Mansoura University
Faculty of Computers and Information
Department of Computer Science
First Semester- 2020-2021

[CS212P] Computer Organization and Architecture

Lecture 3: Instruction Set (Assembly Language)

Grade: 2nd General / 3rd Programs

DR. Muhammad Haggag Zayyan

CS Department

LECTURE TOPICS

- Assembly language
- Evolution of Intel's Microprocessors
- 8086 CPU Architecture
- 8086 Central Processor Unit (CPU): Registers
- Addressing Modes
- 8086 Microprocessor Instructions
- Interrupts

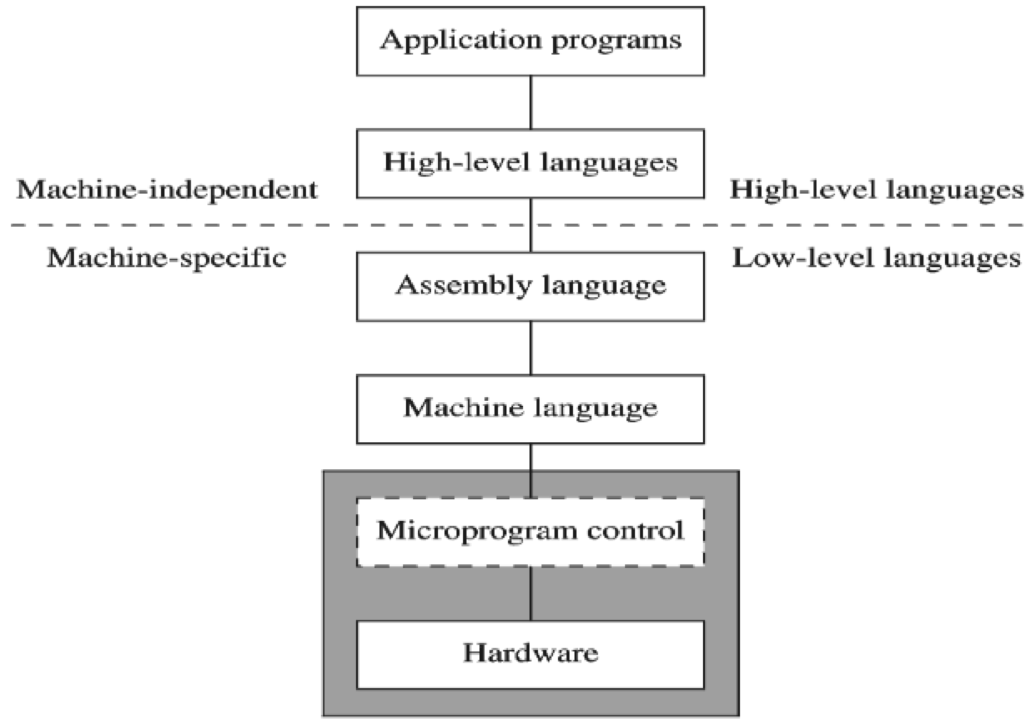
WHAT IS ASSEMBLY LANGUAGE?

- A **processor** understands only machine language instructions, which are strings of 1's and 0's. However, machine language is too obscure and complex for using in software development.

- So, the *low-level assembly language* is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

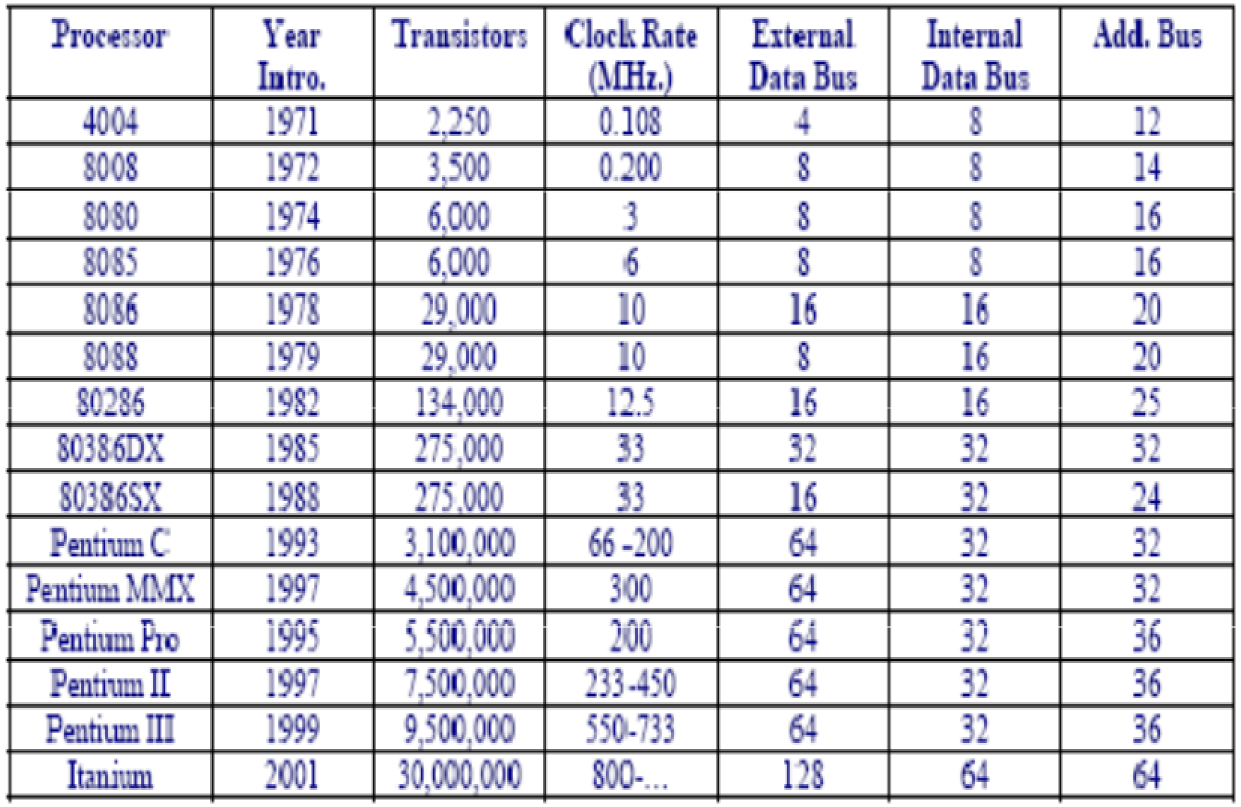
A HIERARCHY OF LANGUAGES

- A **language** is a formal language designed to communicate instructions to a computer



