0D03 IP=0005  NV UP EI PL NZ NA PO NC
0D03:0005 B0A1      MOV    AL,A1
-T

AX=00A1 BX=0000 CX=0000 DX=0000 SP=003C BP=0000 SI=0000 DI=0000
DS=0CF3 ES=0CF3 SS=0D04 CS=0D03 IP=0007  NV UP EI PL NZ NA PO NC
0D03:0007 98          CBW
-T

AX=FFA1 BX=0000 CX=0000 DX=0000 SP=003C BP=0000 SI=0000 DI=0000
DS=0CF3 ES=0CF3 SS=0D04 CS=0D03 IP=0008  NV UP EI PL NZ NA PO NC
0D03:0008 99          CWD
-T

AX=FFA1 BX=0000 CX=0000 DX=FFFF SP=003C BP=0000 SI=0000 DI=0000
DS=0CF3 ES=0CF3 SS=0D04 CS=0D03 IP=0009  NV UP EI PL NZ NA PO NC
0D03:0009 CB          RETF
-G

Program terminated normally
-Q

C:\DOS>
```

(c)

# 5.3 Logic Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
AND	Logical AND	AND D,S	$(S) \cdot (D) \rightarrow (D)$	OF, SF, ZF, PF, CF AF undefined
OR	Logical Inclusive-OR	OR D,S	$(S) + (D) \rightarrow (D)$	OF, SF, ZF, PF, CF AF undefined
XOR	Logical Exclusive-OR	XOR D,S	$(S) \oplus (D) \rightarrow (D)$	OF, SF, ZF, PF, CF AF undefined
NOT	Logical NOT	NOT D	$(\bar{D}) \rightarrow (D)$	None

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Destination
Register
Memory

(c)

(a) Logic Instructions. (b) Allowed operands for AND, OR, and XOR (c) Allowed operands for NOT

▶ Variety of logic instructions provided to support logical computations

- AND → Logical AND
- OR → Logical inclusive-OR
- XOR → Logical exclusive-OR
- NOT → Logical NOT

▶ Logical AND Instruction—AND

- AND format and operation:

AND D,S  
 $(S) \text{ AND } (D) \rightarrow (D)$

- Logical AND of values in two registers

AND AX,BX  
 $(AX) \text{ AND } (BX) \rightarrow (AX)$

- Logical AND of a value in memory and a value in a register

AND [DI],AX  
 $(DS:DI) \text{ AND } (AX) \rightarrow (DS:DI)$

- Logical AND of an immediate operand with a value in a register or memory

AND AX,100H  
 $(AX) \text{ AND IMM16} \rightarrow (AX)$

- Flags updated based on result

- CF, OF, SF, ZF, PF
- AF undefined

## 5.3 Logic Instructions

### ▶ EXAMPLE

Describe the results of executing the following instructions?

```
MOV AL, 01010101B
AND AL, 00011111B
OR AL, 11000000B
XOR AL, 00001111B
NOT AL
```

### ▶ Solution:

$$(AL) = 01010101_2 \cdot 00011111_2 = 00010101_2 = 15_{16}$$

Executing the OR instruction, we get

$$(AL) = 00010101_2 + 11000000_2 = 11010101_2 = D5_{16}$$

Executing the XOR instruction, we get

$$(AL) = 11010101_2 \oplus 00001111_2 = 11011010_2 = DA_{16}$$

Executing the NOT instruction, we get

$$(AL) = (\text{NOT})11011010_2 = 00100101_2 = 25_{16}$$

# 5.3 Logic Instructions

## ▶ EXAMPLE

Verify the previous example using DEBUG program.

## ▶ Solution:

```
C:\DOS>DEBUG
-A
1342:0100 MOV AL,55
1342:0102 AND AL,1F
1342:0104 OR AL,C0
1342:0106 XOR AL,0F
1342:0108 NOT AL
1342:010A
-T
AX=0055 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1342 ES=1342 SS=1342 CS=1342 IP=0102 NV UP EI PL NZ NA PO NC
1342:0102 241F AND AL,1F
-T
AX=0015 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1342 ES=1342 SS=1342 CS=1342 IP=0104 NV UP EI PL NZ NA PO NC
1342:0104 0CC0 OR AL,C0
-T
AX=00D5 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1342 ES=1342 SS=1342 CS=1342 IP=0106 NV UP EI NG NZ NA PO NC
1342:0106 340F XOR AL,0F
-T
AX=00DA BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1342 ES=1342 SS=1342 CS=1342 IP=0108 NV UP EI NG NZ NA PO NC
1342:0108 F6D0 NOT AL
-T
AX=0025 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1342 ES=1342 SS=1342 CS=1342 IP=010A NV UP EI NG NZ NA PO NC
1342:010A 2B04 SUB AX,[SI] DS:0000=20CD
-Q
C:\DOS>
```

## 5.3 Logic Instructions – Mask Application

- ▶ Application– Masking bits with the logic instructions
  - **Mask**—to clear a bit or bits of a byte or word to **0**
    - **AND** operation can be used to perform the mask operation:
      - $1 \text{ AND } 0 \rightarrow 0$ ;  $0 \text{ AND } 0 \rightarrow 0$ 
        - bit or bits are masked by ANDing with 0
      - $1 \text{ AND } 1 \rightarrow 1$ ;  $0 \text{ AND } 1 \rightarrow 0$ 
        - ANDing a bit or bits with **1** results in **no change**
    - Example: Masking the upper 12 bits of a value in a register

**AND AX,000FH**

(AX) = FFFF

IMM16 AND (AX)  $\rightarrow$  (AX)

000FH AND FFFFH = 0000000000001111<sub>2</sub> AND 1111111111111111<sub>2</sub>  
= 0000000000001111<sub>2</sub>  
= 000FH

## 5.3 Logic Instructions – ~~Mask~~ <sup>Setting</sup> Application

- **OR** operation can be used to set a bit or bits of a byte or word to **1**
  - $X \text{ OR } 0 \rightarrow X$ ; result is **unchanged**
  - $X \text{ or } 1 \rightarrow 1$ ; result is always **1**
- Example: Setting a control flag in a byte memory location to 1

```
MOV AL,[CONTROL_FLAGS]
OR AL, 10H ; 00010000 sets fifth bit -b4
MOV [CONTROL_FLAGS],AL
```
- Executing the above instructions, we get
$$(AL) = \text{XXXXXXXX}_2 \text{ OR } 00010000_2 = \text{XXX1XXXX}_2$$



# 5.4 Shift Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
SAL/SHL	Shift arithmetic left/shift logical left	SAL/SHL D,Count	Shift the (D) left by the number of bit positions equal to Count and fill the vacated bits positions on the right with zeros	CF, PF, SF, ZF AF undefined OF undefined if count $\neq$ 1
SHR	Shift logical right	SHR D,Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions on the left with zeros	CF, PF, SF, ZF AF undefined OF undefined if count $\neq$ 1
SAR	Shift arithmetic right	SAR D,Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions on the left with the original most significant bit	SF, ZF, PF, CF AF undefined OF undefined if count $\neq$ 1

(a)

Destination	Count
Register	1
Register	CL
Memory	1
Memory	CL

(b)

(a) Shift Instructions. (b) Allowed operands

- ▶ Variety of shift instructions provided
  - SAL/SHL → Shift **arithmetic** left/shift **logical** left
  - SHR → Shift logical right
  - SAR → Shift arithmetic right
- ▶ Perform a variety of shift left and shift right operations on the bits of a destination data operand
- ▶ Basic shift instructions—**SAL/SHL, SHR, SAR**
  - Destination may be in either a register or a storage location in memory
  - Shift count may be:
    - 1** = one bit shift
    - CL** = 1 to 255 bit shift
- ▶ Flags updated based on result
  - CF, SF, ZF, PF
  - AF undefined
  - OF undefined if Count  $\neq$  1

**Every shift operation is equivalent to :**

**\* multiplication by 2 for Shift left**

**\* dividing by 2 for Logical Shift right**

# 5.4 Shift Instructions – Operation of the SAL/SHL Instruction

- ▶ Typical instruction—count of 1  
`SHL AX,1`

- ▶ Before execution  
Dest = (AX) = 1234H  
= 0001001000110100<sub>2</sub>

Count = 1

CF = X

- ▶ Operation:

- The value in all bits of AX are shifted left one bit position
- Emptied **LSB** is filled with **0**
- Value shifted out of MSB goes to carry flag

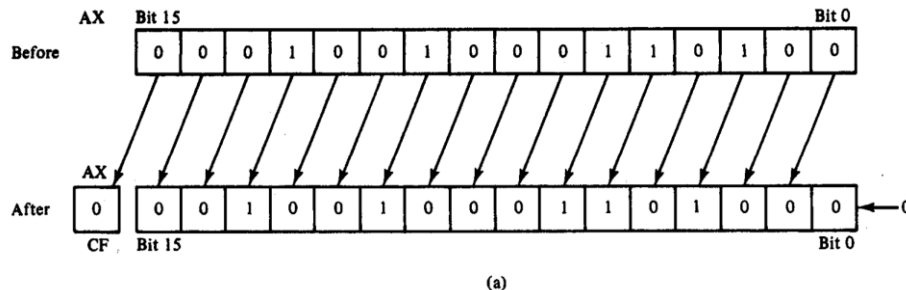
- ▶ After execution

Dest = (AX) = 2468H  
= 0010010001101000<sub>2</sub>

CF = 0

- ▶ Conclusion:

- **MSB** has been isolated in **CF** and can be acted upon by control flow instruction- conditional jump
- Result has been **multiplied by 2**



# 5.4 Shift Instructions – Operation of the SHR Instruction

- ▶ Typical instruction—count in CL

`SHR AX,CL`

- ▶ Before execution

Dest = (AX) = 1234H = 466010  
= 0001001000110100<sub>2</sub>

Count = (CL) = 02H

CF = X

- ▶ Operation:

- The value in all bits of AX are shifted right two bit positions
- Emptied MSBs are filled with 0s
- Value shifted out of LSB goes to carry flag

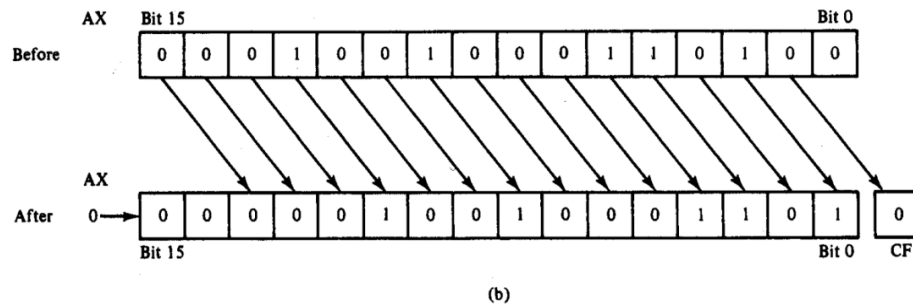
- ▶ After execution

Dest = (AX) = 048DH = 116510  
= 0000010010001101<sub>2</sub>

CF = 0

- ▶ Conclusion

- **Bit 1** has been isolated in CF and can be acted upon by control flow instruction- conditional jump
- Result has been **divided by 4**



# 5.4 Shift Instructions – Operation of the SAR Instruction

- ▶ Typical instruction—count in CL

SAR AX,CL

- ▶ Before execution—arithmetic implies signed numbers

Dest = (AX) = 091AH

= 0000100100011010<sub>2</sub> = +2330

Count = CL = 02H

CF = X

- ▶ Operation:

- The value in all bits of AX are shifted right two bit positions
- Emptied MSB is filled with the value of the sign bit
- Values shifted out of LSB go to carry flag

- ▶ After execution

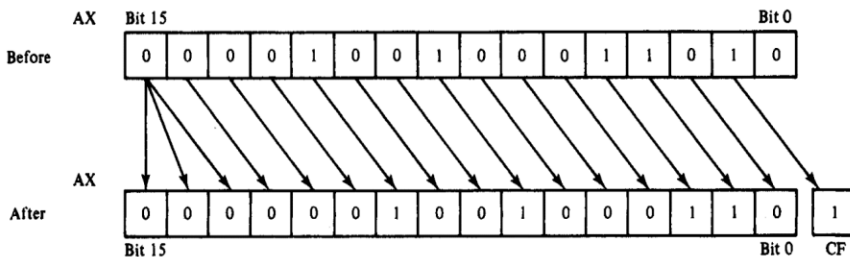
Dest = (AX) = 0246H

= 0000001001000110<sub>2</sub> = +582

CF = 1

- ▶ Conclusion

- **Bit 1** has been isolated in **CF** and can be acted upon by control flow instruction– conditional jump
- Result has been **signed extended**
- Result value has been divided by **4** and **rounded** to integer:  $4 \times +582 = +2328$



## 5.4 Shift Instructions – Operation of the SAR Instruction

### ▶ EXAMPLE

Assume that CL contains  $02_{16}$  and AX contains  $091A_{16}$ .

Determine the new contents of AX and the carry flag after the instruction SAR AX, CL is executed.

### ▶ Solution:

Initial (AX) =  $0000100100011010_2$

shift AX right twice: (AX) = 0000001001000110<sub>2</sub> =  $0246_{16}$

and the carry flag is (CF) =  $1_2$

# 5.4 Shift Instructions – Operation of the SAR Instruction

## ▶ EXAMPLE

Verify the previous example using DEBUG program.

## ▶ Solution:

