

## Lecture 2

### 8086 Microprocessor

The Intel 8086 is a 16-bit microprocessor that is intended to be used as the CPU in a microprocessor. The term 16-bit means that its arithmetic logic unit, its internal registers, and most of its instructions are designed to work with 16-bit binary words. The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time. The 8086 has a 20 bit address bus, so it can address any one of  $2^{20}$  or 1048576, memory locations. Each of the 1048576 memory addresses of the 8086 represents a byte-wide location. 16-bit words will be stored in two consecutive memory locations. If the first byte of a word is at an even address, the 8086 can read the

entire word in one operation. If the first byte of the word is at an odd address, the 8086 will read the first byte with one bus operation and the second byte with another bus operation.

### Internal Architecture:

The 8086 microprocessor is internally divided into two separate functional units:

#### ① The Bus Interface Unit (BIU):

It provides the interface of 8086 to external memory and I/O devices via the system Bus. It performs various machine cycles such as memory read, I/O read etc. to transfer data between memory and I/O devices. BIU performs the following functions -

- It generates the 20 bit physical address for memory access.
- It fetches instructions from the memory
- It transfers data to and from the memory and I/O ports.

- Reads data from ports and memory and writes data to ports and memory.

In other words, BIU handles all transfers of data and addresses on the buses for the execution unit

\* The BIU's instruction queue is a First-In First-Out (FIFO) group of registers in which up to six bytes of instruction code are fetched from memory ahead of time.

\* The BIU contains a dedicated adder, which is used to produce the 20-bit address.

\* The bus control logic of the BIU generates all the bus control signals such as read and write signals for memory and I/O.

The following sections describe the functional parts of the BIU:

- Segment Registers:

The BIU has four 16-bit segment registers. These are the Code segment (CS) register, the Data Segment (DS) register, the Stack Segment (SS) register, and the Extra Segment (ES) register.

It holds the addresses of instructions and data in memory, which are used by the processor to access memory locations. It also contains 1 pointer register IP, which holds the address of the next instruction to be executed by the Execution Unit (EU).

✓ 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.

- The Queue:

To speed up program execution, the BIU fetches as many as six instruction bytes ahead of time from memory. The prefetched instruction bytes are held for the EU in a first-in-first-out group of registers called a queue. When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed. This is much faster than sending out a address to the system memory and waiting for memory to send back the next instruction byte or bytes. Fetching the next instruction while the current instruction executes is called pipelining.

### ■ : Instruction Pointer :

It is a 16-bit register used to hold the address of the next instruction to be executed.

The instruction pointer register holds the 16-bit address or offset of the next code byte within this code segment. The value contained in the IP is referred to as an offset because this value must be offset from the segment base address in CS to produce the required 20-bit physical address sent out by BIU.

(IP register always works together with CS segment register and it points currently executing instruction)

### ■ The Execution Unit :

The execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions.

The following sections describe the functional parts of the execution unit:

#### • Control circuitry, Instruction Decoder, And ALU :

The EU contains control circuitry which directs internal operations. A decoder in the EU translates instructions fetched from memory into a series of actions which the EU carries out. The EU has a 16-bit arithmetic logic unit which can add, subtract, ADD, OR, XOR, increment, decrement, complement or shift binary numbers.

A flag is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU

### • Flag Register :

It is a 16-bit register that behaves like a flip-flop ; it changes its status according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups - conditional flags & control flags.

Six of the nine flags are used to indicate some condition produced by an instruction

For example : (a flip-flop called the carry flag will be set to a 1 if the addition of two 16-bit binary numbers produces a carry out of the most significant bit position. If no carry out of the MSB is produced by the addition, the carry flag will be 0.)

The six conditional flags in this group are :

1. AF (Auxiliary carry flag) : It is used by BCD bit into the high nibble or a

borrow from the high nibble into the low 4 nibble of the low order 8-bit of a 16-bit number

2. CF (Carry Flag) is set if there is a carry from addition or borrow from subtraction.

3. OF (Overflow Flag) is set if there is an arithmetic overflow, that is if the size of the result exceeds the capacity of the destination location. An interrupt on overflow instructions is available which will generate an interrupt in this situation.

4. SF (Sign Flag) is set if the most significant bit of the result is one (negative) and is cleared to zero for non-negative result.

5. PF (Parity Flag) is set if the result has even parity ; PF is 0 for odd parity of the result.

which causes the 6086 internal interrupt after execution of each instruction.

6. ZF (Zero Flag) is set if the result is zero; ZF is zero for nonzero result.

The three remaining flags in the flag register are used to control certain operations of the processor. The six additional flags are set or reset by the EU on the basis of the results of some arithmetic or logic operation but the control flags are set or reset deliberately with specific instructions you put in the program.

These are:

- Trap Flag (TF): <sup>Trace Flag</sup> which is used for single stepping through a program.
- Interrupt Flag (IF): which is used to allow or prohibit the interruption of a program
- Direction Flag (DF): which is used with string instructions.

### General Purpose Registers:

The EU has eight general purpose registers labeled AH, AL, BH, BL, CH, CL, DH and DL. These registers can be used individually for temporary storage of 8-bit data. The AL register is also called the accumulator. It has some features that the other general purpose registers do not have.

These registers can be used individually to store 8-bit data and can be used in pairs to store 16-bit dat. The valid register pairs are AH <sup>and</sup> AL, BH and BL, CH and CL, DH and DL. It is referenced to AX, BX, CX, DX respectively.

#### • AX - The accumulator register (AH and AL)

1. Generates shortest machine code.
2. Arithmetic, logic, and data transfer
3. One number must be in AL or AX
4. Multiplication and Division
5. Input & Output.

It is mainly used to store operands for arithmetic operations.

• BX - The base address register (BH and BL)

It is used to store the starting base address of the memory area within the data segment.

• CX - the count register (CH and CL)

It is used in loop instruction to store the loop counter.

1. Iterative code segments using the Loop instruction.

2. Repetitive operations on strings with the REP command

3. Count (in CL) of bits to shift and rotate

• DX - the data register (DH and DL)

It is used to hold I/O port address for I/O instruction.

1. DX : AX concatenated into 32-bit register for some MUL and DIV operations.

2. Specifying ports in some IN and OUT operations.

• Stack Pointer Register (SP) and Stack Segment Register (SS) :

SP is a 16-bit register, which holds the address from the start of the segment to the memory location, where a word was most recently stored on the stack. The memory location where a word was most recently stored is called the top of stack.

The physical address for a stack read or a stack write is produced by adding the contents of the stack pointer register to the segment base address represented by the upper 16 bits of the base address in SS.

ss	5	0	0	0	0	→ Handwritten zero
sp +	F	F	E	0		
Physical Address	5	F	F	E	0	

(Top of stack)

Fig: Addition of ss and sp to produce physical address of the top of the stack.

#### • Pointers & Index registers in the EU:

In addition to SP, the EU contains a 16-bit Base pointer (BP) register, a 16-bit source index register (SI), a 16-bit destination index register (DI).

These 3 registers can be used for temporary storage of data just as the general purpose registers. (However their main use is to hold the 16-bit offset of a data word in one of the segments).

- SI : [Source Index Register]
  - ✓ 1. Can be used for pointer addressing of data
  - ✓ 2. Used as source in some string processing instructions
  - ✓ 3. Offset address relative to DS
- DI : [Destination Index register]
  - ✓ 1. Can be used for pointer addressing of data
  - ✓ 2. Used as destination in some string processing instructions
  - ✓ 3. Offset address relative to ES
- BP - [Base pointer] :
  - ✓ 1. Primarily used to access parameters passed via the stack
  - ✓ 2. Offset address relative to SS
- SP - [Stack Pointer] :
  - ✓ 1. Always points to top item on the stack
  - ✓ 2. Offset address relative to SS
  - ✓ 3. Always points to word (byte at even address)
  - ✓ 4. An empty stack will had  $SP = FFFEh$

## II Special Purpose Registers:

- IP - the instruction pointer:
  1. Always points to next instruction to be executed
  2. Offset address relative to CS
- ✓ IP register always work <sup>together</sup> with CS segment registers & it points to currently executing instruction

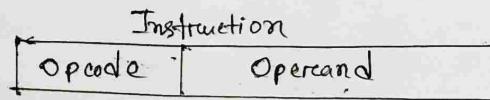
### Addressing Modes of 8086

#### 1. Immediate:

In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

Ex : MOV AX, 0005H

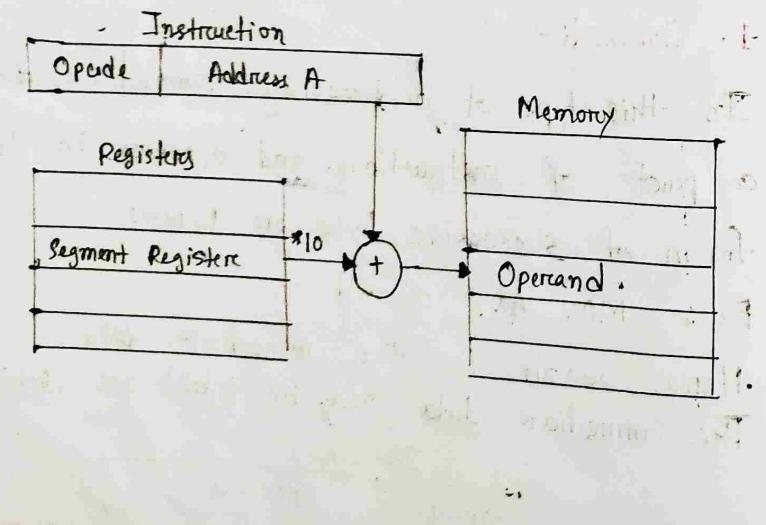
Here, 0005H is the immediate data.  
The immediate data may be 8-bit or 16-bit in size



### 2. Direct :

In the direct addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it.