EVOLUTION OF INTEL'S MICROPROCESSORS

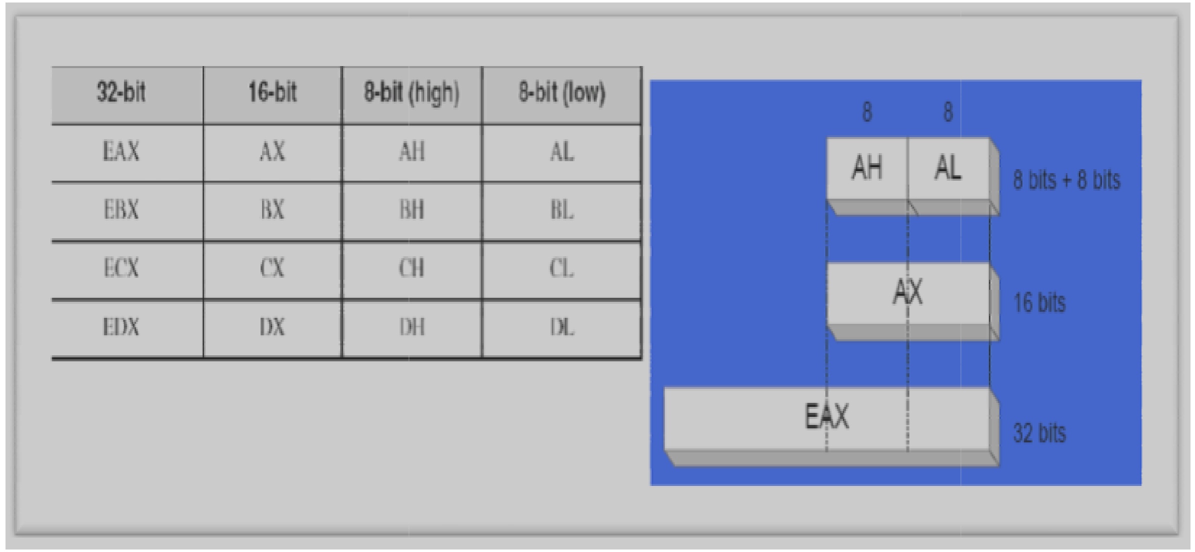
- Processor vary in their speed, capacity of memory, register and data bus



Processor	Year Intro.	Transistors	Clock Rate (MHz.)	External Data Bus	Internal Data Bus	Add. Bus
4004	1971	2,250	0.108	4	8	12
8008	1972	3,500	0.200	8	8	14
8080	1974	6,000	3	8	8	16
8085	1976	6,000	6	8	8	16
8086	1978	29,000	10	16	16	20
8088	1979	29,000	10	8	16	20
80286	1982	134,000	12.5	16	16	25
80386DX	1985	275,000	33	32	32	32
80386SX	1988	275,000	33	16	32	24
Pentium C	1993	3,100,000	66 -200	64	32	32
Pentium MMX	1997	4,500,000	300	64	32	32
Pentium Pro	1995	5,500,000	200	64	32	36
Pentium II	1997	7,500,000	233-450	64	32	36
Pentium III	1999	9,500,000	550-733	64	32	36
Itanium	2001	30,000,000	800-...	128	64	64