```
CONSOLE MODE - DEBUG
-
-A
0B35:0104 SAR AX, CL
0B35:0106
-R AX
AX FD02
:091A
-R CX
CX 00FE
:2
-R F
OV UP EI PL NZ NA PO CY -
-T
AX=0246 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0104 NV UP EI PL NZ NA PO CY
0B35:0104 D3F8 SAR AX, CL
-
```

## 5.4 Shift Instructions – Operation of the SAR Instruction

### ▶ EXAMPLE

Isolate the bit B<sub>3</sub> of the byte at the offset address CONTROL\_FLAGS.

### ▶ Solution:

```
MOV AL, [CONTROL_FLAGS]
MOV CL, 04H
SHR AL, CL
```

Executing the instructions, we get

(AL)=0000B<sub>7</sub>B<sub>6</sub>B<sub>5</sub>B<sub>4</sub>

and

(CF)=B<sub>3</sub>

# 5.5 Rotate Instructions

- ▶ Variety of rotate instructions provided:
  - ROL → Rotate left
  - ROR → Rotate right
  - RCL → Rotate left through carry
  - RCR → Rotate right through carry
- ▶ Perform a variety of rotate left and rotate right operations on the bits of a destination data operand
  - Overview of function
  - Destination may be in either a register or a storage location in memory
  - Rotate count may be:
    - 1 = one bit rotate
    - CL = 1 to 255 bit rotate
  - Flags updated based on result
    - CF
    - OF undefined if Count ≠ 1
- ▶ Used to **rearrange** information

Mnemonic	Meaning	Format	Operation	Flags Affected
ROL	Rotate left	ROL D,Count	Rotate the (D) left by the number of bit positions equal to Count. Each bit shifted out from the leftmost bit goes back into the rightmost bit position.	CF OF undefined if count ≠ 1
ROR	Rotate right	ROR D,Count	Rotate the (D) right by the number of bit positions equal to Count. Each bit shifted out from the rightmost bit goes into the leftmost bit position.	CF OF undefined if count ≠ 1
RCL	Rotate left through carry	RCL D,Count	Same as ROL except carry is attached to (D) for rotation.	CF OF undefined if count ≠ 1
RCR	Rotate right through carry	RCR D,Count	Same as ROR except carry is attached to (D) for rotation.	CF OF undefined if count ≠ 1

(a)

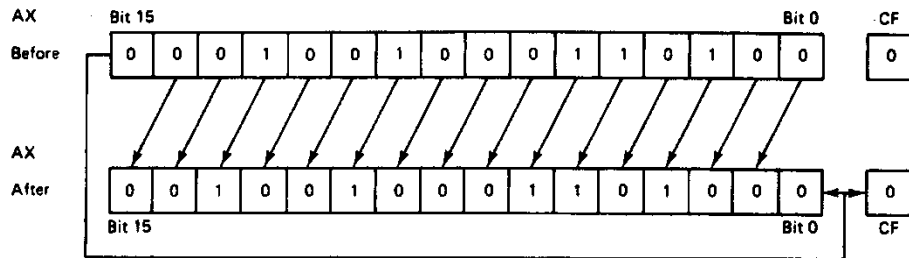
Destination	Count
Register	1
Register	CL
Memory	1
Memory	CL

(b)

(a) Rotate Instructions. (b) Allowed operands



# 5.5 Rotate Instructions – Operation of the ROL Instruction



- ▶ Typical instruction—count of 1  
`ROL AX,1`

- ▶ Before execution

Dest = (AX) = 1234H

= 0001 0010 0011 0100<sub>2</sub>

Count = 1

CF = 0

- ▶ Operation

- The value in all bits of AX are **shifted** left one bit position
- Value rotated out of the MSB is **reloaded** at LSB
- Value rotated out of MSB is **copied** to carry flag

- ▶ After execution

Dest = (AX) = 2468H

= 0010 0100 0110 1000<sub>2</sub>

CF = 0

# 5.5 Rotate Instructions – Operation of the ROR Instruction

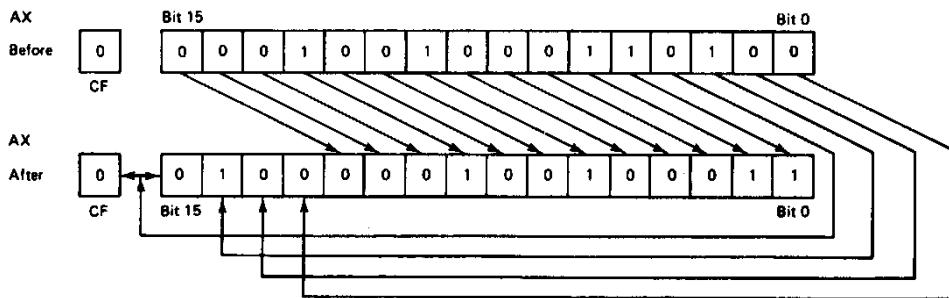
- ▶ Typical instruction—count in CL  
**ROR AX,CL**

- ▶ Before execution  
Dest = (AX) = 1234H  
= 0001001000110100<sub>2</sub>  
Count = 04H  
CF = 0

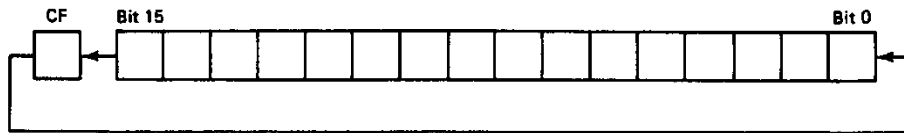
- ▶ Operation
  - The value in all bits of AX are rotated right four bit positions
  - Values rotated out of the LSB are reloaded at MSB
  - Values rotated out of MSB copied to carry flag

- ▶ After execution  
Dest = (AX) = 4123H  
= 0100000100100011<sub>2</sub>  
CF = 0

- ▶ **Conclusion:**
  - Note that the position of hex characters in AX have be rearranged



# 5.5 Rotate Instructions – Operation of the RCL Instruction



- ▶ • Typical instruction—count in CL  
**RCL BX,CL**
- ▶ Before execution
  - Dest = (BX) = 1234H  
= 0001001000110100<sub>2</sub>
  - Count = (CL) = 04H
  - CF = 0
- ▶ Operation
  - The value in all bits of AX are rotated left four bit positions
  - Emptied MSBs are rotated through the carry bit back into the LSB
  - First rotate loads prior value of CF at the LSB
  - Last value rotated out of MSB retained in carry flag
- ▶ After execution
  - Dest = (BX) = 2340H  
= 0010001101000000<sub>2</sub>
  - CF = 1

## 5.5 Rotate Instructions – Operation of the RCR Instruction

### ▶ EXAMPLE

What is the result in BX and CF after execution of the following instructions?

RCR BX, CL

Assume that, prior to execution of the instruction, (CL)=04<sub>16</sub>, (BX)=1234<sub>16</sub>, and (CF)=0

### ▶ Solution:

The original contents of BX are

$$(BX) = 0001001000110100_2 = 1234_{16}$$

Execution of the RCR command causes a 4-bit rotate right through carry to take place on the data in BX, the results are

$$(BX) = 1000000100100011_2 = 8123_{16}$$

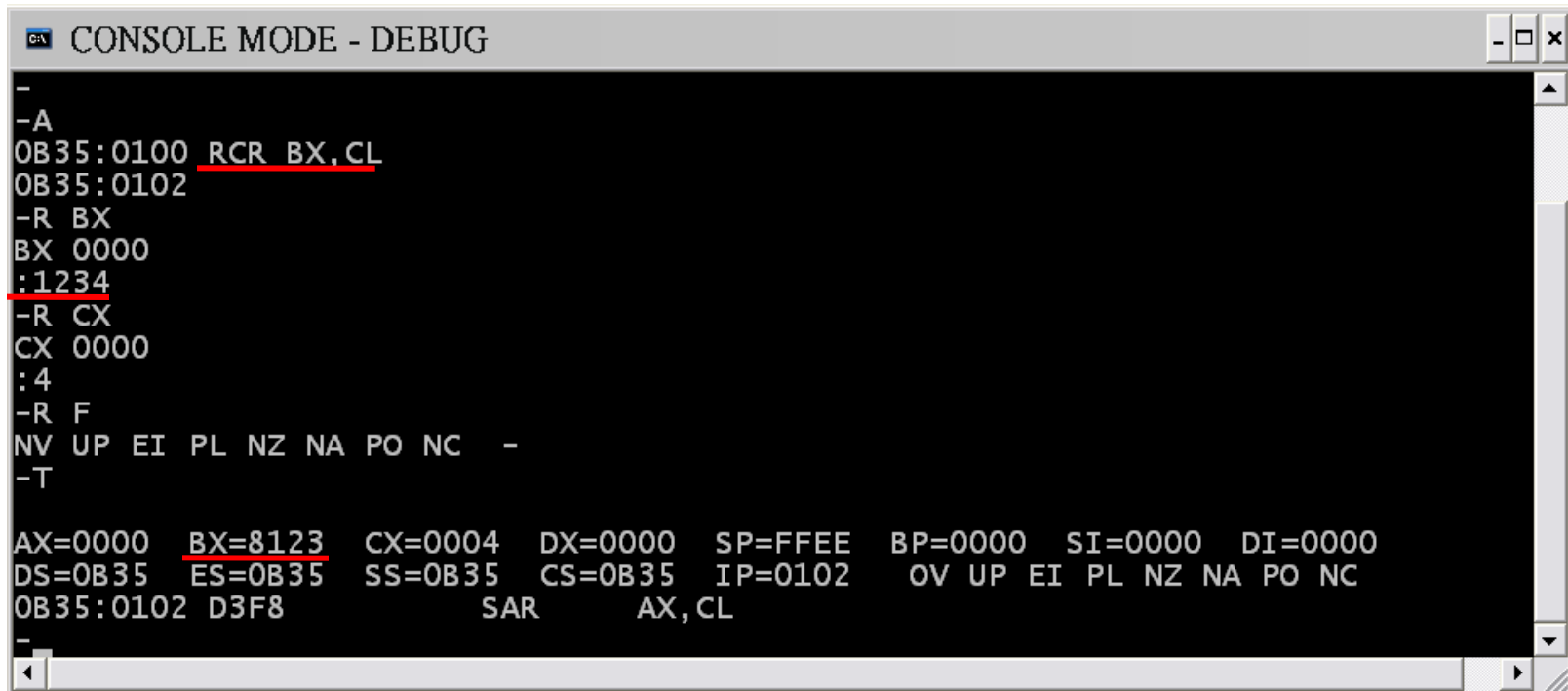
$$(CF) = 0_2$$

# 5.5 Rotate Instructions – Operation of the RCR Instruction

## ▶ EXAMPLE

Verify the previous example using DEBUG program.

## ▶ Solution:



```
CONSOLE MODE - DEBUG
-
-A
0B35:0100 RCR BX,CL
0B35:0102
-R BX
BX 0000
:1234
-R CX
CX 0000
:4
-R F
NV UP EI PL NZ NA PO NC -
-T
AX=0000 BX=8123 CX=0004 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0102 OV UP EI PL NZ NA PO NC
0B35:0102 D3F8 SAR AX,CL
-
```

## 5.5 Rotate Instructions

### ▶ EXAMPLE

Disassembly and addition of 2 hexadecimal digits stored as a byte in memory.

### ▶ Solution:

```
MOV AL,[HEX_DIGITS]
MOV BL,AL
MOV CL,04H
ROR BL,CL
AND AL,0FH
AND BL,0FH
ADD AL,BL
```

1st Instruction → Loads AL with byte containing two hex digits

2nd Instruction → Copies byte to BL

3rd Instruction → Loads rotate count

4th instruction → Aligns upper hex digit of BL with lower digit in AL

5th Instruction → Masks off upper hex digit in AL

6th Instruction → Masks off upper four bits of BL

7th Instruction → Adds two hex digits

# H.W. #5

- Solve the following problems from Chapter 5 from the course textbook:

1, 10, 26, 38, 47

# **CPE 408330**

## **Assembly Language and Microprocessors**

### **Chapter 6: 8088/8086 Microprocessor Programming – Control Flow Instructions and Program Structures**

[Computer Engineering Department,  
Hashemite University]



# Lecture Outline

- ▶ 6.1 Flag–Control Instructions
- ▶ 6.2 Compare Instructions
- ▶ 6.3 Control Flow and Jump Instructions
- ▶ 6.4 Subroutines and Subroutine–Handling Instructions
- ▶ 6.5 The Loop and the Loop–Handling Instructions
- ▶ 6.6 String and String–Handling Instructions

## *6.1 Flag-Control Instructions*

- ▶ The flag-control instructions, when executed, directly affect the state of the flags. These instructions include:
  - ❑ LAHF (Load AH from flags)
  - ❑ SAHF (Store AH into flags)
  - ❑ CLC (Clear carry)
  - ❑ STC (Set carry)
  - ❑ CMC (Complement carry)
  - ❑ CLI (Clear interrupt)
  - ❑ STI (Set interrupt)

# 6.1 Flag-Control Instructions - Loading, Storing, and Modifying Flags

Mnemonic	Meaning	Operation	Flags affected
LAHF	Load AH from flags	$(AH) \leftarrow (\text{Flags})$	None
SAHF	Store AH into flags	$(\text{Flags}) \leftarrow (AH)$	SF,ZF,AF,PF,CF
CLC	Clear carry flag	$(CF) \leftarrow 0$	CF
STC	Set carry flag	$(CF) \leftarrow 1$	CF
CMC	Complement carry flag	$(CF) \leftarrow (\overline{CF})$	CF
CLI	Clear interrupt flag	$(IF) \leftarrow 0$	IF
STI	Set interrupt flag	$(IF) \leftarrow 1$	IF

- Variety of flag control instructions provide support for loading, saving, and modifying content of the flags register

- LAHF/SAHF → Load/store control flags

- CLC/STC/CMC → Modify carry flag

- CLI/STI → Modify interrupt flag

- Modifying the carry flag—CLC/STC/CMC

- Used to initialize the carry flag

- Clear carry flag

CLC

$0 \rightarrow (CF)$

- Set carry flag

STC

$1 \rightarrow (CF)$

- Complement carry flag

CMC

$(CF^*) \rightarrow (CF)$  \* stands for over bar (NOT)

- Modifying the interrupt flag—CLI/STI

- Used to turn on/off external hardware interrupts

- Clear interrupt flag

CLI

$0 \rightarrow (IF)$  Disable interrupts

- Set interrupt flag

STI

$1 \rightarrow (IF)$  Enable interrupts

# 6.1 Flag-Control Instructions - Debug Example

```
C:\DOS>DEBUG
-A
1342:0100 CLC
1342:0101 STC
1342:0102 CMC
1342:0103
-R F
NV UP EI PL NZ NA PO NC -CY
-R F
NV UP EI PL NZ NA PO CY -
-T

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1342 ES=1342 SS=1342 CS=1342 IP=0101 NV UP EI PL NZ NA PO NC
1342:0101 F9          STC
-T

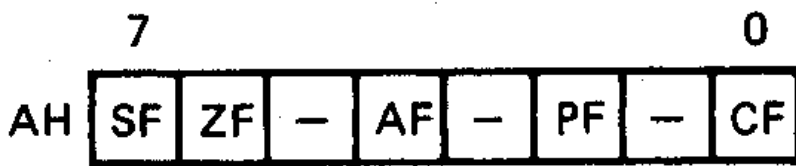
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1342 ES=1342 SS=1342 CS=1342 IP=0102 NV UP EI PL NZ NA PO CY
1342:0102 F5          CMC
-T

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1342 ES=1342 SS=1342 CS=1342 IP=0103 NV UP EI PL NZ NA PO NC
1342:0103 8AFF          MOV     BH,BH
-Q

C:\DOS>
```

- Debug flag notation
  - CF → CY = 1, NC = 0
  - Example—Execution of carry flag modification instructions
- CY=1 → initial state
- CLC ;Clear carry flag
- STC ;Set carry flag
- CMC ;Complement carry flag

# 6.1 Flag-Control Instructions - Loading and Storing the Flags Register



SF = Sign flag  
ZF = Zero flag  
AF = Auxiliary  
PF = Parity flag  
CF = Carry flag  
— = Undefined (do not use)

- Format of the flags in the AH register
- All loads and stores of flags take place through the AH register
  - B0 = CF
  - B2 = PF
  - B4 = AF
  - B6 = ZF
  - B7 = SF
- Load the AH register with the content of the flags registers
  - LAHF**  
(Flags) → (AH)  
Flags **unchanged**
- Store the content of AH into the flags register
  - SAHF**  
(AH) → (Flags)  
SF,ZF,AF,PF,CF → **updated**
- Application: saving a copy of the flags in memory and initializing with new values from memory

# 6.1 Flag-Control Instructions - Loading and Storing the Flags Register

## ▶ EXAMPLE

Write an instruction sequence to save the current contents of the 8088's flags in the memory location at offset MEM1 of the current data segment and then reload the flags with the contents of the storage location at offset MEM2.

## ▶ Solution:

```
LAHF                ; Load AH from flags
MOV [MEM1], AH      ; Move content of AH to MEM1
MOV AH, [MEM2]      ; Load AH from MEM2
SAHF                ; Store content of AH into flags
```

# 6.1 Flag-Control Instructions - Loading and Storing the Flags Register

## ▶ EXAMPLE

```
C:\> CONSOLE MODE - DEBUG
-
-A 0:0110
0000:0110 LAHF
0000:0111 MOV [0150],AH
0000:0115 MOV AH, [0151]
0000:0119 SAHF
0000:011A
-E 0:0150 FF 01
-R CS
CS 0B37
:0
-R IP
IP 0100
:0110
-R DS
DS 0B37
:0
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0000 ES=0B37 SS=0B37 CS=0000 IP=0110 NV UP EI PL NZ NA PO NC
0000:0110 9F LAHF
-
```

# 6.1 Flag-Control Instructions - Loading and Storing the Flags Register

## ▶ EXAMPLE

```
C:\> CONSOLE MODE - DEBUG

AX=0200  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0000  ES=0B37  SS=0B37  CS=0000  IP=0111  NV UP EI PL NZ NA PO NC
0000:0111 88265001      MOV      [0150],AH      DS:0150=FF
-T

AX=0200  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0000  ES=0B37  SS=0B37  CS=0000  IP=0115  NV UP EI PL NZ NA PO NC
0000:0115 8A265101      MOV      AH,[0151]      DS:0151=01
-D 150 151
0000:0150  02 01
-T

AX=0100  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0000  ES=0B37  SS=0B37  CS=0000  IP=0119  NV UP EI PL NZ NA PO NC
0000:0119 9E      SAHF
-T

AX=0100  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0000  ES=0B37  SS=0B37  CS=0000  IP=011A  NV UP EI PL NZ NA PO CY
0000:011A 00F0      ADD      AL,DH
```



# 6.1 Flag-Control Instructions - Loading and Storing the Flags Register

## ▶ EXAMPLE

Of the three carry flag instructions CLC, STC, and CMC, only one is really independent instruction. That is, the operation that it provides cannot be performed by a series of the other two instructions. Determine which one of the carry instruction is the independent instruction.

## ▶ Solution:

CLC  $\Leftrightarrow$  STC followed by a CMC

STC  $\Leftrightarrow$  CLC followed by a CMC

Therefore, only CMC is the independent instruction.

# 6.1 Flag-Control Instructions - Loading and Storing the Flags Register

## ▶ EXAMPLE

Verify the operation of the following instructions that affect the carry flag,

CLC

STC

CMC

by executing them with the DEBUG program. Start with CF flag set to 1 (CY).

# 6.1 Flag-Control Instructions - Loading and Storing the Flags Register

## ▶ Solution:

```
CONSOLE MODE - DEBUG
0B37:0100 CLC
0B37:0101 STC
0B37:0102 CMC
0B37:0103
-R F
NV UP EI PL NZ NA PO NC -CY
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0101 NV UP EI PL NZ NA PO NC
0B37:0101 F9 STC
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0102 NV UP EI PL NZ NA PO CY
0B37:0102 F5 CMC
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0103 NV UP EI PL NZ NA PO NC
0B37:0103 AC LODSB
-
```

# 6.2 Compare Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
CMP	Compare	CMP D,S	(D) – (S) is used in setting or resetting the flags	CF,AF,OF,PF,SF,ZF

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

- Compare instruction
  - Used to **compare** two values of data and **update** the state of the flags to reflect their relationship
- General format:

CMP D,S

- Operation: Compares the content of the source to the destination; updates flags based on result
- (D) – (S) → Flags updated to reflect relationship
- Source and destination contents **unchanged**
- Allowed operand variations:
  - Values in two registers
  - Values in a memory location and a register
  - Immediate source operand and a value in a register or memory
- Allows SW to perform **conditional** control flow—typically testing of a flag by **jump** instruction
  - ZF = 1 → D = S = Equal
  - ZF = 0, CF = 1 → D < S = Unequal, less than
  - ZF = 0, CF = 0 → D > S = Unequal, greater than

# 6.2 Compare Instruction

## ▶ EXAMPLE

Describe what happens to the status flags as the sequence of instructions that follows is executed.