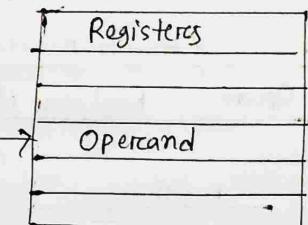
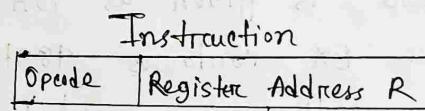
Ex: `MOV AX, [5000H]`. Hence, data resides in a memory location in the data segment whose effective address may be computed using `5000H` as the offset address and content of DS as segment address. The effective address here is `10H * DS + 5000H`



### 3. Register :

In register addressing mode, the data is stored in a register and it is referenced using the particular register. All the registers, except IP, may be used in this mode.

Example: `MOV BX, AX`; copies the contents of Suppose the 16-bit AX register into 16-bit CX register

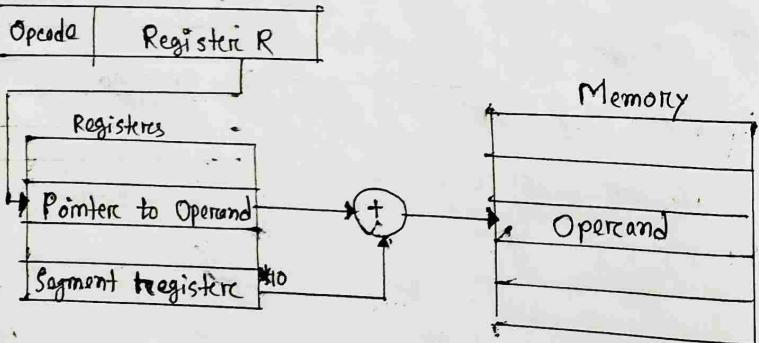


#### 4. Register Indirect :

In this addressing mode, the offset address of data is in either BX or SI or DI registers. The default regi. segment is either DS or ES

Example : MOV AX, [BX]. Here data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as  $10H * DS + [BX]$  [Suppose, the register BX contains 489H, then the contents 489H are moved to AX]

Instruction -

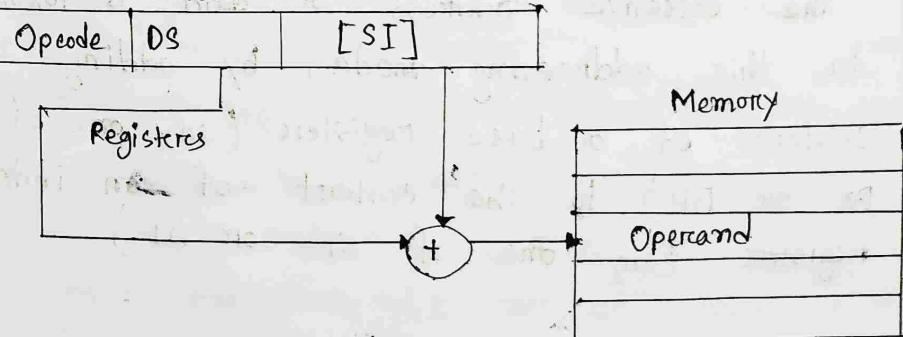


#### 5. Indexed :

In this addressing mode, offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers SI and DI respectively. This mode is a special case of the above discussed register indirect addressing mode.

Example : MOV AX, [SI]. Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as  $10H * DS + [SI]$

Instruction



### 6. Register Relative :

In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment. The example given before explains this mode.

Ex : MOV AX, 50H [BX] . Hence, effective address is given as  $10H * DS + 50H + [BX]$

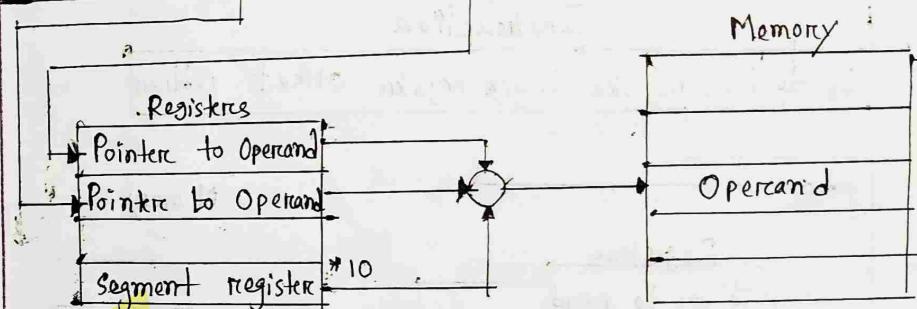
### 7. Based Indexed :

The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI.)

Ex : MOV AX, [BX][SI] . Here, BX is the base register and SI is the index register. The effective address is computed as  $10H * DS + [BX] + [SI]$ .

#### Instruction

Opcode	Base Register	Index Register
--------	---------------	----------------



### 8. Relative Based Indexed :

The effective address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the bases registers (BX or BP) and any one of the index registers, in a default segment.

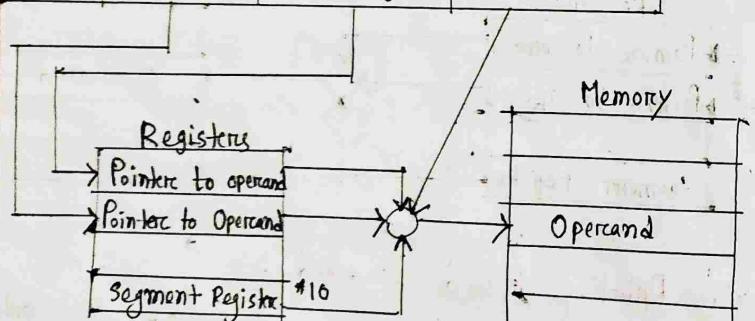
Physical Address = Base Address + offset

Ex: MOV AX, 50H [BX][SI]

Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as  $10H * DS + [BX] + [SI] + 50H$ .

#### Instruction

Opcode | Base Register | Index Register | Offset Address



\* How is the Physical address calculated?

= The physical address, which is 20-bits long is calculated using the segment and offset registers, each 16-bits long. The segment address is shifted left bit-wise four times and offset address is added to this to produce a 20-bit physical address.

Example:

Segment Address  $\rightarrow$  1005H

Offset Address  $\rightarrow$  5555H

$\Rightarrow$  Segment address  $\rightarrow$  1005H  $\rightarrow$  0001 0000 0000 0011

Shifted by 4 bit position  $\rightarrow$  0001 0000 0000 0101 0000

Offset Address  $\rightarrow$  + 0101 0101 0101 0101

Physical Address  $\rightarrow$  0001 0101 0101 1010 0101  
[155A5]      1      5      5      A      5

$$\begin{bmatrix} 0001 & 0000 & 0000 & 0101 & 0000 \\ 0101 & 0101 & 0101 & 0101 & 0101 \end{bmatrix}$$

0001 0101 0101 1010 0101

\* Why does an 8086 microprocessor has 1 MB memory?

→ 8086 has a concept memory segmentation.

It is a method where the whole memory is segmented (divided) into smaller parts called segments. These segments are:

- Code segment (CS)
- Stack " (SS)
- Data " (DS)
- Extra " (ES)

Each segment has a corresponding 16-bit segment Register which holds the Base Address of the segment. At any given time, 8086 can address  $16\text{ bit} \times 64\text{ KB} = 256\text{ KB} = 1\text{ MB}$

8086 has 20 bit address line, so the maximum value of address that can be addressed by 8086 is  $2^{20} = 1\text{ MB}$

So, 8086 can address the locations ranging between  $00000\text{ H}$  to  $FFFFF\text{ H}$ .

\*\* In order to access memory location, you can't pass 20-bit address directly to the processor. You need to tell the 16-bit address with respect to the segment. This 16-bit address with respect to the part of the memory bank is called the offset.

Questions:

1. ① If the code segment for an 8086 program starts at address  $70400\text{H}$ , what number will be in the CS register?