ACCESSING PARTS OF REGISTERS

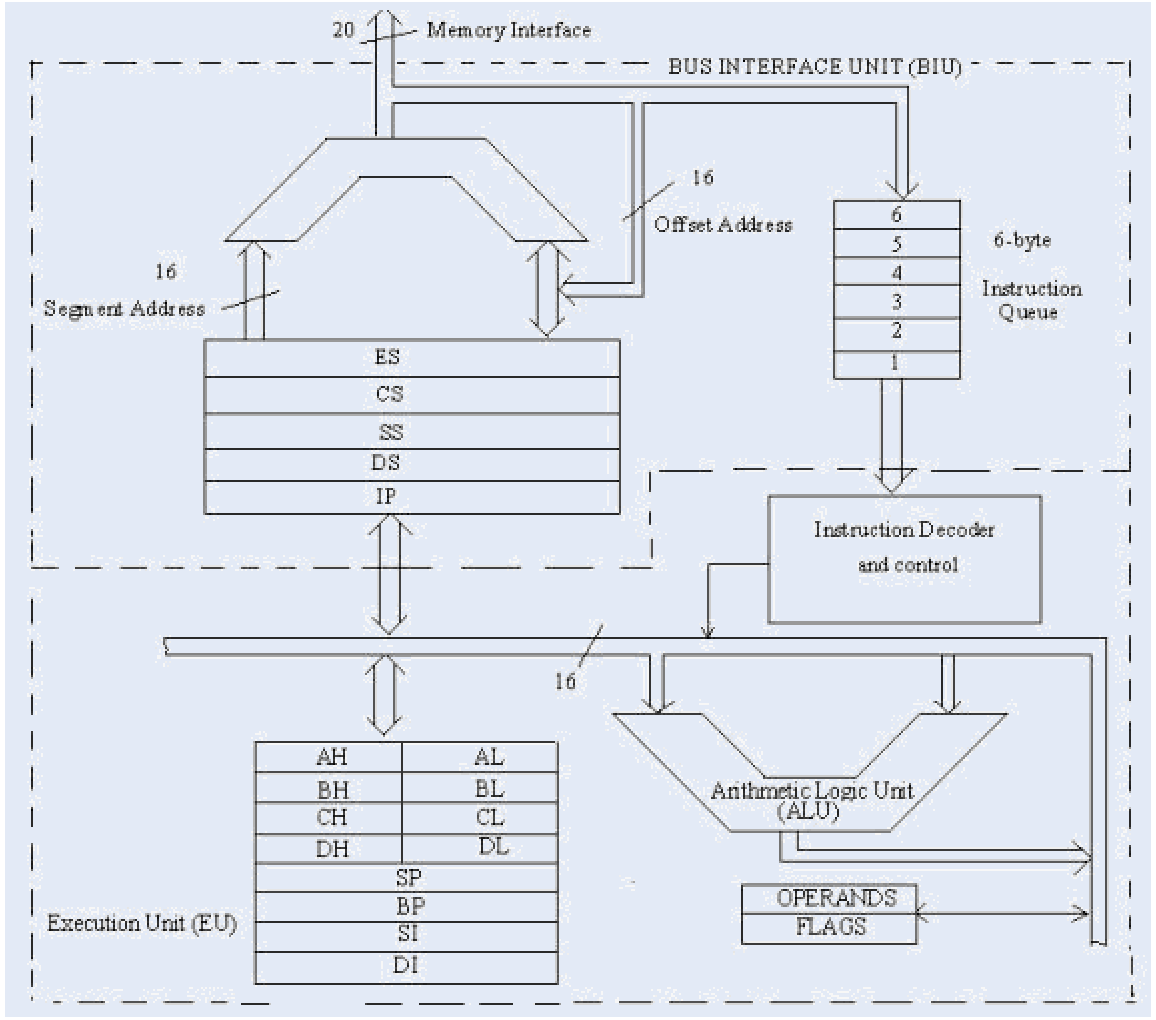
- Use 8-bit name, 16-bit name, or 32-bit name: Applies to EAX, EBX, ECX, and EDX



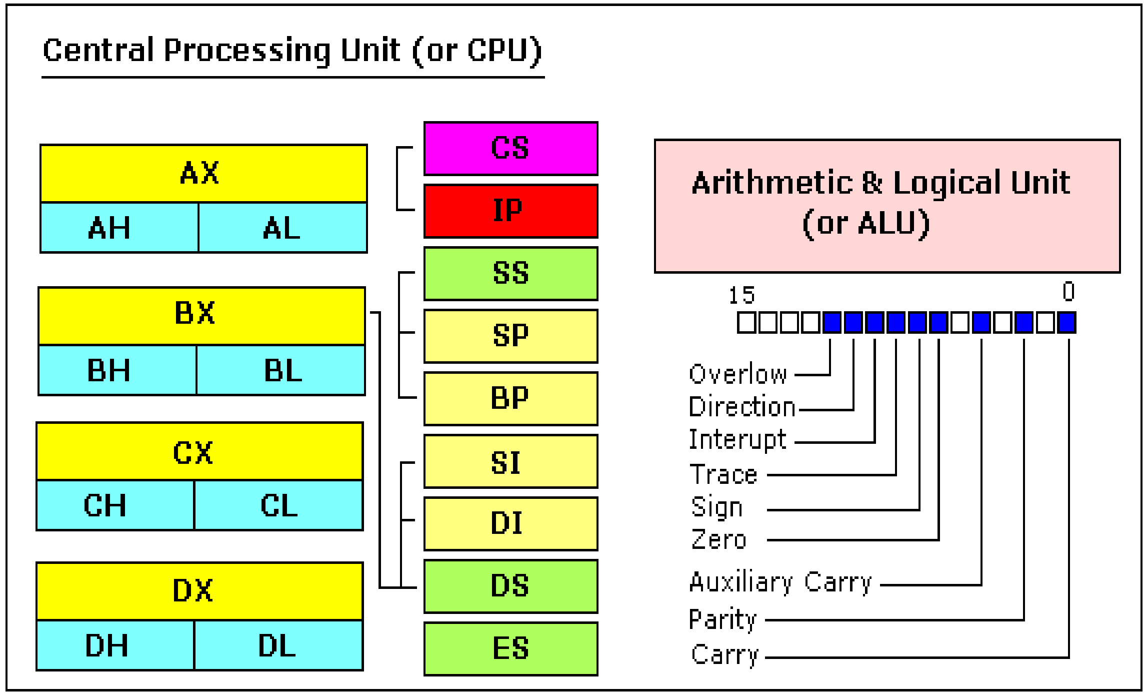
32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

BASIC ARCHITECTURE OF 8086 MICROPROCESSOR

- Bus Interface Unit.
  - Fetch the instruction or data from memory.
- Execution Unit.
  - To tell BIU where to fetch the instructions or data from.
  - To **decode** the instructions.
  - To **execute** the instructions.



INSIDE THE 8086 CENTRAL PROCESSOR UNIT (CPU)



- .