```
MOV AX, 1234H  
MOV BX, ABCDH  
CMP AX, BX
```

## ▶ Solution:

$$(AX) = 1234_{16} = 0001001000110100_2$$

$$(BX) = ABCD_{16} = 10101011111001101_2$$

$$(AX) - (BX) = 0001001000110100_2 - 10101011111001101_2 \\ = 0110\ 0110\ 0110\ 0111_2$$

Therefore, ZF = 0, SF = 0, OF = 0, PF = 0, CF = 1, AF = 1

# 6.2 Compare Instruction

## ▶ EXAMPLE

```
CONSOLE MODE - DEBUG
-
-A
0B35:0100 MOV AX, 1234
0B35:0103 MOV BX, ABCD
0B35:0106 CMP AX, BX
0B35:0108
-T
AX=1234 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0103 NV UP EI PL NZ NA PO NC
0B35:0103 BBCDAB MOV BX,ABCD
-T
AX=1234 BX=ABCD CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0106 NV UP EI PL NZ NA PO NC
0B35:0106 39D8 CMP AX,BX
-T
AX=1234 BX=ABCD CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0108 NV UP EI PL NZ AC PO CY
0B35:0108 41 INC CX
-
```

# 6.3 Control Flow and Jump Instructions

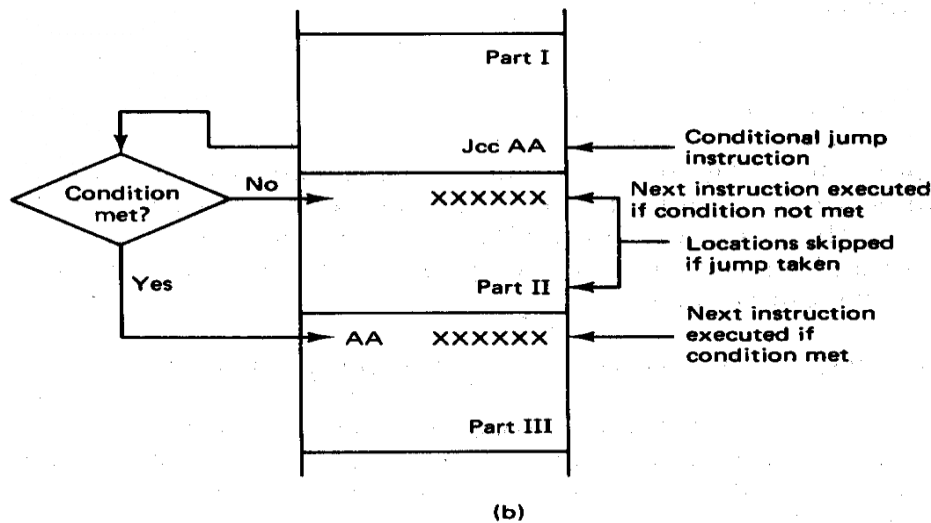
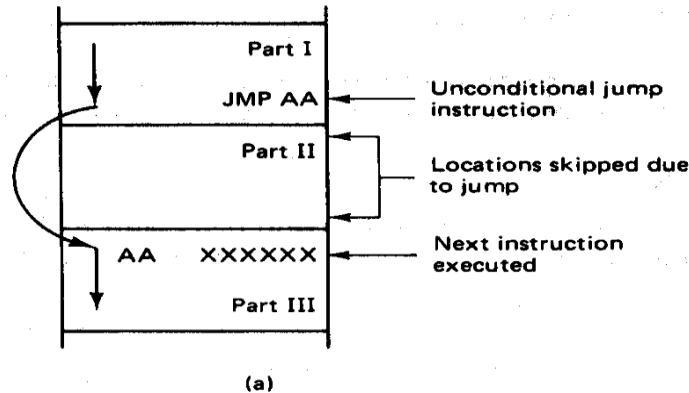
- Control flow: alternate the execution path of instructions in the program
  - What are the pointers that keep track of the instructions being executed ?
  - How it is possible to alternate the sequence of instructions being executed ?
- Unconditional jump instruction
  - Conditional jump instruction
  - Branching structure - IF-THEN
  - Loop program structure - REPEAT- UNTIL and WHILE-DO
  - Applications using the loop and branch software structures

# Type of Jumps: How IP and CS are modified with Jumps

- ▶ **Intra-segment jump**: **modify IP**
  - **Short-label** specify 8-bit signed displacement (relative to the jump instruction)
  - **Near-label** specify IP with 16-bit immediate operand
  - **Memptr-16** and **Regptr-16** are same as Near-label but IP is specified as content of memory or register.
- ▶ **Inter-segment jump**: **modify IP and CS**
  - **Far-label**: uses 32-bit immediate operand to specify IP and CS
  - **Memptr-32**: uses 4 byte memory locations to specify IP and CS.

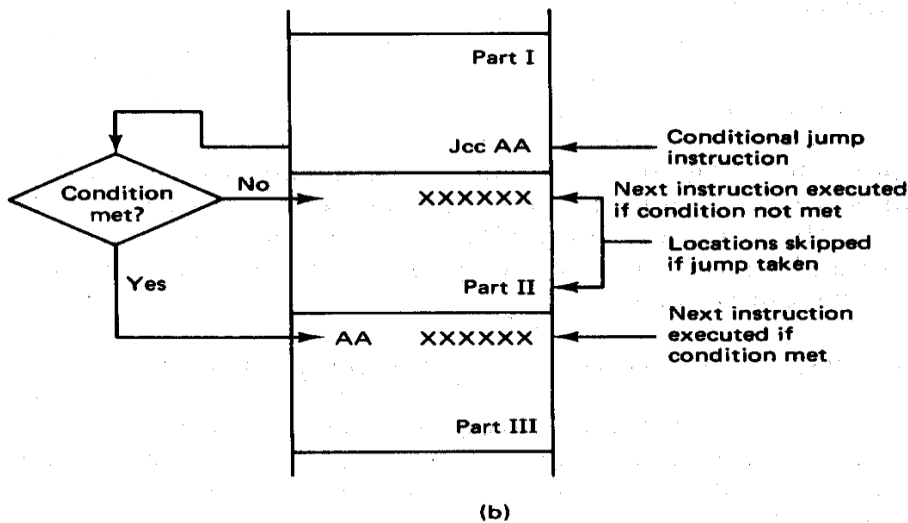
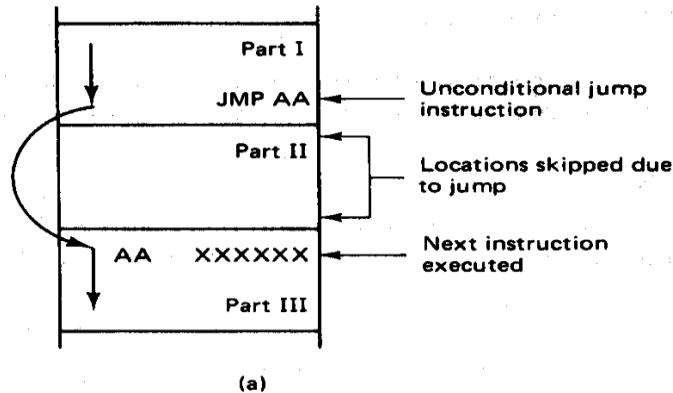


# 6.3 Control Flow and Jump Instructions— Unconditional and Conditional Jump Control Flow



- Jump operation alters the execution path of the instructions in the program—flow control
- **Unconditional** Jump
  - Always takes place
  - No status requirements are imposed
- Example (part a)
  - **JMP** AA instructions in Part I executed
  - Control passed to next instruction identified by AA in Part III
  - Instructions in Part II skipped

# 6.3 Control Flow and Jump Instructions– Unconditional and Conditional Jump Control Flow



- **Conditional** jump
  - May or may not take place
  - Status conditions must be satisfied
- Example (part b)
  - **Jcc AA** instruction in Part I executed
  - Conditional relationship specified by **cc** is evaluated
  - If conditions met, jump takes place and control is passed to next instruction identified by **AA** in Part III
  - Otherwise, execution continues **sequentially** with first instruction in Part II
- Condition cc specifies a relationship of status flags such as CF, PF, ZF, etc.

No return linkage is saved when the JUMP is performed

# 6.3 Control Flow and Jump Instructions— Unconditional Jump Instruction

Mnemonic	Meaning	Format	Operation	Affected flags
JMP	Unconditional jump	JMP Operand	Jump is initiated to the address specified by the operand	None

(a)

Operands
Short-label
Near-label
Far-label
Memptr16
Regptr16
Memptr32

(b)

CS:100	lab	ADD BX,1234
CS:104		INC AX
CS:106		JMP lab
CS:108		NEG BX

Unconditional jump instruction:

- Implements the unconditional jump operation needed by:

- Branch program control flow structures
- Loop program control flow structures

- General format:

JMP Operand

- Types of unconditional jumps

- **Intrasegment—branch** to address is located in the current code segment

- Only IP changes value

- **short-label**

- **8-bit** signed displacement coded into the instruction

- Immediate addressing

- Range equal **-126 to +129**

- New address computed as:

(Current IP) + short-label → (IP)

Jump to address = (Current CS) + (New IP)

- **near-label**

- **16-bit** signed displacement coded in the instruction

- Example

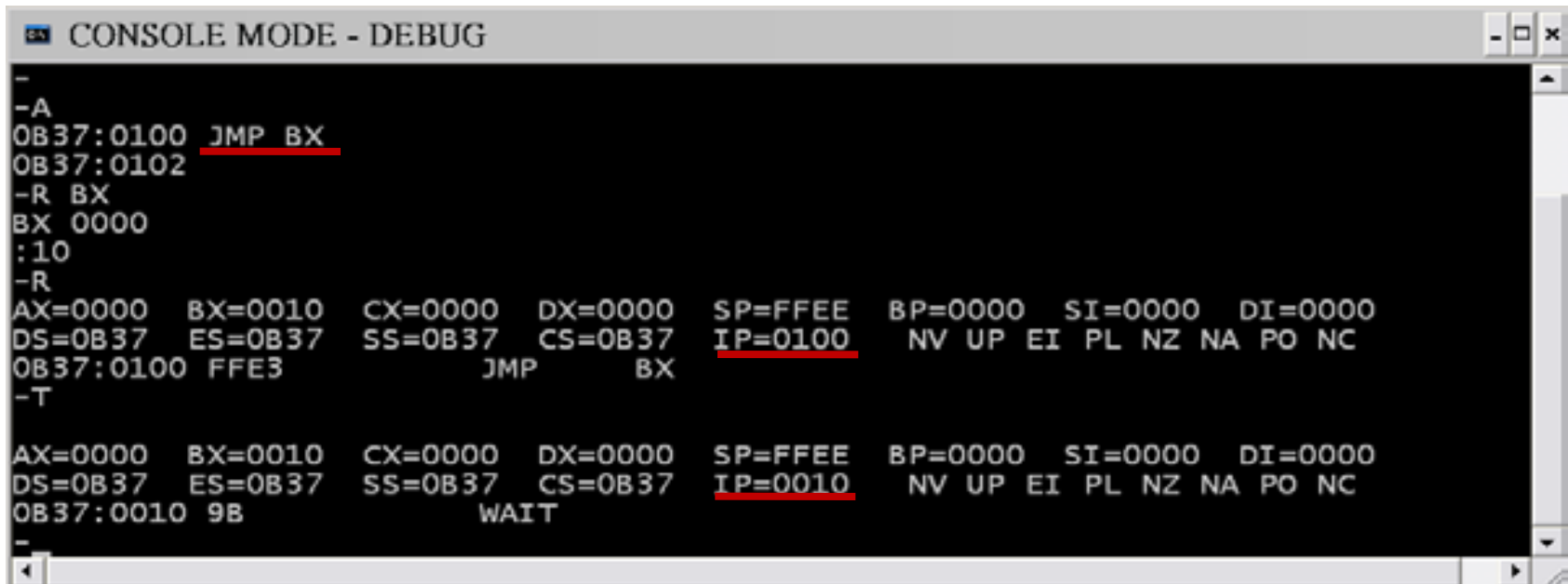
JMP 1234H

## 6.3 Control Flow and Jump Instructions – Unconditional Jump Instruction

### ▶ EXAMPLE

Verify the operation of the instruction `JMP BX` using the `DEBUG` program. Let the contents of `BX` be `001016`.

### ▶ Solution:



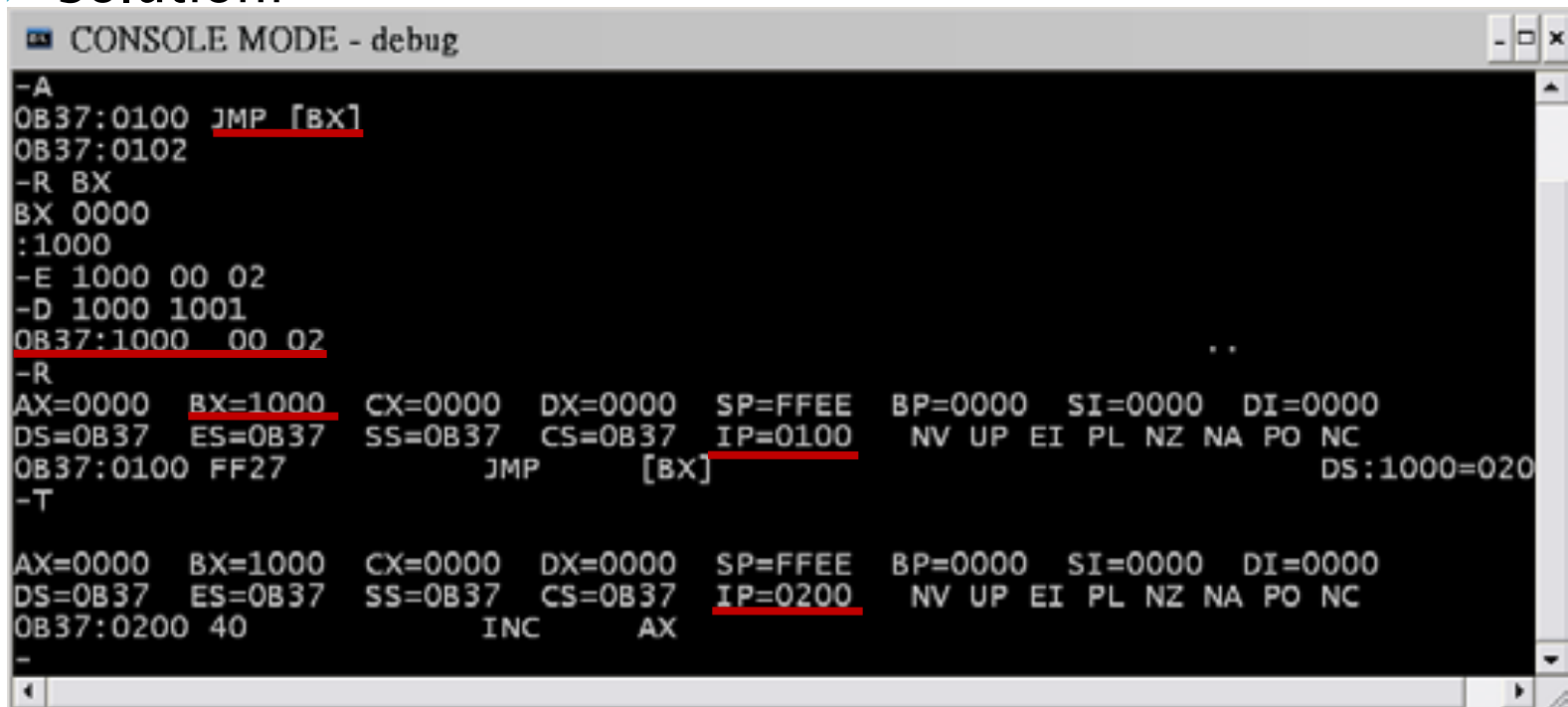
```
CONSOLE MODE - DEBUG
-
-A
0B37:0100 JMP BX
0B37:0102
-R BX
BX 0000
:10
-R
AX=0000  BX=0010  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B37  ES=0B37  SS=0B37  CS=0B37  IP=0100  NV UP EI PL NZ NA PO NC
0B37:0100 FFE3          JMP          BX
-T
AX=0000  BX=0010  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B37  ES=0B37  SS=0B37  CS=0B37  IP=0010  NV UP EI PL NZ NA PO NC
0B37:0010 9B          WAIT
-
```

## 6.3 Control Flow and Jump Instructions – Unconditional Jump Instruction

### ▶ EXAMPLE

Use the DEBUG program to observe the operation of the instruction `JMP [BX]`.

### ▶ Solution:



```
CONSOLE MODE - debug
-A
0B37:0100 JMP [BX]
0B37:0102
-R BX
BX 0000
:1000
-E 1000 00 02
-D 1000 1001
0B37:1000 00 02
-R
AX=0000 BX=1000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0100 NV UP EI PL NZ NA PO NC
0B37:0100 FF27 JMP [BX] DS:1000=020
-T
AX=0000 BX=1000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0200 NV UP EI PL NZ NA PO NC
0B37:0200 40 INC AX
-
```

## 6.3 Control Flow and Jump Instructions— Intersegment Unconditional Jump Operation

- **Intersegment—branch** to address is located in **another code segment**
  - Both **CS** and **IP** change values
  - **far-label**
    - **32-bit immediate** operand coded into the instruction
    - New address computed as:
      - **1st** 16 bits → (**IP**)
      - **2nd** 16 bits → (**CS**)
- **Jump to address = (New CS):(New IP)**
- **memptr32**
  - **32-bit** value specified in **memory**
  - Memory indirect addressing
- Example

**JMP DWORD PTR [DI]**

- Operation:
  - (**DS:DI**) → new **IP**
  - (**DS:DI + 2**) → new **CS**
  - Jump to address = (New CS):(New IP)

# 6.3 Control Flow and Jump Instructions– Conditional Jump Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
Jcc	Conditional jump	Jcc Operand	If the specified condition cc is true the jump to the address specified by the operand is initiated; otherwise the next instruction is executed.	None

(a)

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

(b)

- Condition jump instruction
- Implements the conditional jump operation
- General format:

Jcc Operand

- cc = one of the supported conditional relationships
- Supports the same operand types as unconditional jump
- Operation: Flags tested for conditions defined by cc and:

If cc test **True**:

IP, or IP and CS are updated with new value

- Jump is taken
- Execution resumes at jump to target address

If cc test **False**:

IP, or IP and CS are unchanged

- Jump is not taken
  - Execution continues with the next sequential instruction
- Examples of conditional tests:

JC = jump on carry → CF = 1

JPE/JP = jump on parity even → PF = 1

JE/JZ = jump on equal → ZF = 1

These instructions are associated with the compare instruction usually

# 6.3 Control Flow and Jump Instructions— Branch Program Structures

```

CMP    AX, BX
JE     EQUAL
---    ---
; Next instruction if (AX) ≠ (BX)

.
.
.
EQUAL: ---
; Next instruction if (AX) = (BX)

.
.
.
---    ---

```

- Example—**IF-THEN-ELSE**: comparing values

- One of the most widely used flow control program structure
- Implemented with **CMP**, **JE**, and **JMP** instructions
- Operation:
  - AX compared to BX to update flags
  - JE tests for ZF = 1
  - If (AX) ≠ (BX); ZF = 0 → **ELSE** path—next sequential instruction is executed
  - If (AX) = (BX); ZF = 1 → **THEN** path—instruction pointed to by **EQUAL** executes
  - **JMP** instruction used in **ELSE** path to bypass the **THEN** path.



# 6.3 Control Flow and Jump Instructions— Branch Program Structures

```

AND    AL, 04H
JNZ    BIT2_ONE
---    ---    ; Next instruction if B2 of AL = 0
.
.
.
---    ---
BIT2_ONE:
---    ---    ; Next instruction if B2 of AL = 1
.
.
.
---    ---

```

- Example—IF-THEN-ELSE using a **register bit** test
- Conditional test is made with JNZ instruction and branch taken if

ZF = 0

- Generation of test condition  
(AL) = xxxxxxxx AND 00000100  
= 00000x00

if **bit 2** = 1 ZF = 0 (not zero)  
if bit 2 = 0 ZF = 1

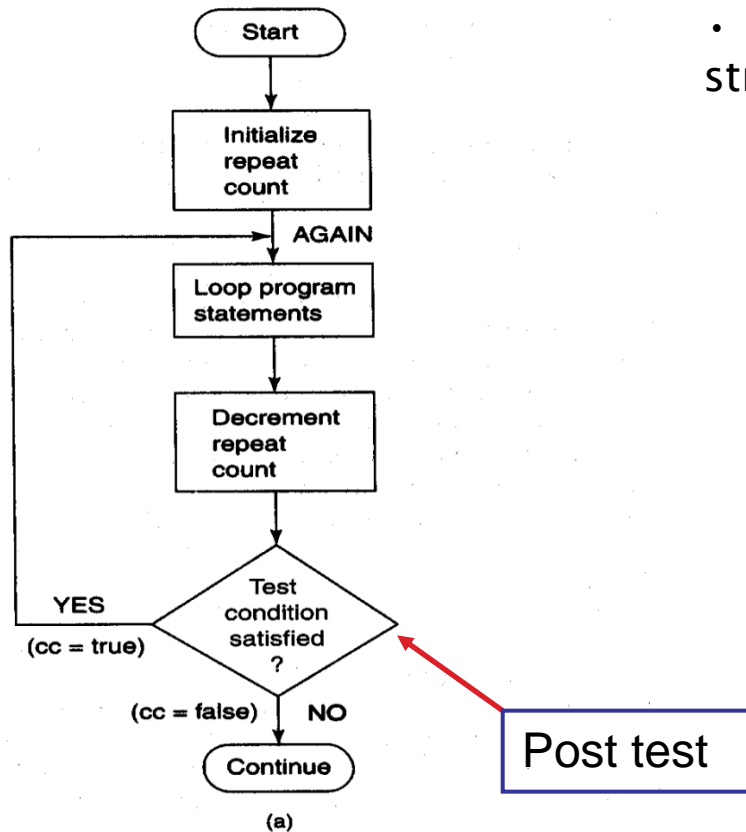
Therefore, jump to BIT2\_ONE only takes place if bit 2 of AL equals 1

- Same operation can be performed by shifting bit 2 to the CF and then testing with **JC**

CF = 1

# 6.3 Control Flow and Jump Instructions– Loop Program Structures

## Repeat Until structure