2) The CS register will hold  $7040\text{H}$

$$\begin{array}{r}
 7040 \\
 539 \\
 \hline
 7579
 \end{array}$$

(b) Assuming the same code segment base what physical address will a code byte be fetched from if the instruction pointer contains  $539\text{CH}$ ?

$\Rightarrow$  The next code byte fetched from the physical address =  $70400 + 539\text{CH}$   
 $= 7579\text{CH}$

Q8: What physical address is represented by:  
 (a)  $4370 : 561\text{EH}$   
 (b)  $7A32 : 0028\text{H}$

$\Rightarrow$  (a)  $4370 : 561\text{EH}$  represents  $43700\text{H} + 561\text{EH} = 48D1\text{EH}$

~~4370  
561  
48D1~~

$7A32 : 0028\text{H}$  represents  $7A320\text{H} + 0028\text{H} = 7A348\text{H}$

~~7A320  
0028  
7A348~~

(10) If the stack segment register contains  $3000\text{H}$  and the stack pointer register contains  $8434\text{H}$ , what is the physical address of the top of the stack?

$\Rightarrow SS = 3000\text{H}$  and  $SP = 8434\text{H}$   
 $TOS = 30000 + 8434\text{H} = 38434\text{H}$  or  
 $3000 : 8434$

(13) Describe the operation that an 8086 will perform when it executes each of the following instructions:

(a)  $MOV BX, 03FFH$ ; Load the number  $03FF$  into the BX register.

(b)  $MOV AL, 0DBH$ ; Load the number  $DBH$  into the AL register.

(c)  $MOV DH, CL$ ; The contents of the CL register are copied into the DH register. CL is unchanged.

①  $MOV BX, AX$ ; The contents of the AX register are copied into the BX register. AX is unchanged.

②  $ADD AX, BX$ ; the contents of the BX register are added to the contents of the AX register and the result is left in the AX register - BX is left unchanged.

14

① Load the number 7986H into the BP register  $\Rightarrow MOV BP, 7986H$

② Copy the BP register contents to the SP register  $\Rightarrow MOV SP, BP$

③ Copy the contents of the AX register to the DS register  $\Rightarrow MOV DS, AX$

④ Load the number F3H into the AL register  $\Rightarrow MOV AL, OF3H$

⑮ If the 8086 execution unit calculates an effective address of 19A3H and DS contains 7000H, what physical address will the BIU produce?

$$= EA = 19A3H \text{ and } DS = 7000H, \\ \text{address produced by the BIU} = 719A3H$$

⑯ If the data segment register, DS, contains 4000H. What physical address will the instruction  $MOV AL, [234BH]$  read?

$$= DS = 4000H, \text{ physical address for } [234BH] \\ 40000 + 234BH = 4234BH$$

⑰ If the 8086 data segment register contains 7000H. Write the instruction that will copy the contents of DL to address 79B2CH

$$= MOV [79B2CH], DL$$

⑧ Describe the difference between the instructions  $\text{MOV AX, } 2437\text{H}$  and  $\text{MOV AX, } [2437\text{H}]$

→  $\text{MOV AX, } 2437\text{H}$  = loads the **AX register** with the number  **$2437\text{H}$** .

$\text{MOV AX, } [2437\text{H}]$  = copies the contents of memory location  $\text{DS} + 2437\text{H}$  into AL and the contents of memory location ~~DS~~  $\text{DS} + 2437\text{H} + 1$  into AH

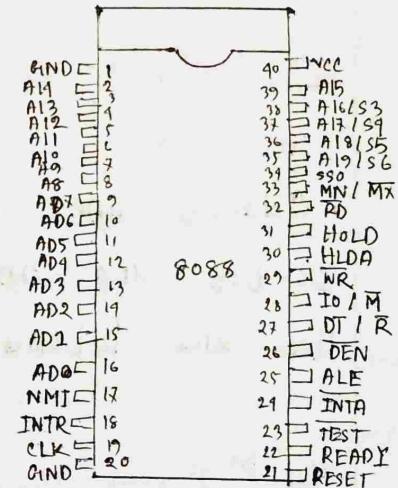
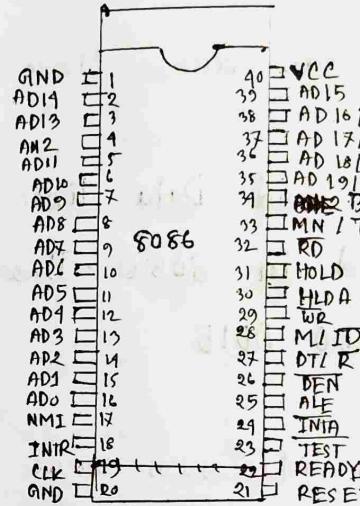
[i.e. loads the 16-bit AX register from two 8-bit memory locations]

## Lecture - 3

### 8086 / 8088 Microprocessor and its pin configuration

#### Basic Features :

- 8086 announced in 1978 ; 8086 is a 16-bit microprocessor with a 16-bit data bus.
- 8088 announced in 1979 ; 8088 is a 16-bit microprocessor with an 8-bit data bus.
- Both manufactured using High-performance Metal Oxide Semiconductor (MOS) technology
- Both contain about 29000 transistors.
- Both are packaged in 40 pin dual in-line package (DIP)



\* BHE has no meaning on the 8088 and has been eliminated.

■ Multiplex of Data and Address Lines in 8088.

- Address lines A<sub>0</sub>-A<sub>7</sub> and data lines D<sub>0</sub>-D<sub>7</sub> are multiplexed in 8088. These lines are labelled as A<sub>D0</sub>-A<sub>D7</sub>
- By multiplexed we mean that the same physical pin carries an address bit at one

time and the data bit another time.

### ■ Multiplex of Data and Address Lines in 8086 :

- Address lines A<sub>0</sub> - A<sub>15</sub> and Data lines D<sub>0</sub> - D<sub>15</sub> are multiplexed in 8086. These lines are labeled as ADO - AD<sub>15</sub>

### ■ Minimum-mode and Maximum-mode Systems :

8088 and 8086 microprocessors can be configured to work in either of the two modes : the minimum mode and the maximum mode.

#### • Minimum Mode :

- Pull MN/M<sub>X</sub> to logic 1 [33]
- Typically smaller systems & contains a single microprocessor.

→ Cheaper since all control signals for memory and I/O are generated by the Microprocessor.

#### • Maximum mode :

- Pull MN/M<sub>X</sub> logic 0 [33]
- Larger systems with more than one processor (Designed to be used when a coprop. coprocessor (8087) exists in the system)

Let us discuss the signals in detail :

#### • Power supply and frequency signals :

It uses 5V DC supply at V<sub>cc</sub>, pin 40 and uses ground at V<sub>ss</sub>, pin 1 and 20 for its operation.

• AND = AND (or)

#### • Clock Signal :

Clock signal is provided through Pin-19. It provides timing to the processor for operation. Its frequency is different for different versions, i.e. 5MHz, 8MHz, 10MHz.

### • Address / data bus :

AD<sub>0</sub> - AD<sub>15</sub>. These are 16 address/ data bus. AD<sub>0</sub> - AD<sub>7</sub> carries low order byte data and AD<sub>8</sub>-AD<sub>15</sub> carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

### • Address / status bus :

A<sub>16</sub> - A<sub>19</sub> / S<sub>3</sub> - S<sub>6</sub>. These are the 9 address/ status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

### • BHE/S<sub>7</sub>:

BHE stand for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D<sub>8</sub> - D<sub>15</sub>.

### • INTA :

It is an interrupt acknowledgement signal and is available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

### • MN / MX :

It stands for minimum/maximum and is available at pin 33. It indicate what mode the processor is to operate in; when it is high ; it works in the minimum mode & and vice - versa.

### • ALE :

It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation . This signal indicates the availability of a valid address on the address/ data lines.

### • DEN:

It stands for Data Enable and is available at pin 26 . It is used to enable Transceiver 8266 . The transceiver is device used to separate data from the address / data bus.

### • DT/R :

It stands for Data Transmit / Receive signal and is available at pin 27 . It decides the direction of data flow through the transceiver.

When it is high, data is transmitted out and vice-versa

- M/I<sub>O</sub>:

This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at 28

- WR:

It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/I<sub>O</sub> signal.

- HLDA : [output]

It is Hold Acknowledge output and is available at pin 30. This signal acknowledges the HOLD signal.