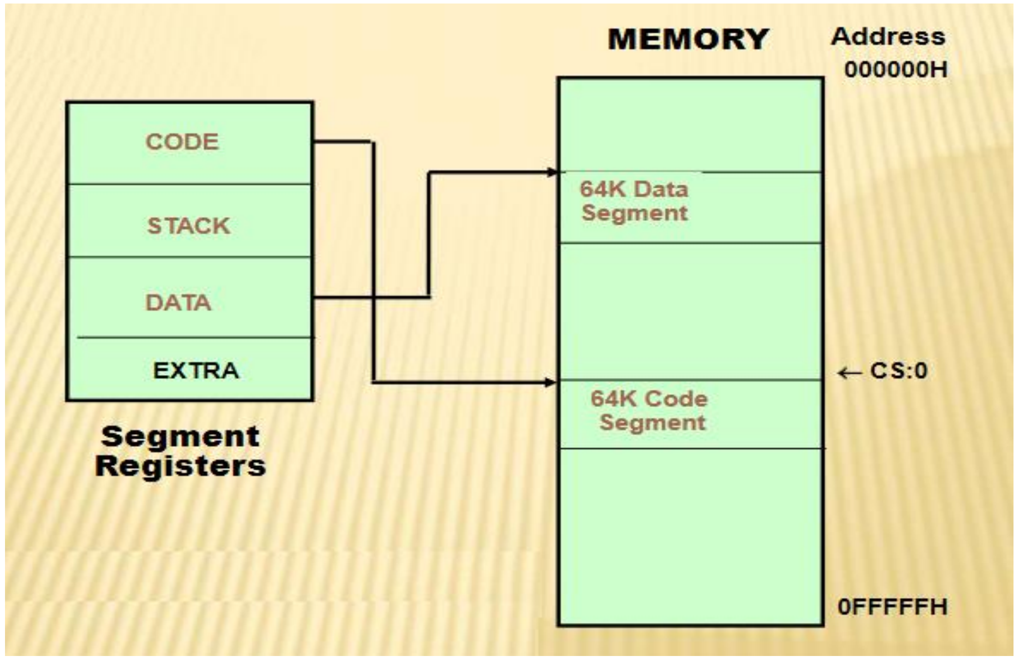
GENERAL PURPOSE REGISTERS

- **AX** - the accumulator register (divided into **AH / AL**)
- **BX** - the base address register (divided into **BH / BL**)
- **CX** - the count register (divided into **CH / CL**)
- **DX** - the data register (divided into **DH / DL**)

- **SI** - source index register
- **DI** - destination index register
- **BP** - base pointer
- **SP** - stack pointer

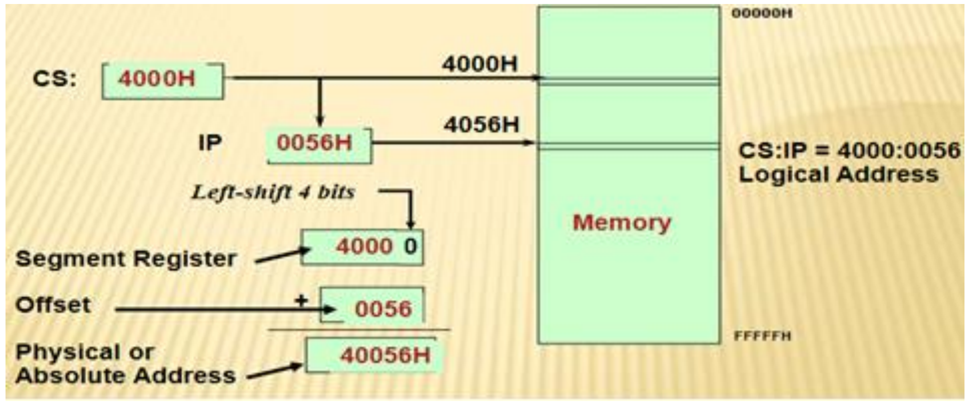
SEGMENT REGISTERS

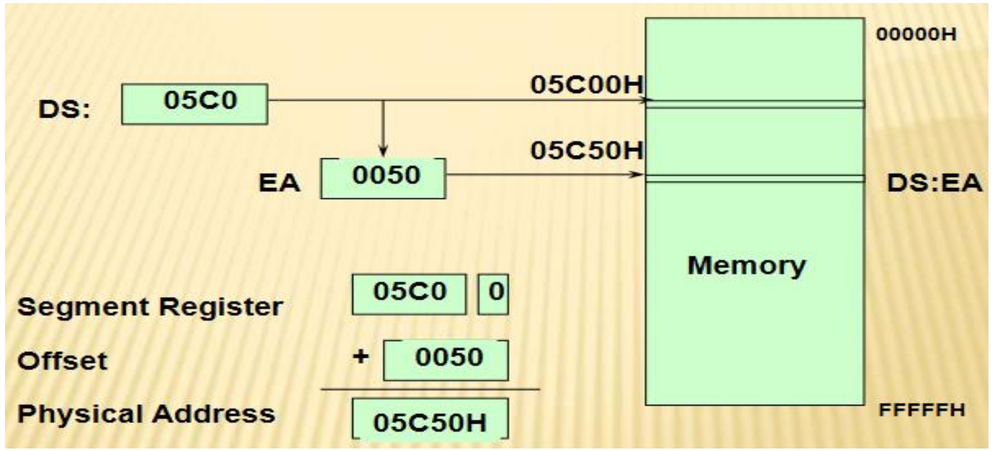
- **CS** - points at the segment containing the current program.
- **DS** - generally points at segment where variables are defined.
- **ES** - extra segment register, it's up to a coder to define its usage.
- **SS** - points at the segment containing the stack.



ACCESS MUCH MORE MEMORY THAN WITH A SINGLE REGISTER THAT IS LIMITED TO 16 BIT VALUES.

- Physical address = Segment register * 10_H + General purpose register (offset)





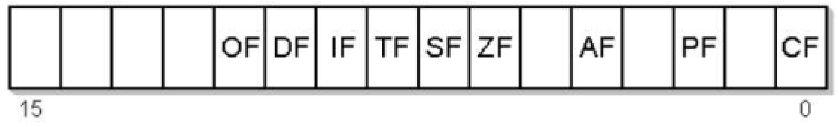
SPECIAL PURPOSE REGISTERS

- **IP** - the instruction pointer:
  - Always points to next instruction to be executed
  - Offset address relative to CS

- **IP** register always works together with **CS** segment register and it points to currently executing instruction.

FLAGS REGISTER

- **Flags Register** is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program.