```

AGAIN:  MOV CL,COUNT    ;Set loop repeat count
        ---          ;1st instruction of loop
        ---          ;2nd instruction of loop
        .
        .
        .
        ---          ;nth instruction of loop
        DEC CL       ;Decrement repeat count by 1
        JNZ AGAIN    ;Repeat from AGAIN if (CL) ≠ 00H or (ZF) = 0
        ---          ;First instruction executed after the loop is
        ---          ;complete, (CL) = 00H, (ZF) = 1
    
```

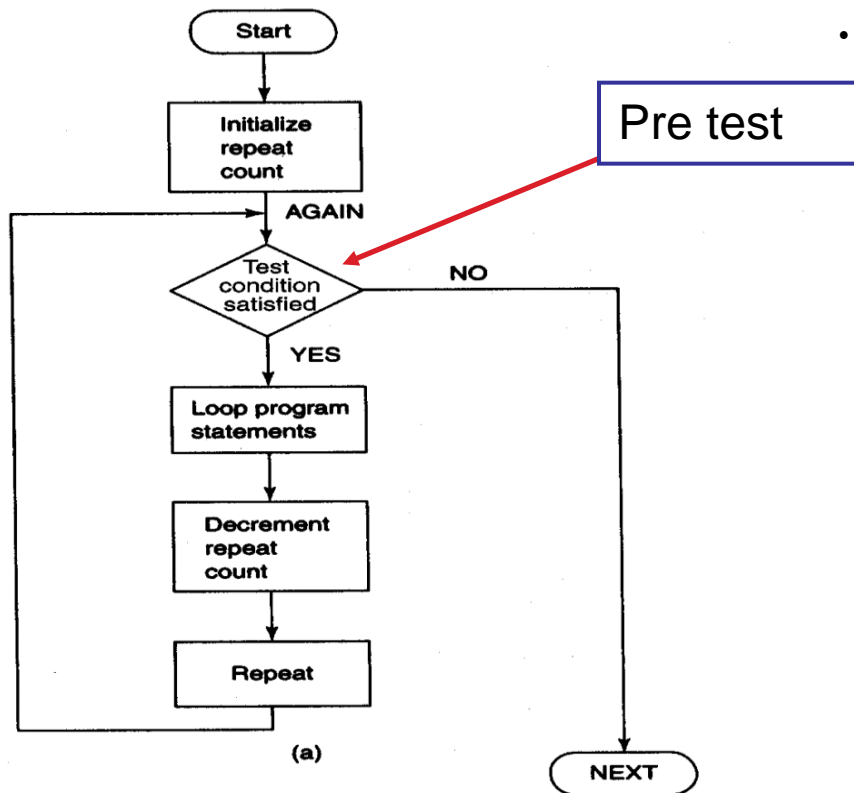
(b)

- Example—Repeat-Until program structure
  - Allows a **part of a program** to be conditionally **repeated** over and over
  - Employs post test—conditional test at end of sequence; always performs one iteration
  - Important **parameters**:
    - Initial count → count register
    - Terminal count → zero or other value
  - Program flow of control:
    - Initialize count  
**MOV CL,COUNT**
    - Perform body of loop operation  
AGAIN: --- --- first of multiple instructions
    - Decrement count  
**DEC CL**
    - Conditional test for completion  
**JNZ AGAIN**

Post test

# 6.3 Control Flow and Jump Instructions– Loop Program Structures

While do structure



- Example—While–Do program structure
  - Allows a part of a program to be conditionally repeated over and over
  - Employs pre–test—at entry of loop; may perform no iterations
  - Important parameters
    - Initial count → count register
    - Terminal count → zero or other value
  - Program flow/control:
    - Initialize count  
`MOV CL,COUNT`
    - Pre–test  
`AGAIN: JZ NEXT`
    - Perform body of loop operation  
--- --- first of multiple instructions
    - Decrement count  
`DEC CL`
    - Unconditional return to start of loop  
`JMP AGAIN`

```

AGAIN:  MOV CL,COUNT  ;Set loop repeat count
        JZ  NEXT   ;Loop is complete if CL = 00H (ZF = 1)
        ---      ;1st instruction of loop
        ---      ;2nd instruction of loop
        .
        .
        .
        ---      ;nth instruction of loop
        DEC CL    ;Decrement CL by 1
        JMP AGAIN ;Repeat from AGAIN
NEXT:   ---      ;First instruction executed after loop is complete
  
```

(b)

## 6.3 Control Flow and Jump Instructions – Loop Program Structures

### ▶ EXAMPLE

Implement an instruction sequence that calculates the absolute difference between the contents of AX and BX and places it in DX.

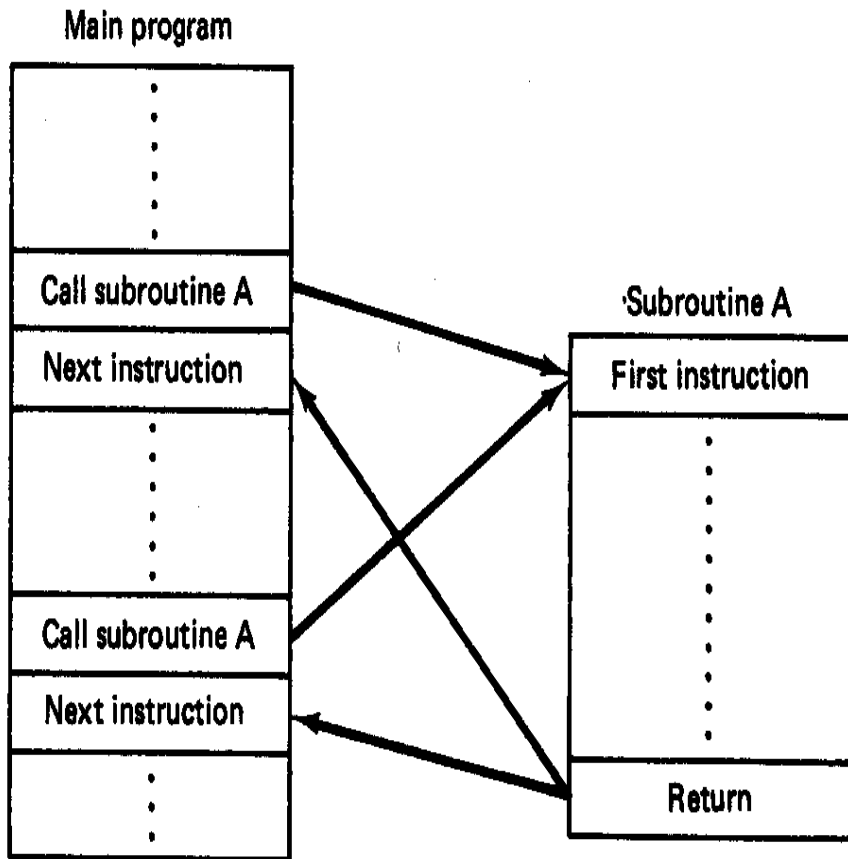
### ▶ Solution:

```
        CMP AX, BX
        JC DIFF2
DIFF1:  MOV DX, AX
        SUB DX, BX ; (DX)=(AX)-(BX)
        JMP DONE
DIFF2:  MOV DX, BX
        SUB DX, AX ; (DX)=(BX)-(AX)
DONE:  NOP
```

## *6.4 Subroutines and Subroutine-Handling Instructions*

- **Subroutine**—special segment of program that can be called for execution from any point in a program (**like function**)
- A subroutine is also known as a procedure.
- Program structure that implements HLL “functions” and “procedures”
- Written to perform an operation (function/procedure) that must be performed at various points in a program
- Written as a subroutine and only included once in the program
- A **return** instruction must be included at the end of the subroutine to initiate the return sequence to the main program environment.
- **CALL** and **RET** instructions
- **PUSH** and **POP** instructions

# 6.4 Subroutines and Subroutine-Handling Instructions



- Example:
  - Instruction in Main part of program calls “**Subroutine A**”
  - Program flow of control transferred to first instruction of Subroutine A
  - Instructions of Subroutine A execute sequentially
  - Return initiated by last instruction of Subroutine A
  - Same sequence repeated when the subroutine is called again later in the program
- Instructions:
  - **Call** instruction—initiates the subroutine from the main part of program
  - **Return** instruction—initiates return of control to the main program at completion of the subroutine
  - **Push** and **pop** instructions used to save register content and pass parameters

**The subroutine may be called and executed more than one time, but it is written one time**

# 6.4 Subroutines and Subroutine- Handling Instructions – Call Instruction

- Call Instruction
- Implements two types of calls:
  - Intra-segment call
  - Inter-segment call
- **Intra-segment** call—starting address of subroutine is located in the **current** code segment
- Only **IP** changes value
- **near-proc**
  - 16-bit offset coded in the instruction
  - Example  
**CALL 1234H**
- Operation:
  1. IP of next instruction saved on top of stack
  2. SP is decremented by 2
  3. New value from call instruction is loaded into IP
  4. Instruction fetch restarts with first instruction of subroutine

Current CS:**New IP**

Mnemonic	Meaning	Format	Operation	Flags Affected
CALL	Subroutine call	CALL operand	Execution continues from the address of the subroutine specified by the operand. Information required to return back to the main program such as IP and CS are saved on the stack.	None

(b)

Operand
Near-proc
Far-proc
Memptr16
Regptr16
Memptr32

(c)

## 6.4 Subroutines and Subroutine– Handling Instructions – Call Instruction

- **regptr16**
  - 16-bit value of IP specified as the content of a register
  - Register addressing
  - Example:  
`CALL BX`
  - Operation:
  - Same as near-proc except  
(BX) → New IP
- **memptr16**
  - 16-bit value of IP specified as the content of a storage location in memory
  - Memory addressing modes—register addressing
  - Example  
`CALL [BX]`
  - Same as near-proc except  
(DS:BX) → New IP



## 6.4 Subroutines and Subroutine– Handling Instructions – Call Instruction

- **Intersegment**—start address of the subroutine points to another code segment

- Both **CS** and **IP** change values
- **far-proc**
  - **32-bit** immediate operand coded into the instruction
  - New address computed as:
    - 1st 16 bits → New IP
    - 2nd 16 bits → New CS

Subroutine starts at = New CS:New IP

- **memptr32**
  - **32-bit** value specified in memory
  - Memory addressing modes—register indirect addressing
  - Example

**CALL DWORD PTR [DI]**

- Operation:
  - (DS:DI) → New IP
  - (DS:DI +2) → New CS
  - Starting address of subroutine = New CS:New IP

# 6.4 Subroutines and Subroutine- Handling Instructions – Return Instruction

- Return instruction
- Every subroutine must end with a return instruction
- Initiates return of execution to the instruction in the main program following that which called the subroutine

Mnemonic	Meaning	Format	Operation	Flags Affected
RET	Return	RET or RET Operand	Return to the main program by restoring IP (and CS for fat-proc). If Operand is present, it is added to the contents of SP.	None

(a)

Operand
None
Disp16

(b)

- Example:

**RET**

- Causes the value of IP (intra-segment return) or both IP and CS (inter-segment return) to be **popped from** the stack and put back into the IP and CS registers
- Increments SP by 2/4

# 6.4 Subroutines and Subroutine– Handling Instructions – Return Instruction

## ▶ EXAMPLE