(Hold acknowledge is made high to indicate to the DMA controller that the processor has entered hold state and it can take control over the system bus for DMA operation)

- RQ / GT<sub>I</sub> and RQ / GT<sub>O</sub>:

Request / grant pins request / grant direct memory accesses (DMA) during maximum mode operation

- LOCK:

Lock output is used to lock peripherals off the system. Activated by using the LOCK prefix on any instruction

- S<sub>0</sub>, S<sub>1</sub>, S<sub>2</sub>:

These are the status signals that provide the status of operation which is used by the Bus Controller to generate memory & I/O control signals. These are available at pin 26, 27, 28

S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	Characteristics
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Inactive

### A17/S4, A16/S3 Address / Status:

- A17/S4, A16/S3 Address / Status:

		Function
0	0	Extra Segment Access
0	1	Stack Segment Access
1	0	Code Segment Access
1	1	Data Segment Access

### A18/S5:

The status of the interrupt enable flag bit is updated at the beginning of each cycle. The status of the flag is indicated through this pin.

### A19/S6:

When low, it indicates that 8086 is in control of the bus. During a "Hold acknowledge" clock period, the 8086 tri-states the S6 pin and thus allows another bus master to take control of the status bus.

### Q<sub>S1</sub> and Q<sub>S0</sub>:

These are queue status signals, and are available at pin 24 and 25. These signals provide the status of instruction queue.

QS1	Q <sub>S0</sub>	Characteristics
0	0	No Operation
0	1	First byte of opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

### INTR (Input) : [Hardware Interrupt Request pin]

✓ INTR is used to request a hardware interrupt.

✓ It is recognized by the processor only when IF = 1, otherwise it is ignored (STI instruction sets this flag bit)

✓ The request on this line can be disabled (or masked) by making IF = 0 (use instruction CLI)

✓ If INTR becomes high and IF = 1, the 8086 enters an interrupt acknowledge cycle (INTA)

becomes active) after the current instruction has completed execution

- NMI (input) : [Non-Maskable Interrupt Line]

- The Non Maskable interrupt, input is similar to INTR except that the NMI interrupt doesn't check to see if the IF flag bit is at logic 1.

- This interrupt can't be masked (or disabled) and no acknowledgement is required.

- It should be reserved for "catastrophic" events such as power failure or memory errors.

- Test (input) :

- The TEST pin is an input that is tested by the WAIT instruction

- If TEST is at logic 0, the WAIT instruction functions as a NOP.

- If TEST is at logic 1, then the WAIT instruction causes the 8086 to idle, until TEST input becomes a logic 0

- This pin is normally driven by the 8087 coprocessor (numeric coprocessor)

- Ready (input) :

- This input is used to insert wait states into processor Bus Cycle

- If the READY pin is placed at a logic 0 level, the microprocessor enters into Wait states and remains idle.

- If the REAY pin is placed at a logic 1 level, it has no effect on the operation of the processor.

- It is sampled at the end of the T2 clock pulse.

- Usually driven by a slow memory device

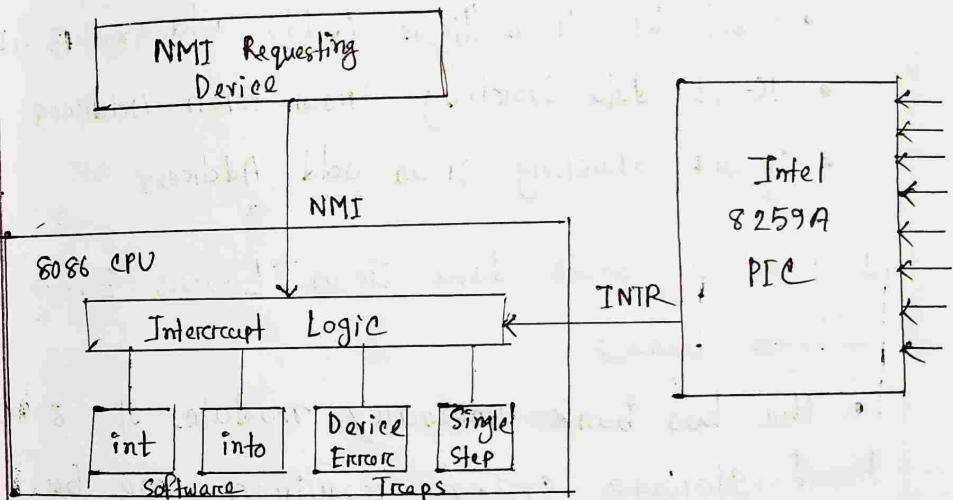
### • HOLD (input)

- The HOLD input is used by DMA controller to request a Direct Memory Access (DMA) operation
- If the HOLD signal is at logic 1, the microprocessor places its address, data and control bus at the high impedance state.
- If the HOLD pin is at logic 0, the microprocessor works normally.

8086 External Interrupt Connections:

NMI - Non Maskable Interrupt

INTR - Interrupt Request

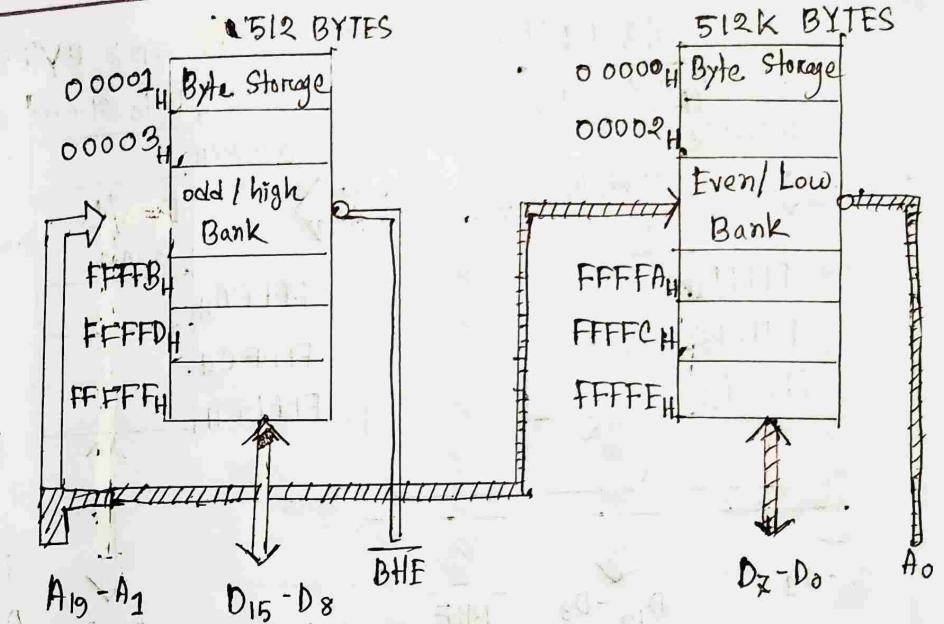


Data can be accessed from the memory  
in four different ways

- 8-bit data from Lower (Even) address Bank
- 8-bit data from Higher (Odd) address Bank
- 16-bit data starting from Even Address
- 16-bit starting from odd Address

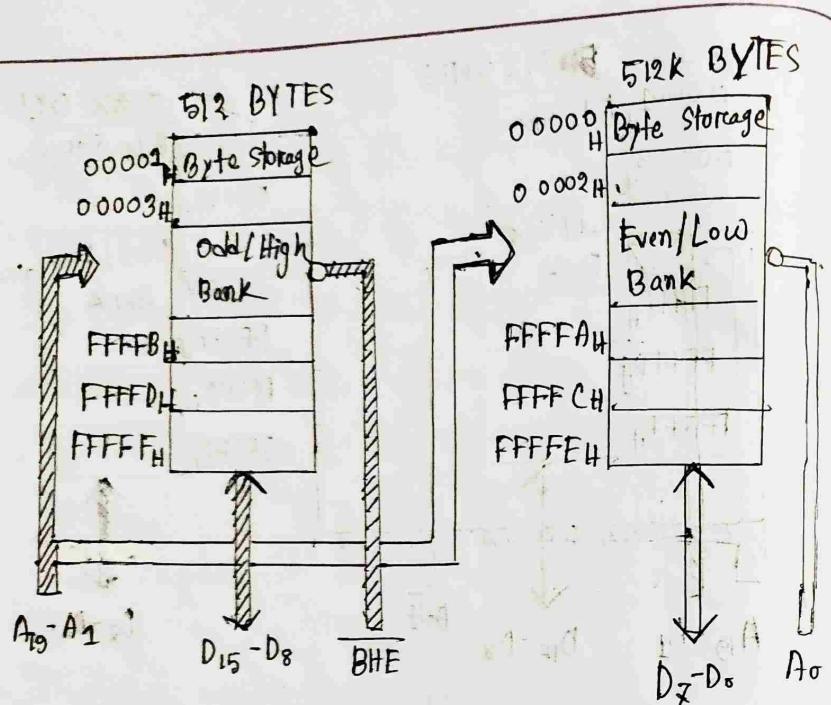
### 1. Accessing 8-bit data from Lower (Even) address Bank:

- The two bank memory module of 8086 based storage system requires one bus cycle to read/write a data byte.
- To access a Byte of data in Low-Bank, valid address is provided via address pins A<sub>1</sub> to A<sub>19</sub> together with A<sub>0</sub> = '0' and BHE = '1'