- **Carry Flag (CF) -** this flag is set to **1** when there is an **unsigned overflow**. For example when you add bytes 255 + 1
- **Parity Flag (PF) -** this flag is set to **1** when there is even number of one bits in result. Even if result is a word only 8 low bits are analyzed!
- **Auxiliary Flag (AF) -** set to **1** when there is an **unsigned overflow** for low nibble (4 bits).
- **Zero Flag (ZF) -** set to **1** when result is **zero**. For none zero result this flag is set to **0**.
- **Sign Flag (SF) -** set to **1** when result is **negative**. When result is **positive** it is set to **0**.
- **Trap Flag (TF) -** Used for on-chip debugging.
- **Interrupt enable Flag (IF) -** when this flag is set to **1** CPU reacts to interrupts from external devices.
- **Direction Flag (DF) -** this flag is used by some instructions to process data chains, when this flag is set to **0** - the processing is done forward.
- **Overflow Flag (OF) -** set to **1** when there is a **signed overflow**. For example, when you add bytes **100 + 50** (result is not in range -128...127).

ADDRESSING MODES

Assuming the following data

- DATA1 **DW** 25H	int DATA1 = 0x25;

DATA1 is defined as a word (16-bit) variable, i.e., a memory location that contains 25H.

- DATA2 **EQU** 20H	const int DATA2 = 0x20;

DATA2 is not a memory location but a constant

1. **Direct Addressing**

   MOV AX, DATA1

   [DATA1] → AX, the contents of DATA1 is put into AX.

   The CPU goes to memory to get data. 25H is put in AX.

2. **Immediate Addressing**

   MOV AX, DATA2

   DATA2 = 20H → AX, 20H is put in AX.

   Does not go to memory to get data.

   Data is in the instruction.

   MOV AX, OFFSET DATA1 (pointer)

   The offset of the variable is just a number.

3. **Register Addressing**

   MOV AX, BX	AX ← BX

4. **Register Indirect Addressing**

   MOV AX, [BX]	AX ← DS:BX

   (location of an operand is held in a register)

8086 MICROPROCESSOR INSTRUCTIONS

- **Data Transfer Instructions**
  - **mov, push, echg, ..**
- **Arithmetic Instructions**
  - **add, mul, div, ..**
- **Logical Instructions**
  - **and, or, xor, ..**
- **Condition/Branch instructions**
  - cmp, jmp, loop, ..

DATA TRANSFER INSTRUCTIONS

- **MOV instruction** copies data from one location to another.
- *Format :* **MOV destination, source**
- The MOV instruction may have one of the following five forms:
  - MOV register, register
  - MOV register, immediate
  - MOV memory, immediate
  - MOV register, memory
  - MOV memory, register

Operand Types:

- **Immediate**
- **Register**
- **Memory**

MOV - CONTINUE

- Both the operands in MOV operation should be of same size
- The value of source operand remains unchanged

- For example, instruction

```
mov     [ESI], al
```

; Store a byte-size value in memory location pointed by ESI

suggests that an 8-bit quantity should be moved because AL is an 8-bit register.

When instruction has no reference to operand size,

```
mov     [ESI], 5
```

; Error: operand must have the size specified

- To get around this instance, we must use a *pointer directive*, such as

```
mov     BYTE PTR [ESI], 5       ; Store 8-bit value
mov     WORD PTR [ESI], 5       ; Store 16-bit value
mov     DWORD PTR [ESI], 5      ; Store 32-bit value
```

PUSH AND POP

- **Stacks Data Structure:**
  - A stack is an array-like data structure in the memory in which data can be stored and removed from a location called the 'top' of the stack.
  - The data need to be stored is 'pushed' into the stack and data to be retrieved is 'popped' out from the stack.
  - Stack is a LIFO data structure, i.e., the data stored first is retrieved last.

- *Format:*
- **PUSH address/register**
- **POP address/register**

- **Note:** Only words or double-words could be saved into the stack, not a byte.

XCHG

- **XCHG instruction** swap the two data items
- As in the MOV instruction, both operands cannot be located in memory.
- It can take one of the following forms:
  - XCHG register, register
  - XCHG register, memory
  - XCHG memory, register
- The XCHG instruction do not need a third register to hold a temporary value in order to swap two values. For example, we need three MOV instructions to perform exchange AX,DX registers.
  - MOV CX,AX
  - MOV AX,DX
  - MOV DX,CX

- **Example:**

MOV AL, 5

MOV AH, 2

XCHG AL, AH **; AL = 2, AH = 5**

ARITHMETIC INSTRUCTIONS

- **<u>INC Instruction</u>** is used for incrementing an operand by one. It works on a single operand that can be either in a register or in memory.

- *Format:* **INC destination**
- The operand destination could be an 8-bit, 16-bit or 32-bit operand.

- *EXAMPLE:*
- INC DL ; Increments 8-bit register
- INC [count] ; Increments the count variable

- **<u>DEC instruction</u>** (decrementing an operand by one)

ADD AND SUB

- **ADD and SUB Instructions** are used for performing simple addition/subtraction of binary data in byte, word and double-word size, i.e., for adding or subtracting 8-bit, 16-bit or 32-bit operands respectively.
- *Format:* **ADD/SUB destination, source**
- The ADD/SUB instruction can take place between:
  - Register to register
  - Memory to register
  - Register to memory
  - Register to constant data
  - Memory to constant data

MUL/IMUL

- **MUL/IMUL Instruction** There are two instructions for multiplying binary data. The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag.