```
TITLE EXAMPLE 6.10
PAGE                ,132
STACK_SEG S        SEGMENT STACK 'STACK'
                   DB 64 DUP(?)
STACK_SEG          ENDS
CODE_SEG SEGMENT   'CODE'
EX610              PROC    FAR
ASSUME CS:CODE_SEG, SS:STACK_SEG
;To return to DEBUG program put return address on the stack
                   PUSH DS
                   MOV AX, 0
                   PUSH AX
;Following code implements Example 6.10
                   CALL SUM
                   RET
SUM                PROC NEAR
                   MOV DX, AX
                   ADD DX, BX           ; (DX)=(AX)+(BX)
                   RET
SUM                ENDP
EX610              ENDP
CODE_SEG          ENDS
END                EX610
```

## 6.4 Subroutines and Subroutine- Handling Instructions – Return Instruction

### ▶ EXAMPLE

```
CONSOLE MODE - DEBUG EX610.EXE
-U 0 D
OBB1:0000 1E          PUSH    DS
OBB1:0001 B80000     MOV     AX,0000
OBB1:0004 50          PUSH    AX
OBB1:0005 E80100     CALL   0009
OBB1:0008 CB          RETF
OBB1:0009 8BD0     MOV     DX,AX
OBB1:000B 03D3     ADD     DX,BX
OBB1:000D C3          RET
-G 5

AX=0000  BX=0000  CX=0314  DX=0000  SP=003C  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=0005  NV UP EI PL NZ NA PO NC
OBB1:0005 E80100     CALL   0009
-R AX
AX 0000
:2
-R BX
BX 0000
:4
-
```

# 6.4 Subroutines and Subroutine- Handling Instructions – Return Instruction

## ▶ EXAMPLE

```
C:\> CONSOLE MODE - DEBUG EX610.EXE
-T
AX=0002  BX=0004  CX=0314  DX=0000  SP=003A  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=0009  NV UP EI PL NZ NA PO NC
0BB1:0009 8BD0          MOV     DX,AX
-D SS:3A 3B
0BAD:0030          08 00          ..
-T
AX=0002  BX=0004  CX=0314  DX=0002  SP=003A  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=000B  NV UP EI PL NZ NA PO NC
0BB1:000B 03D3          ADD     DX,BX
-T
AX=0002  BX=0004  CX=0314  DX=0006  SP=003A  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=000D  NV UP EI PL NZ NA PE NC
0BB1:000D C3          RET
-
```

## 6.4 Subroutines and Subroutine- Handling Instructions – Return Instruction

### ▶ EXAMPLE

```
CONSOLE MODE
-T
AX=0002  BX=0004  CX=0314  DX=0006  SP=003A  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=000D  NV UP EI PL NZ NA PE NC
0BB1:000D C3          RET
-T
AX=0002  BX=0004  CX=0314  DX=0006  SP=003C  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=0008  NV UP EI PL NZ NA PE NC
0BB1:0008 CB          RETF
-G
Program terminated normally
-Q
C:\>
```

# 6.4 Subroutines and Subroutine– Handling Instructions – Structure of a Subroutine

To save registers and parameters on the stack

{ PUSH XX  
PUSH YY  
PUSH ZZ

Main body of the subroutine

{ .  
. .  
. .  
. .  
. .

To restore registers and parameters from the stack  
Return to main program

{ POP ZZ  
POP YY  
POP XX  
RET

- **Elements** of a subroutine
  - Save of information to stack—**PUSH**
  - Main body of subroutine—Multiple instructions
  - Restore of information from stack—**POP**
  - Return to main program—**RET**
- **Save** of information
  - Must save content of registers/memory locations to be used or other program parameters (FLAGS)
  - **PUSH, PUSHF**
- **Main** body
  - Retrieve **input** parameters passed from main program via stack—stack pointer indirect address
  - Performs the **algorithm/function/operation** required of the subroutine
  - Prepare **output** parameters/results for return to main body via stack—stack pointer indirect addressing
- **Restore** information
  - Register/memory location contents saved on stack at entry of subroutine must be restored before return to main program—**POP, POPF**

# 6.4 Subroutines and Subroutine– Handling Instructions – Push and Pop Instruction

- Push instruction
  - General format:

**PUSH S**

- Saves a value on the stack—content of:
  - Register/segment register
  - Memory
- Example:

**PUSH AX**

$(AH) \rightarrow ((SP)-1)$

$(AL) \rightarrow ((SP)-2)$

$(SP)-2 \rightarrow (SP) = \text{New top of stack}$

- Pop instruction
  - General format:

**POP D**

- Restores a value on the stack—content to: register, segment register, memory

• Example:

**POP AX**

$((SP)) \rightarrow AL$

$((SP)+1) \rightarrow AH$

$((SP)+2) \rightarrow SP = \text{Old top of stack}$

Mnemonic	Meaning	Format	Operation	Flags Affected
PUSH	Push word onto stack	PUSH S	$((SP)) \leftarrow (S)$ $(SP) \leftarrow (SP)-2$	None
POP	Pop word off stack	POP D	$(D) \leftarrow ((SP))$ $(SP) \leftarrow (SP)+2$	None

(a)

Operand (S or D)
Register
Seg-reg (CS illegal)
Memory

(b)



## 6.4 Subroutines and Subroutine– Handling Instructions – Push and Pop Instruction

### ▶ EXAMPLE

Write a procedure named SQUARE that squares the contents of BL and places the result in BX

### ▶ Solution:

```
;Subroutine:      SQUARE
;Description:    (BX)=square of (BL)
SQUARE  PROC  NEAR
    PUSH AX      ; Save the register to be used
    MOV AX, BX  ; Place the number in AL
    IMUL BL     ; Multiply with itself
    MOV BX, AX  ; Save the result
    POP AX     ; Restore the register used
    RET
SQUARE ENDP
```

## 6.4 Subroutines and Subroutine- Handling Instructions – Push Flag Instruction

- Push flags instruction
  - General formats:  
**PUSHF**
  - Saves flags onto the stack
  - Operation  
 $(\text{FLAGS}) \rightarrow ((\text{SP}))$   
 $(\text{SP})-2 \rightarrow (\text{SP}) = \text{New top of stack}$

Mnemonic	Meaning	Operation	Flags Affected
PUSHF	Push flags onto stack	$((\text{SP})) \leftarrow (\text{Flags})$ $(\text{SP}) \leftarrow (\text{SP})-2$	None
POPF	Pop flags from stack	$(\text{Flags}) \leftarrow ((\text{SP}))$ $(\text{SP}) \leftarrow (\text{SP})+2$	OF, DF, IF, TF, SF, ZF, AF, PF, CF

- Pop flags instruction
  - General formats:  
**POPF**
  - Restores flags from the stack  
 $((\text{SP})) \rightarrow \text{FLAGS}$   
 $(\text{SP})+2 \rightarrow (\text{SP}) = \text{Old top of stack}$

## 6.5 The Loop and the Loop-Handling Instructions – Loop Instructions

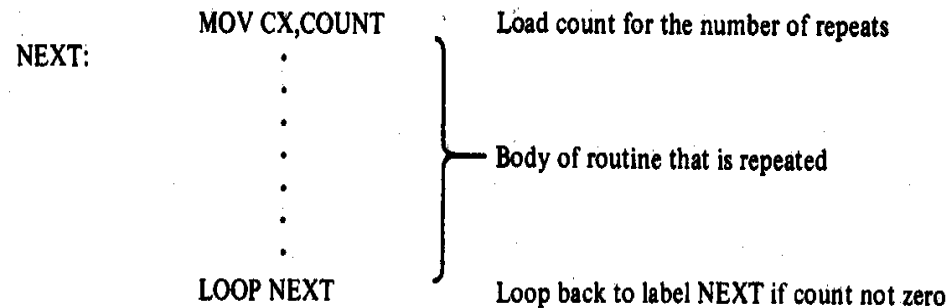
- **Loop**—segment of program that is repeatedly executed
  - Can be implemented with compare, conditional jump, and decrement instructions
- Loop instructions:
  - Special instructions that efficiently perform basic loop operations
  - Replace the multiple instructions with a single instruction
  - **LOOP**—loop while not zero (**while CX is not zero**)
    - $CX \neq 0$  — repeat while count not zero
    - **→Equivalent to Dec CX followed by JNZ**
  - **LOOPE/LOOPZ**— loop while equal
    - $CX \neq 0$  — repeat while count **not zero**, and
    - $ZF = 1$ —result of prior instruction was **equal**
  - **LOOPNE/LOOPNZ**—loop while not equal
    - $CX \neq 0$  — repeat while count **not zero**, and
    - $ZF = 0$ —result from prior instruction was **not equal**

**NOTE** All **LOOPS** instructions does not affect the flag register

## 6.5 The Loop and the Loop-Handling Instructions – Loop Instructions

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label	$(CX) \leftarrow (CX) - 1$ Jump is initiated to location defined by short-label if $(CX) \neq 0$ ; otherwise, execute next sequential instruction
LOOPE/LOOPZ	Loop while equal/ loop while zero	LOOPE/LOOPZ Short-label	$(CX) \leftarrow (CX) - 1$ Jump to location defined by short-label if $(CX) \neq 0$ and $(ZF) = 1$ ; otherwise, execute next sequential instruction
LOOPNE/ LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ Short-label	$(CX) \leftarrow (CX) - 1$ Jump to location defined by short-label if $(CX) \neq 0$ and $(ZF) = 0$ ; otherwise, execute next sequential instruction

# 6.5 The Loop and the Loop-Handling Instructions – Loop Program Structure and Operation



(a)

```

MOV    AX,DATASEGADDR
MOV    DS,AX
MOV    SI,BLK1ADDR
MOV    DI,BLK2ADDR
MOV    CX,N
NEXTPT: MOV    AH,[SI]
        MOV    [DI],AH
        INC    SI
        INC    DI
        LOOP  NXTPT
        HLT

```

(b)

- Structure of a loop
- **Initialization** of the count in CX
- **Body**—instruction sequence that is to be repeated; short label identifying beginning
- **Loop** instruction— determines if loop is complete or if the body is to repeat
- Example

1. Initialize data segment, source and destination block pointers, and loop count
2. Body of program is executed—source element read, written to destination, and then both pointers incremented by 1
3. Loop test
  - a. Contents of CX decremented by 1
  - b. Contents of CX check for zero
  - c. If  $CX = 0$ , loop is complete and next sequential instruction (HLT) is executed
  - d. If  $CX \neq 0$ , loop of code is repeated by returning control to the instruction corresponding to the Short-Label (NXTPT:) operand

## *6.5 The Loop and the Loop-Handling Instructions* – Loop Program Structure and Operation

### ➤ EXAMPLE

Given the following sequence of instructions, explain what happens as they are executed.