### 2. Accessing 8-bit data from Higher (Odd) address Bank:

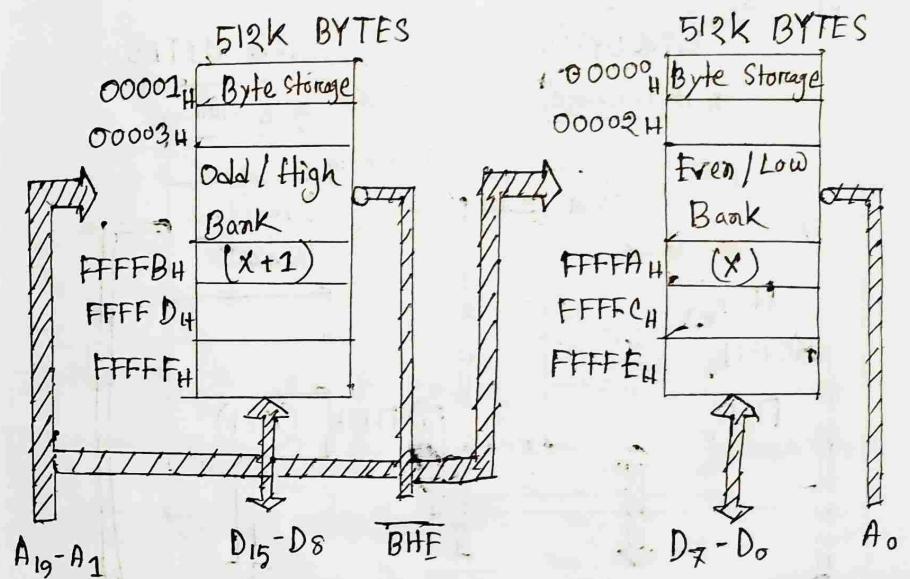
- Similarly, to access a Byte of data in High-bank, valid address in pins A<sub>1</sub> to A<sub>19</sub>, A<sub>0</sub> = '1' and BHE = '0' are required to access the data through D<sub>8</sub> to D<sub>15</sub> of the data-bus.
- These signals disable the Low bank and enable the High bank to transfer (in/out) data through D<sub>8</sub> to D<sub>15</sub> of the data-bus.



3. Accessing 16-bit data starting from Even Addresses:

- For even-addressed (aligned) words, only one bus-cycle is needed to access the word, as both low and high banks are activated at the same time, using  $A_0 = '0'$  and  $\overline{BHE} = '0'$

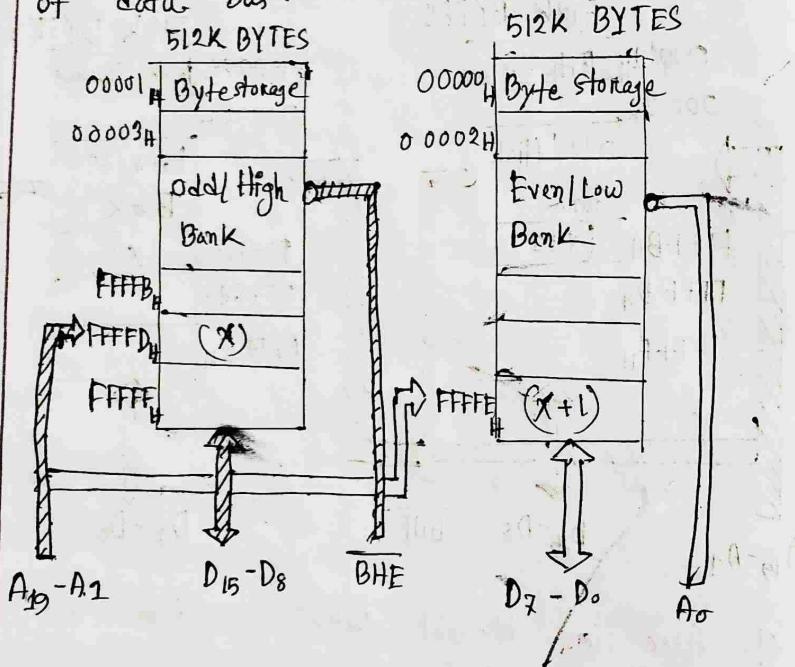
- Note that during this bus-cycle, all 16-bit data is transferred via  $D_0$  to  $D_{15}$  of the data bus.



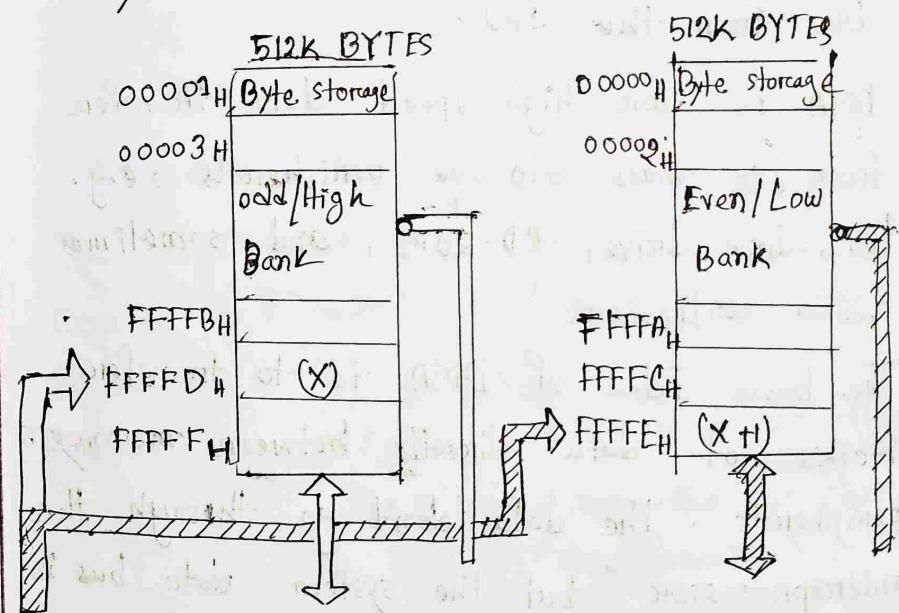
4. Accessing 16-bit data starting from Odd Address:

- For odd-addressed (unaligned) words (with odd P.A. of the LSB), two bus-cycles are required to access the Word-data.

- During the 1st bus-cycle, odd addressed LSB of the word is accessed from the High -memory -bank via D<sub>8</sub> to D<sub>15</sub> of data bus.



- Note that A<sub>0</sub> and BHE signals are reset (violet) accordingly to enable the required memory bank.



- During 2<sup>nd</sup> bus-cycle, P.A. is auto-incremented to access the even address MSB of the word from the Low Bank via D<sub>0</sub> to D<sub>7</sub>.

## ■ Direct Memory Access (DMA) :

DMA is a process in which an external device takes over the control of system bus from the CPU.

DMA is for high-speed data transfer from / to mass storage peripherals, e.g. hard-disk drive, CD-ROM, and sometimes video controllers.

The basic idea of DMA is to transfer blocks of data directly between memory & peripherals. The data don't go through the microprocessor but the system data bus is occupied.

"Normal" transfer of one data byte takes up to 29 clock cycles. The DMA transfer requires only 5 clock cycles.

Nowadays, DMA can transfer data as fast as 60 MB per second or more. The transfer rate is limited by the speed of memory and peripheral devices.

## ■ Basic process of DMA :

- For 8088/8086 in minimum mode :

The HOLD and HLDA pins are used to receive and acknowledge and the hold request respectively.

Normally the CPU has full control of the system bus. In a DMA operation, the peripheral takes over bus control temporarily.

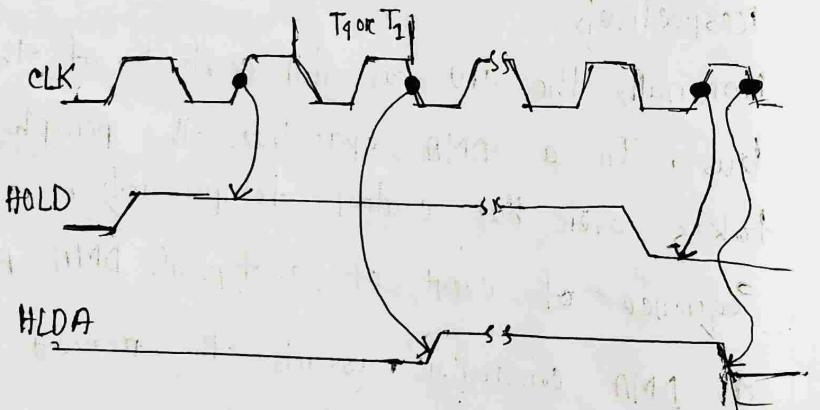
Sequence of events of a typical DMA process.

- ① DMA controller asserts the request on the HOLD pin.
- ② 8086 completes its current bus cycle and enters into a HOLD state.