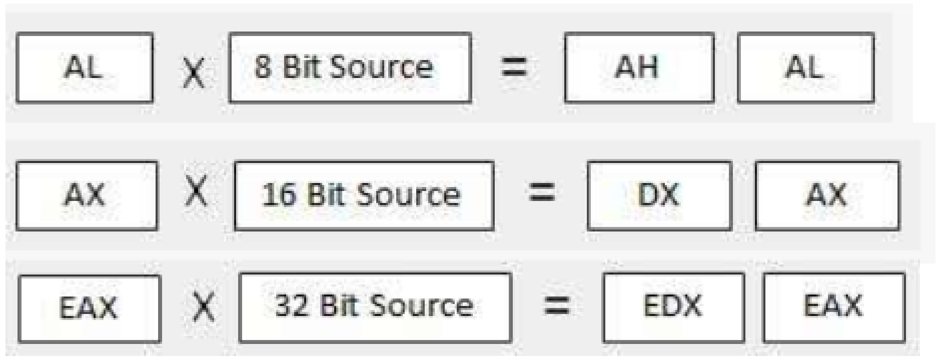
- *Format:* **MUL/IMUL multiplier**

- Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands.

MUL – CARRY FLAG

- Different cases of multiplied.
- **Example**

```
MOV AX,100
MOV BX,2000
MUL  BX           ;DX:AX = 00200000h, CF=1
```



The Carry flag indicates whether or not the upper half of the product contains significant digits

- **Example**

```
MOV AL, 5H
MOV BL, 10H
MUL  BL           ; AX = 0050H, CF = 0
```

(no overflow - the Carry flag is 0 because the upper half of AX is zero)

DIV/IDIV

- **DIV/IDIV Instruction** The division operation generates two elements - a **quotient** and a **remainder**.

- *Format:* **DIV/IDIV divisor**

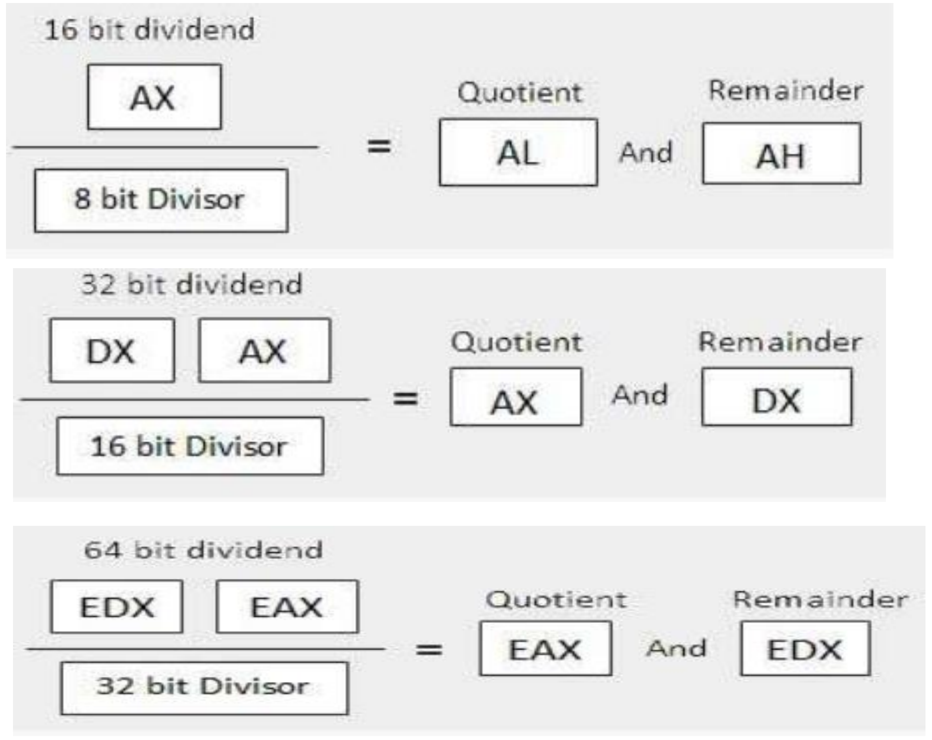
- The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands. Both instructions affect the Carry and Overflow flag.

- Three cases of division with different operand size.
  - When the divisor is 1 byte
  - When the divisor is double-word
  - When the divisor is double-word

Example

MOV DX,0 ; *clear dividend, high*
MOV AX,8003H ;*dividend, low* MOV
CX,100H ; *divisor*

DIV CX ; **AX = 0080h, DX = 3**



LOGICAL INSTRUCTIONS

- **AND** operand1, operand2
  - bitwise AND operation
- **OR** operand1, operand2
  - bitwise OR operation
- **XOR** operand1, operand2
  - bitwise XOR operation

- **TEST** operand1, operand2
  - works same as the AND operation, but unlike AND instruction, it does not change the first operand.
- **NOT** operand1
  - bitwise NOT operation

AND

- **<u>The AND instruction</u>** is used for supporting logical expressions by performing **bitwise** AND operation.

```
                    Operand1:   0101
                    Operand2:   0011
                    ------------------------
After AND ->        Operand1:   0001
```

- The AND operation can be used for clearing one or more bits.
  - For example, say, the BL register contains **0011 1010.** If you need to clear the high order bits to zero, you AND it with **0000 1111** (0FH)
  - <u>AND BL,   0FH   ;This sets BL to 0000 1010</u>
- Another example.
  - If you want to check whether a given number is odd or even, a simple test would be to check the least significant bit of the number. If this is 1, the number is odd, else the number is even.
  - <u>AND AL, 01H   *;ANDing with 0000 0001*</u>

** OR

- **The OR instruction** is used for supporting logical expressions by performing **bitwise** OR operation.
-

```
                    Operand1:    0101
                    Operand2:    0011
                    ------------------------------------
After OR ->         Operand1:    0111
```

- OR used to set one or more Bits, assume AL has a character.   **A= 01000001,   a= 01100001**
  - Convert Upper to Lower: set the 5th bit to 1
    - OR AL,   00100000b
  - Convert Lower to Upper: clear the 5th bit to 0
    - AND AL,   11011111b

CONDITIONS INSTRUCTIONS

- **CMP instruction** compares two operands. This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not. It does not disturb the destination or source operands. It is used along with the conditional jump instruction for decision making.