```
MOV DL, 05
MOV AX, 0A00H
MOV DS, AX
MOV SI, 0
MOV CX, 0FH
AGAIN: INC SI
      CMP [SI], DL
      LOOPNE AGAIN
```

# 6.5 The Loop and the Loop-Handling Instructions – Loop Program Structure and Operation

## ➤ EXAMPLE

```
CONSOLE MODE - DEBUG EX615.EXE
C:\>DEBUG EX615.EXE
-U 0 17
0BB1:0000 1E          PUSH    DS
0BB1:0001 B80000          MOV     AX,0000
0BB1:0004 50          PUSH    AX
0BB1:0005 B205          MOV     DL,05
0BB1:0007 B8000A          MOV     AX,0A00
0BB1:000A 8ED8          MOV     DS,AX
0BB1:000C BE0000          MOV     SI,0000
0BB1:000F B90F00          MOV     CX,000F
0BB1:0012 46          INC     SI
0BB1:0013 3814          CMP     [SI],DL
0BB1:0015 EOFB          LOOPNZ 0012
0BB1:0017 CB          RETF
-G 12

AX=0A00  BX=0000  CX=000F  DX=0005  SP=003C  BP=0000  SI=0000  DI=0000
DS=0A00  ES=0B9D  SS=0BAD  CS=0BB1  IP=0012  NV UP EI PL NZ NA PO NC
0BB1:0012 46          INC     SI
-
```

# 6.5 The Loop and the Loop-Handling Instructions – Loop Program Structure and Operation

## ➤ EXAMPLE

```
CONSOLE MODE - DEBUG EX615.EXE
-
-E A00:0 4, 6, 3, 9, 5, 6, D, F, 9
-D A00:0 F
0A00:0000 04 06 03 09 05 06 0D 0F-09 38 75 19 99 61 16 27 .....8u..a.
-G 17

AX=0A00  BX=0000  CX=000B  DX=0005  SP=003C  BP=0000  SI=0004  DI=0000
DS=0A00  ES=0B9D  SS=0BAD  CS=0BB1  IP=0017  NV UP EI PL ZR NA PE NC
0BB1:0017 CB          RETF
-G

Program terminated normally
-
```



# 6.6 String and String-Handling Instructions – String Instructions

- **String**—series of bytes or words of data that reside at consecutive memory addresses
  - **String instructions**
    - Special instructions that efficiently perform basic string operations
    - Replaces multiple instructions with a single instruction
- Examples
  - Move string
  - Compare string
  - Scan string
  - Load string
  - Store string
  - Repeated string
- Typical string operations
  - Move a string of data elements from one part of memory to another—block move
  - Scan through a string of data elements in memory looking for a specific value
  - Compare the elements of two strings of data elements in memory to determine if they are the same or different
  - Initialize a group of consecutive storage locations in memory

# 6.6 String and String-Handling Instructions – String Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
MOVS	Move string	MOVSB/MOVSW	$((ES)0 + (DI)) \leftarrow ((DS)0 + (SI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	None
CMPS	Compare string	CMPSB/CMPSW	Set flags as per $((DS)0 + (SI)) - ((ES)0 + (DI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	CF, PF, AF, ZF, SF, OF
SCAS	Scan string	SCASB/SCASW	Set flags as per $(AL \text{ or } AX) - ((ES)0 + (DI))$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	CF, PF, AF, ZF, SF, OF
LODS	Load string	LODSB/LODSW	$(AL \text{ or } AX) \leftarrow ((DS)0 + (SI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$	None
STOS	Store string	STOSB/STOSW	$((ES)0 + (DI)) \leftarrow (AL \text{ or } AX) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	None

## 6.6 String and String-Handling Instructions – Move String Instruction

- Move string instruction
  - Used to move an element of data between a source and destination location in memory:
  - General format:
    - MOVSB**—move string byte
    - MOVSW**—move string word
  - Operation: Copies the content of the source to the destination; autoincrements/decrements both the source and destination addresses  
 **$((DS)0+(SI)) \rightarrow ((ES)0+(DI))$**   
 $(SI) \pm 1 \text{ or } 2 \rightarrow (SI)$   
 $(DI) \pm 1 \text{ or } 2 \rightarrow (DI)$
  - **Direction flag** determines increment/decrement  
DF = 0  $\rightarrow$  autoincrement  
DF = 1  $\rightarrow$  autodecrement

## 6.6 String and String-Handling Instructions – Move String Instruction

```
MOV     AX,DATASEGADDR
MOV     DS,AX
MOV     ES,AX
MOV     SI,BLK1ADDR
MOV     DI,BLK2ADDR
MOV     CX,N
CLD
NXTPT: MOVSB
        LOOP  NXTPT
        HLT
```

Reset



- Application example— The block-move program using the move-string instruction:
  1. Initialize DS & ES to same value
  2. Load SI and DI with block starting addresses
  3. Load CX with the count of elements in the string
  4. ~~Set DF for~~ autoincrement
  5. Loop on string move to copy N elements
- **MOVSB** and **LOOP** replaces multiple move and increment/decrement instructions

## 6.6 String and String-Handling Instructions – Compare/Scan String Instructions

- **Compare** string instruction
  - Used to compare the destination element of data in memory to the source element in memory and reflect the result of the comparison in the flags
  - General format:  
**CMPSB,SW**—compare string byte, word
  - Operation: Compares the content of the destination to the source; updates the flags; autoincrements/decrements both the source and destination addresses  
 $((DS)0+(SI)) - ((ES)0+(DI))$   
update status flags  
 $(SI) \pm 1 \text{ or } 2 \rightarrow (SI)$   
 $(DI) \pm 1 \text{ or } 2 \rightarrow (DI)$
- **Scan** string instruction—**SCAS**
  - Same operation as **CMPS** except destination is compared to a value in the **accumulator** (A) register  
 $(AL,AX) - ((ES)0+(DI))$

## 6.6 String and String-Handling Instructions – Compare/Scan String Instructions

```
MOV     AX,0
MOV     DS,AX
MOV     ES,AX
MOV     AL,05
MOV     DI,0A000H
MOV     CX,0FH
CLD
AGAIN:  SCASB
        LOOPNE AGAIN
NEXT:
```

- Application example—block scan:
  1. Initialize DS & ES to same value
  2. Load AL with search value; DI with block starting address; and CX with the count of elements in the string; **clear DF**
  3. Loop on scan string until the first element equal to 05H is found

## 6.6 String and String-Handling Instructions – Load/Store String Instructions

- **Load** string instruction
- Used to load a source element of data from memory into the accumulator register.
- General format:  
    **LODSB,SW**—load string byte, word
- Operation: Loads the content of the source element in the accumulator; autoincrements/decrements the source addresses  
     $((DS)0+(SI)) \rightarrow (AL \text{ or } AX)$   
    update status flags  
     $(SI) \pm 1 \text{ or } 2 \rightarrow (SI)$
- **Store** string instruction—**STOS**
- Same operation as **LODS** except value in accumulator is stored in destination is memory  
     $(AL,AX) \rightarrow ((ES)0+(DI))$

## 6.6 String and String-Handling Instructions – Load/Store String Instructions

```
MOV     AX,0
MOV     DS,AX
MOV     ES,AX
MOV     AL,05
MOV     DI,0A000H
MOV     CX,0FH
CLD
AGAIN:  STOSB
        LOOP    AGAIN
```

- Application example—  
initializing a block of memory  
with a store string instruction:
  1. Initialize DS & ES to same  
value
  2. Load AL with initialization  
value; DI with block starting  
address, CX with the count of  
elements in the string; and  
clear DF
  3. Loop on store string until  
all element of the string are  
initialized to 05H

How many times will this loop execute ?



## 6.6 String and String-Handling Instructions – Repeat String Instructions

- **Repeat** string—in most applications the basic string operations are repeated
  - Requires addition of loop or compare & conditional jump instructions
  - Repeat prefix provided to make coding of repeated string more efficient
  - Repeat prefixes
    - **REP**
      - $CX \neq 0$  — repeat while not end of string
      - Used with: **MOVS** and **STOS**
    - **REPE/REPZ**
      - $CX \neq 0$ —repeat while not end of string, and  
ZF = 1—strings are equal
      - Used with: **CMPS** and **SCAS**
    - **REPNE/REPZ**—Used with: **CMPS** and **SCAS**
      - $CX \neq 0$ —repeat while not end of string, and  
ZF = 0—strings are not equal
      - Used with: **CMPS** and **SCAS**

## 6.6 String and String-Handling Instructions – Repeat String Instructions

Prefix	Used with:	Meaning
REP	MOVS STOS	Repeat while not end of string CX $\neq$ 0
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal CX $\neq$ 0 and ZF = 1
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string and strings are not equal CX $\neq$ 0 and ZF = 0

## 6.6 String and String-Handling Instructions – Repeat String Instructions

```
MOV     AX,0
MOV     DS,AX
MOV     ES,AX
MOV     AL,05
MOV     DI,0A000H
MOV     CX,0FH
CLD
REPSTOSB
```

- General format:

**REPXXXX**

Where: XXXX = one of string instructions

- Examples:

**REPMOVB**

**REPESCAS**

**REPNECAS**

- Application example initializing a block of memory:

1. Initialize DS & ES to same value
2. Load AL with initialization value; DI with block starting address, and CX with the count of elements in the string
4. Clear the direction flag for autoincrement mode
4. Repeat store string until all elements of the string are initialized to 05H

# 6.6 String and String-Handling Instructions - Repeat String Instructions

## ➤ EXAMPLE

```
CONSOLE MODE - DEBUG EX617.EXE
-
-U 0 18
OBB5:0000 1E          PUSH    DS
OBB5:0001 B80000          MOV     AX,0000
OBB5:0004 50          PUSH    AX
OBB5:0005 B8B10B          MOV     AX,0BB1
OBB5:0008 8ED8          MOV     DS,AX
OBB5:000A 8EC0          MOV     ES,AX
OBB5:000C FC          CLD
OBB5:000D B92000          MOV     CX,0020
OBB5:0010 BE0000          MOV     SI,0000
OBB5:0013 BF2000          MOV     DI,0020
OBB5:0016 F3          REPZ
OBB5:0017 A4          MOVSB
OBB5:0018 CB          RETF
-G 16

AX=0BB1  BX=0000  CX=0020  DX=0000  SP=003C  BP=0000  SI=0000  DI=0020
DS=0BB1  ES=0BB1  SS=0BAD  CS=0BB5  IP=0016  NV UP EI PL NZ NA PO NC
OBB5:0016 F3          REPZ
OBB5:0017 A4          MOVSB
-
```

# 6.6 String and String-Handling Instructions - Repeat String Instructions

## ➤ EXAMPLE

```
C:\> CONSOLE MODE - DEBUG EX617.EXE
-F DS:0 1F FF
-F DS:20 3F 00
-D DS:0 3F
OBB1:0000  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0010  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
OBB1:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
-G 18
AX=0BB1  BX=0000  CX=0000  DX=0000  SP=003C  BP=0000  SI=0020  DI=0040
DS=0BB1  ES=0BB1  SS=0BAD  CS=0BB5  IP=0018  NV UP EI PL NZ NA PO NC
OBB5:0018  CB                RETF
-D DS:0 3F
OBB1:0000  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0010  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0020  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0030  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
-G
Program terminated normally
-
```

# H.W. #6

- Solve the following problems from Chapter 6 from the course textbook:

1, 8, 14, 28, 39, 43