③ 8086 grants the right of bus control by asserting a grant signal via the HLDA pin. 8086 pins (Address, Data, ~~Control~~  
~~3-state~~)

④ DMA operation starts.

⑤ Upon completion of the DMA operation, the DMA controller asserts low the request/grant pin again to relinquish bus control.



For 8088/8086 in maximum mode:

The RQ/GT<sub>1</sub> and RQ/GT<sub>0</sub> pins are used to issue DMA request and receive acknowledge signals.

Sequence of events of a typical DMA process:

① DMA controller asserts one of the request pins, e.g., RQ/GT<sub>1</sub> or RQ/GT<sub>0</sub> (RQ/GT<sub>0</sub> has higher priority)

② 8086 completes its current bus cycle and enters into a HOLD state.

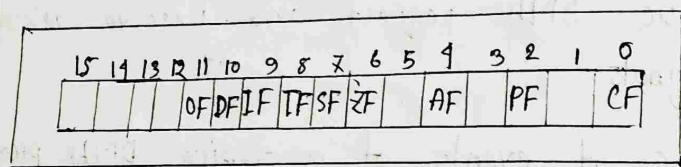
③ 8086 grants the right of bus control by asserting a grant signal via the same pin as the request signal

④ DMA operation starts

⑤ Upon completion of the DMA operation, the DMA controller asserts the request/grant pin again to relinquish bus control

## Lecture - 1

### Chapter 5 - The Processor Status and the Flags Registers



This figure shows the FLAGS Registers.

The status flags are located in bits 0, 2, 4, 6, 7 and 11 and the control flags are located in bits 8, 9 and 10. The other bits have no significance.

#### The status flags:

The processor uses the status flags to reflect the result of an operation. For example : If SUB AX, AX is executed,

the zero flag becomes 1, thereby indicating that a zero result was produced.

Let's get to know the status flags :

#### Carry Flag (CF) :

$CF = 1$  if there is a carry out from the most significant bit (msb) on addition, or there is a borrow into the msb on subtraction ; otherwise, it is 0. CF is also affected by shift and rotate instructions

#### Parity Flag (PF) :

$PF = 1$  if the low byte of a result has an even number of one bits (even parity). It is 0 if the low byte has odd parity. For example, if the result of a word addition is FFFEh, then the low byte contains 8 one bits, so  $PF = 0$

#### Auxiliary Carry Flag (AF) :

$AF = 1$  if there is a carry out from bit 3 on addition, or a borrow into bit 3 on subtraction. AF is used in binary-coded decimal (BCD) operations.



### • Zero Flag (ZF):

$ZF = 1$  for a zero result, and  $ZF = 0$  for a nonzero result.

### • Sign Flag (SF):

$SF = 1$  if the msb of a result is 1; it means the result is negative. If you are giving a signed interpretation  $SF = 0$  if the msb is 0.

### • Overflow Flag (OF):

$OF = 1$  if signed overflow occurred, otherwise it is 0. The meaning of overflow is dis-

### Overflow

Signed and unsigned overflows are independent phenomena. When we perform an arithmetic operation such as addition, there are four possible outcomes:

1. No overflow
2. Signed overflow only
3. Unsigned overflow only, and
4. both signed and unsigned overflows

### Unsigned Overflow

- On addition, unsigned overflow occurs when there is a carry out of the msb. This means that the correct answer is larger than the biggest unsigned number; that is, FFFFh for a word and FFh for a byte.

- On subtraction, unsigned overflow occurs when there is a borrow into the msb. This means the correct answer is smaller than 0.

- As an example of unsigned overflow but not signed overflow, suppose AX contains FFFFh, BX contains 0001h, and ADD AX, BX is executed. The binary result is -

$$\begin{array}{r}
 1111 1111 1111 1111 \\
 (+) 0000 0000 0000 0001 \\
 \hline
 1 0000 0000 0000 0000
 \end{array}$$

- If we are giving an unsigned interpretation, the correct ans is  $10000h = 65536$ , but this is out of range for a word operation. A 1 is carried out of the msb and the answer stored in AX,  $0000h$ , is wrong, so unsigned overflow occurred. But the stored answer is correct as a signed number, for  $FFFFh = -1$ ,  $0001h = 1$ , and  $FFFFh + 0001h = -1 + 1 = 0$ , so signed overflow did not occur.

### Signed Overflow

- On addition of numbers with the same sign, signed overflow occurs when the sum has a different sign. This happened in the preceding example when we were adding  $FFFh$  and  $FFFh$  (two positive numbers), but got  $FFFEh$  (a negative result).

- Subtraction of numbers with different signs is like adding numbers of the same sign. For example,  $A - (-B) = A+B$  and  $-A - (+B) = -A + -B$ . Signed overflow occurs if the result has a different sign than expected.
- In addition of numbers with different signs, overflow is impossible, because a sum like  $A + (-B)$  is really  $A - B$ , and because A and B are small enough to fit in the destination, so is  $A - B$ . For exactly the same reason, subtraction of numbers with the same sign cannot give overflow.
- Actually, the processor uses the following method to see the OF : If the carries into and out of the msb don't match — that is, there is a carry into the msb but no carry out, or if there is a

carry out but no carry in - then signed overflow has occurred, and OF is set to 1.

- As an example of signed but not unsigned overflow, suppose AX and BX both contain  $\text{FFFFh}$  and we execute

ADD AX, BX . The binary result is :

$$\begin{array}{r}
 0111\ 1111\ 1111\ 1111 \\
 + 0111\ 1111\ 1111\ 1111 \\
 \hline
 1111\ 1111\ 1111\ 1110 = \text{FFFFh}
 \end{array}$$

- The signed and unsigned decimal interpretation of  $\text{FFFFh}$  is  $32767$ . Thus for both signed and unsigned addition,  $\text{FFFFh} + \text{FFFFh} = 32767 + 32767 = 65534$ .

- This is out of range for signed numbers; the signed interpretation of the stored

answer FFFEH is 2. So, signed overflow occurred. However, the unsigned interpretation of FFFEH is 65534, which is the right answer, so there's no unsigned overflow.

### How Instructions Effect the Flags

#### Instruction

#### Affect flags

MOV / XCHG

none

ADD / SUB

all

INC / DEC

all except CF

NEG

all ( CF = 1 unless result is 0 )

OF = 1, if word operand is  $8000h$  or byte operand is  $80h$

- In general, each time processor executes an instruction, the flags are altered to reflect the result. But, those instructions don't affect any of the flags, affects only some of them, one may leave them undefined.

### Example

Example 5.1: ADD AX, BX ; where AX contains FFFFh, BX contains FFFFh.

### Solution:

$$\begin{array}{r} \text{FFFFh} (1111\ 1111\ 1111\ 1111) \\ (+) \text{FFFFh} (1111\ 1111\ 1111\ 1111) \\ \hline \text{1FFEh} (1111\ 1111\ 1111\ 1100) \end{array}$$

The result stored in AX is FFFEh =  
1111 1111 1111 1110

- SF = 1, because the msb is 1.
- PF = 0, because there are 7 (odd number) of 1 bits in the low byte of the result
- ZF = 0, because the result is nonzero.
- CF = 1, because there is a carry out of the msb on addition.
- OF = 0, because the sign of the stored result is the same as that of the numbers being added (as a binary addition, there is a carry into the msb and also a carry out)

### Example

Example 5.2: ADD AL, BL ; where AL contains 80h, BL contains 80h

$$\begin{array}{r} 80h \\ (+) 80h \\ \hline 100h \end{array}$$

The result stored in AL is 00h

- SF = 0, because the msb is 0
- PF = 1, because all the bits in the result are 0
- ZF = 1, because the result is 0
- CF = 1, because there is a carry out of the msb on addition.
- OF = 1, because the numbers being added are both negative, but the result is 0 (as a binary addition, there is no carry into the msb but there is a carry out)

- Example : 5.3 : SUB AX, BX where AX contains 8000h and BX contains 0001h

Solution :

$$\begin{array}{r}
 8000h \\
 (-) 0001h \\
 \hline
 \text{XFFFFh} = 0111\ 1111\ 1111\ 1111
 \end{array}$$

The result stored in AX is XFFFFh

- SF = 0, because the msb is 0
- PF = 1, because there are 8 (even number) one bits in the low byte of the result.
- ZF = 0, because the result is nonzero.
- CF = 0, because a smaller unsigned number is being subtracted from a larger one.

Now for OF, In a signed sense ; we are subtracting a positive number from a negative one, which is like adding two negative numbers. Because the result is positive (the wrong sign), OF = 1.

- Example : 5.4 : INC AL, where AL contains FFh

Solution :

$$\begin{array}{r}
 \text{FFh} \\
 + 1h \\
 \hline
 \text{100h}
 \end{array}$$

The result stored in AL is 00h.

• SF = 0, PF = 1, ZF = 1. Even though there is a carry out, CF is unaffected by INC. This means that if CF = 0, before the execution of the instruction, CF will still be 0 afterwards.