- *Format:* **CMP destination, source**

```
CMP DX, 00        ; Compare the DX value with zero
JE  L7            ; If yes, then jump to label L7
...
L7:
```

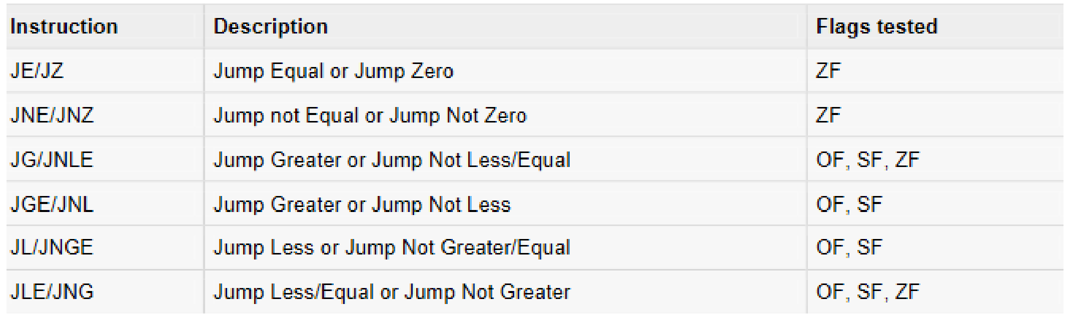
JMP

- **Unconditional Jump** this is performed by the **JMP** instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction.
- Transfer of control may be forward to execute a new set of instructions, or backward to re-execute the same steps.

- *Format:* **JMP label**

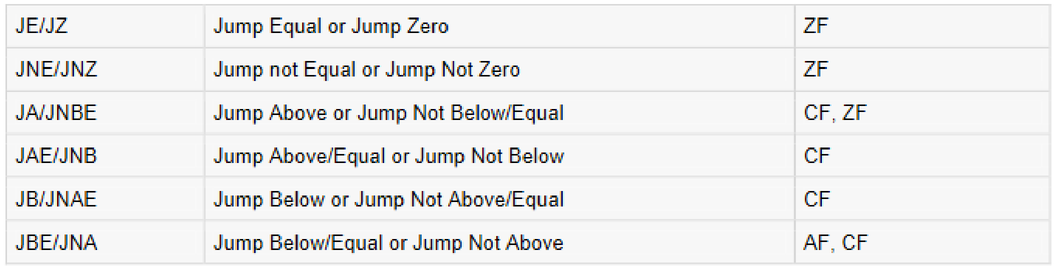
JXX

- **Conditional Jump** If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction.
- **Conditional jump instructions used on signed data**



Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

- **Conditional jump instructions used on unsigned data**



JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

LOOPS INSTRUCTION

- **LOOP instruction** assumes that the CX register contains the loop count. When the loop instruction is executed, the CX register is decremented and the control jumps to the target label, until the CX register value, i.e., the counter reaches the value zero.

- *Format:* **LOOP label**

```
mov CX,10
l1:
    <loop body>
loop l1
```

INTERRUPTS

- Interrupts can be seen as a number of functions. These functions make the programming much easier. We call such functions **software interrupts**.

  Interrupts are also triggered by different hardware, these are called **hardware interrupts**. Currently we are interested in **software interrupts** only.
- To make a **software interrupt** there is an **INT** instruction, it has very simple syntax:
- **INT value**
- Where **value** can be a number between 0 to 255 (or 0 to 0FFh). Each interrupt may have sub-functions
- To specify a sub-function **AH** register should be set before calling interrupt.

INT 21H

- Here are some of the most basic ones for console input and output here **INT 21H**.
- **Input a character**.

```
MOV     AH, 01h
INT     21h
```

- After the interrupt, AL contains the ASCII code of the input character. The character is echoed (displayed on the screen). **Use function code 8 instead of 1 for no echo.**

INT 21h/01h
INT 21h/02h
INT 21h/05h
INT 21h/06h
INT 21h/07h
INT 21h/09h
INT 21h/0Ah
INT 21h/0Bh

- **Output a character**.

```
MOV     DL, ...
MOV     AH, 02h
INT     21h
```

- Load the desired character into DL, then call the interrupt with function code 2 in AH.

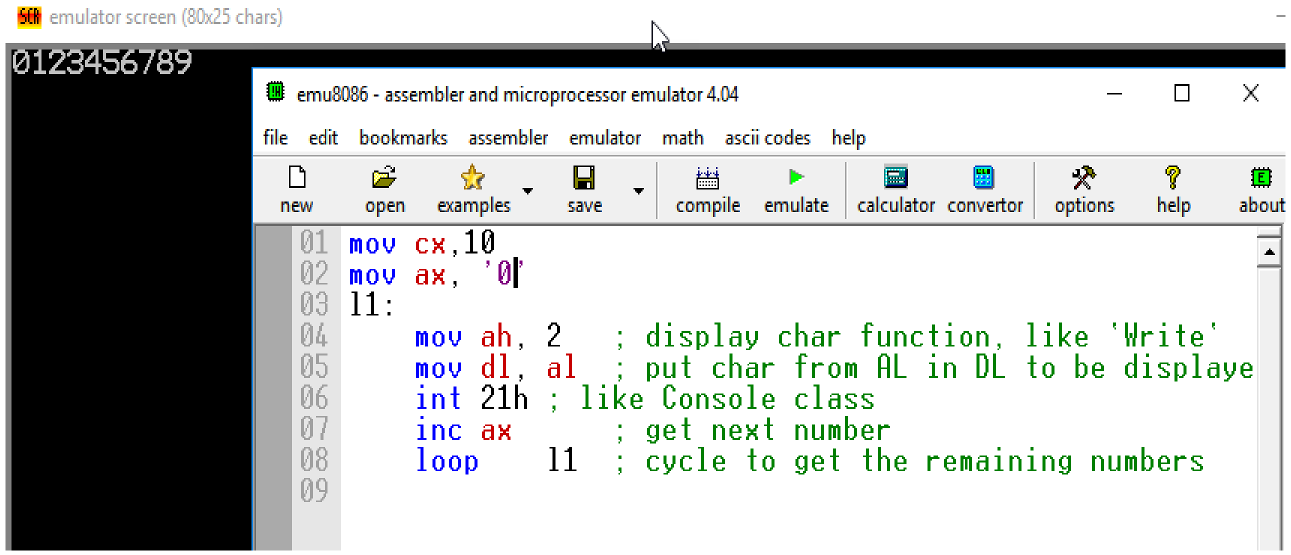
- **Output a string**.

```
MOV     DX, ...
MOV     AH, 09h
INT     21h
```

- Load the address of a '$'-terminated string into DX, then call the interrupt with function code 9 in AH.

LAB

- The following program prints the number 0 to 9 on the screen:

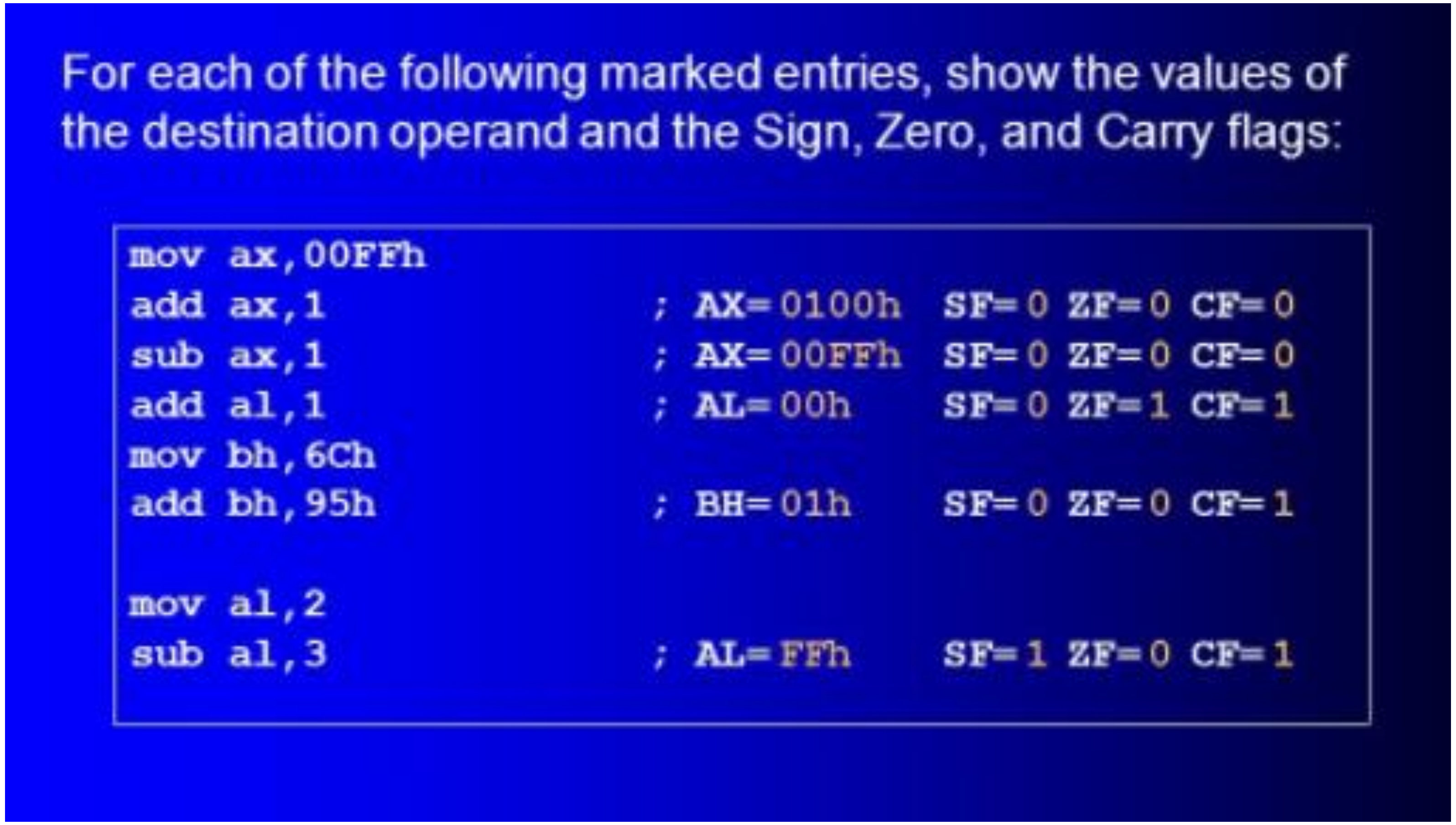

emulator screen (80x25 chars)
emu8086 - assembler and microprocessor emulator 4.04
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about

```
0123456789
```

```
mov cx,10
mov ax, '0'
l1:
    mov ah, 2    ; display char function, like 'Write'
    mov dl, al   ; put char from AL in DL to be displaye
    int 21h ; like Console class
    inc ax       ; get next number
    loop    l1   ; cycle to get the remaining numbers
```

- The output will be **0123456789**

SOLVED EXERCISE



For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

```
mov  ax,00FFh
add  ax,1                   ; AX=0100h   SF=0 ZF=0 CF=0
sub  ax,1                   ; AX=00FFh   SF=0 ZF=0 CF=0
add  al,1                   ; AL=00h     SF=0 ZF=1 CF=1
mov  bh,6Ch
add  bh,95h                 ; BH=01h     SF=0 ZF=0 CF=1

mov  al,2
sub  al,3                   ; AL=FFh     SF=1 ZF=0 CF=1
```

T

h

a

n

k

Y

o

u