OF = 0 because numbers of unlike sign are being added (there is a carry into the msb and also a carry out).

- Example 5.5 : MOV AX, -5

Solution :

The result stored in AX is -5 = FFBh

None of the flags are affected by MOV.

## Lecture - 5

### Flow Control Instructions

Example : 5.6

NEG AX, where AX contains 8000h

Solution:

$$\begin{array}{r} 8000h = 1000 \ 0000 \ 0000 \ 0000 \\ \text{one's complement} = 0111 \ 1111 \ 1111 \ 1111 \\ \hline +1 \\ \hline 1000 \ 0000 \ 0000 \ 0000 = 8000h \end{array}$$

The result stored, in AX is 8000h

SF = 1, PF = 1, ZF = 0

CF = 1, because for NEG CF is always 1 unless the result is 0

OF = 1, because the result is 8000h; when a number is negated, we would expect a sign change, but because 8000h is its own two's complement, there is no sign change.

For assembly language programs to carry out useful tasks, there must be a way to make decisions and repeat sections of code.

The jump and loop instructions transfer control to another part of the program. This transfer can be unconditional or can depend on a particular combination of status flag settings.

#### Conditional Jumps:

JNZ is an example of a conditional jump instruction. The syntax is :

Jxxx destination\_label

If the condition for the jump is true, the next instruction to be executed is the one at destination\_label, which may precede or follow the jump instruction itself. If the condition is false, the instruction immediately following the jump is done next. For JNZ,

the condition is that the result of the previous operation is not zero.

### ■ Range of a Conditional Jump:

The structure of the machine code of a conditional jump requires that destination-label must precede the jump instruction by no more than 126 bytes, or follow it by no more than 127 bytes.

### ■ How the CPU implements a Conditional Jump:

To implement a conditional jump, the CPU looks at the FLAGS register. It reflects the result of the last thing the processor did. If the conditions for the jump are true; the CPU adjusts the IP to point to the destination label, so that the instruction at this label will be done next. If the jump

condition is false, then IP is not altered; this means that the next instruction in line will be done.

There are three categories:

- ① the signed jumps are used when a signed interpretation is being given to results.
- ② the unsigned jumps are used for an unsigned interpretation,
- ③ the single-flag jumps, which operate on settings of individual flags.

### ■ Table 6.1 - Conditional Jumps

#### • Signed Jumps

##### Symbol

JG / JNLE

JGE / JNL

##### Description

jump if greater than • jump if not less than or equal to

jump if greater than or equal to  
jump if not less than or equal to

##### Condition for Jump

ZF = 0 and SF = OF

SF = OF

JL / JNGE

jump if less than SF  $\neq$  OF  
jump if not greater than or equal

JLE / JNG

jump if less than or equal ZF = 1 or jump if not greater than SF  $\neq$  OF

### • Unsigned Conditional Jumps:

Symbol

Description

Condition for Jumps

JA / JNBE

jump if above.

CF = 0 and

jump if not below ZF = 0  
or ~~below~~ equal

JAE / JNB

Jump if above or equal CF = 0

jump if not below

Jump if below CF = 1

Jump if not above or equal

JBE / JNA

jump if equal CF = 1 or  
jump if not above ZF = 1

### • Single-Flag Jumps:

Symbol

Description

Condition for Jumps

JE / JZ

jump if equal

ZF = 1

JNE / JNZ

Jump if equal to zero

ZF = 0

JC

Jump if carry

CF = 1

JNC

Jump if no carry

CF = 0

JO

Jump if overflow

OF = 1

JNO

Jump if no overflow

OF = 0

JS

Jump if sign negative

SF = 1

JNS

Jump if nonnegative sign

SF = 0

JP / JPE

Jump if parity even

PF = 1

JNP / JPO

Jump if parity odd

PF = 0

## ■ The CMP Instruction:

The jump condition is often provided by the CMP (compare) instruction. It has the form:

CMP destination source

This instruction compares destination and source by computing destination contents minus source contents. The result is not stored, but the flags are affected. The operands of CMP may not both be memory locations. Destination may not be a constant.

[Note: CMP is just like SUB except that destination is not changed]

For example:

```
CMP AX, BX  
JG BELOW
```

where  $AX = \text{FFFFh}$ , and  $BX = 0001$ . The result of  $CMP AX, BX$  is  $\text{FFFFh} - 0001h = \text{FFEH}$ .

Here, the jump condition for JG is satisfied, because,  $ZF = SF = OF = 0$ , so control transfers to label BELOW.

\* ইবান্ত

যা কৃষ্ণ প্রস্তুত করতে হবে, সময় নষ্ট করে শায়িতানকে  
জুড়ে দেয়া যাবে না।

## ■ Signed Versus Unsigned Jumps:

Each of the signed jumps corresponds to an analogous unsigned jump, for example, the signed jump JG and the unsigned jump JA. Whether to use a signed or unsigned jump depends on the interpretation being given.

For example,

Suppose we are giving a signed interpretation. If  $AX = \text{FFFFh}$ ,  $BX = 8000h$ , and we execute

```
CMP AX, BX  
JA BELOW
```

then, even though  $\text{FFFFh} > 8000h$  in a signed sense, the program does not jump to BELOW.

The reason is that  $\text{FFFFh} < 8000h$  in an unsigned sense, and we are using the unsigned jump JA.

**Example 6.1:** Suppose AX and BX contain signed numbers. Write some code to put the biggest one in CX.

```
⇒ MOV CX, AX ; put AX in CX  
      CMP BX, CX ; is BX bigger?  
      JLE NEXT ; no, go on  
      MOV CX, BX ; yes, put BX in CX
```

### ④ The JMP Instruction:

The JMP (jump) instruction causes an unconditional transfer of control (unconditional jump). The syntax is:

JMP destination

where destination is usually a label in the same segment as the JMP itself.

JMP can be used to get around the range restriction of a conditional jump.

### ④ High-Level Language Structures:

#### • Branching structures:

In high-level languages, branching structures enable a program to take different paths, depending on conditions. In this section, we'll look at three structures.

##### 1. IF-THEN

##### 2. IF-THEN-ELSE

##### 3. Case

#### ① IF-THEN:

The IF-THEN structure may be expressed in pseudocode as follows:

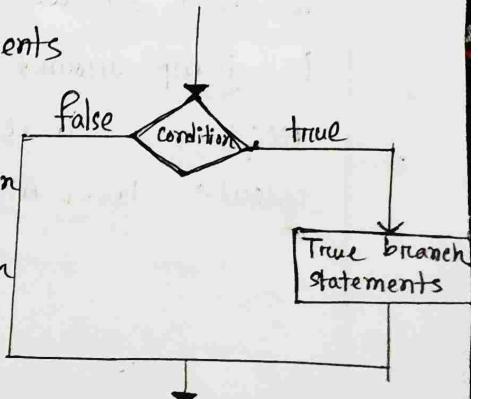
IF condition is true

THEN

execute true-branch statements

END-IF

\* The condition is an expression that is true or false. If it is true, the true-branch statements are executed. If it is false, nothing is done, and the program goes on to whatever follows.



### Example 6.2

Replace the numbers in AX by its absolute value.

⇒ A pseudocode Algorithm is :

#### Pseudocode

If  $AX < 0$

Then

Replace AX by -AX

END-IF

#### Code

```
CMP AX, 0 ;  $AX < 0?$ 
JNL END-IF ; no, exit
NEG AX ; yes, change sign
END-IF :
```

The condition  $AX < 0$  is expressed by `CMP AX, 0`. If  $AX$  is not less than  $0$ , there is nothing to do, so we use a `JNL` (jump if not less) to jump around the `NEG AX`. If condition  $AX < 0$  is true, the program goes on to execute `NEG AX`.

### ② IF-THEN-ELSE :

• Pseudocode in HLL:

IF condition is true

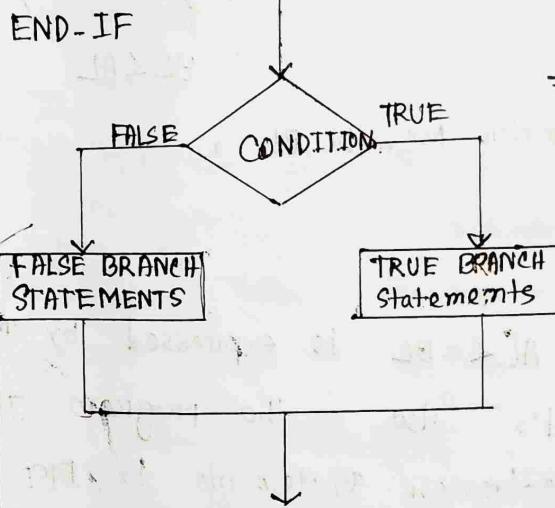
THEN

execute true-branch statements

ELSE

execute false-branch statements

END-IF



⇒ If condition is true, the true-branch statements are executed. If condition is false, the false-branch statements are done

Example 6.3 Suppose AL and BL contain extended ASCII characters. Display the one that comes first in the character sequence.

⇒

### Pseudocode

IF AL < BL

### Code

MOV AH,2 ; prepare to display  
 CMP AL,BL ; AL <= BL?  
 JNBE ELSE- ; no ,display character

THEN

display the character MOV DL,AL ; move character to be displayed  
 in AL  
 JMP DISPLAY ; go to display

ELSE

; BL < AL

display the character MOV DL,BL ,  
 in BL

END-IF

The condition  $AL <= BL$  is expressed by  $CMP AL,BL$ . If it's false , the program jumps around the true-branch statements to ELSE-. We use the unsigned jump JNBE (jump if not below or equal) , because we're comparing extended characters.

If  $AL <= BL$  is true , the true-branch statements are done . Note that JMP DISPLAY is needed to skip the false branch . This is different from the high-level language IF-THEN-ELSE , in which the false-branch statements are automatically skipped if the true-branch statements are done .

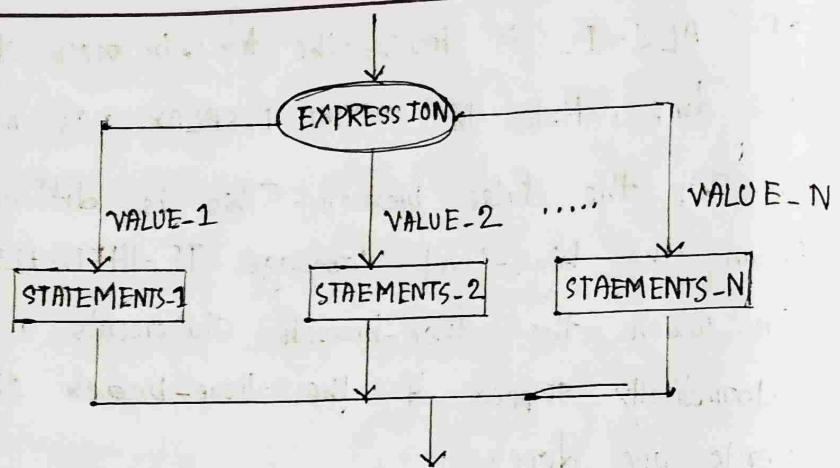
### 3 Case :

A case is a multiway branch structure that tests a register , variable , or expression for particular values or range of values . The general form is as follows :

case CASE expression :

values -1 : Statements -1  
 values -2 : statements -2  
 .....  
 values -n : statements -n

END-CASE



In this structure, expression is tested, if its value is a member of the set  $\{ \text{values-1}, \dots, \text{values-n} \}$ , then statements-1 are executed. We assume that sets  $\{ \text{values-1}, \dots, \text{values-n} \}$  are disjoint.

#### Example : 6.9

If AX contains a negative number, put -1 in BX ; if AX contains 0, put 0 in BX ; if AX contains a positive number, put 1 in BX.

$\Rightarrow$

CASE AX

$\leq 0$  : put -1 in BX  
 $= 0$  : put 0 in BX  
 $> 0$  : put 1 in BX

END-CASE

It can be coded as follows :

; Case AX

```

CMP AX, 0      ; test if ax
JL NEGATIVE   ; AX < 0
JE ZERO        ; AX = 0
JG POSITIVE    ; AX > 0
  
```

NEGATIVE :

```

MOV BX, -1     ; put -1 in BX
JMP END-CASE  ; and exit
  
```

ZERO :

```

MOV BX, 0      ; put 0 in BX
JMP END-CASE  ; and exit
  
```

POSITIVE :

```

MOV BX, 1      ; put 1 in BX
  
```

END-CASE :

- NOTE : only one CMP is needed, because jump instruction don't affect the flags.

Example : 6.5 If AL contains 1 or 3, display "0"  
"0" ; If AL contains 2 or 4, display "e"  
"e"

Solution:

CASE AL  
1,3 : display '0'  
2,4 : display 'e'

END-CASE

The code is :

```
; case AL
; 1,3;
    CMP AL, 1      ; AL = 1 ?
    JE ODD          ; yes, display '0'
    CMP AL, 3      ; AL = 3 ?
    JE ODD          ; yes, display '0'
; 2,4 :
    CMP AL, 2      ; AL = 2 ?
    JE EVEN         ; yes, display 'e'
    CMP AL, 4      ; AL = 4 ?
    JE EVEN         ; yes, display 'e'
    JMP END-CASE    ; not 1..4
ODD;
                    ; display '0'
```

```
MOV DL, '0'        ; get '0'
JMP DISPLAY        ; go to display
; display 'e'
```

EVEN :

```
MOV DL, 'e'        ; get 'e'
```

DISPLAY :

```
MOV AH, 2
INT 21H            ; display the charc
```

END-CASE ;

Branches with Compound Conditions :

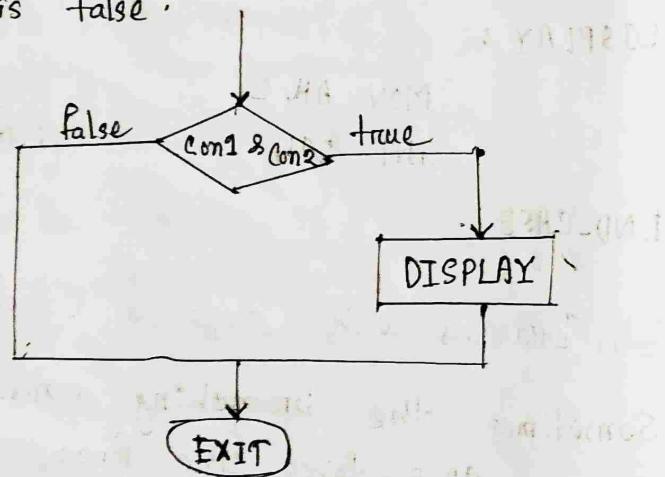
Sometimes the branching condition in an IF or CASE takes the form ..

→ Condition-1 and Condition-2

→ Condition-1 or Condition-2

## AND Conditions:

- An AND condition is true if and only if condition-1 and condition-2 are both true.
- If any condition is false, then the whole thing is false.



### Example 6.6

Read a character, and if it's an uppercase letter, display it.

⇒

### Pseudocode:

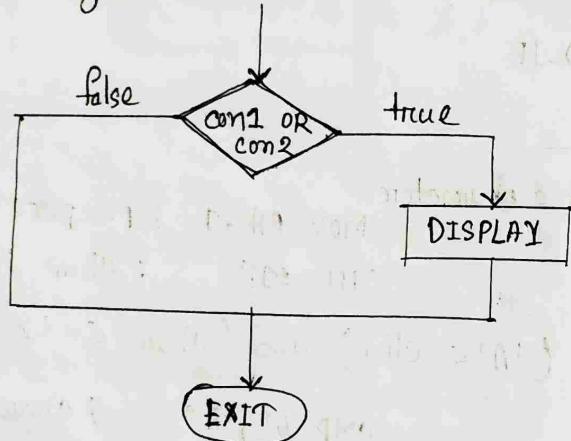
```
Read a character (into AL)
IF ('A' ≤ charc) and (charc ≤ 'Z')
    THEN
        display character
    END-IF
```

### Code

```
; read a character
MOV AH,1 ; prepare to read
INT 21H ; Charc in AL
; if ('A' ≤ charc) and (charc ≤ 'Z')
CMP AL,'A' ; charc >= 'A'?
JNGE END-IF ; no, exit
CMP AL,'Z' ; charc <= 'Z'?
JNLE END-IF ; no, exit
; then display charc
MOV DL,AL ; get charc
MOV AH,2 ; prepare to display
INT 21H ; display charc
END-IF :
```

## OR Conditions:

- If condition-1 OR condition-2 is true then an OR conditions is true.
- If both conditions are false, then the whole thing is false.



### Example 6.7

Read a character. If it's "y" or "Y", display it; otherwise, terminate the program.

$\Rightarrow$

## Pseudocode:

Read a character (into AL)

IF (characterc = 'y') OR (characterc = 'Y')

THEN  
display it

ELSE  
terminate the program

END-IF

## Code:

; read a character

MOV AH, 1

; prepare to read  
; charc in AL

INT 21H

; IF (characterc = 'y') OR (characterc = 'Y')

CMP AL, 'y'

; charc = 'y'?  
; yes, go to display it

JE THEN

CMP AL, 'Y'

; charc = 'Y'?  
; yes, go to display it

JE THEN

JMP ELSE -

; no, terminate

## THEN:

MOV AH, 2

; prepare to display

MOV DL, AL

; get charc

INT 21H

; display it

JMP END\_IF

; and exit

ELSE :

```
MOV AH, 4CH  
INT 21H ; DOS exit
```

END-IF :

### Lecture : 6

#### Looping Structure :

A loop is sequence of instructions that is repeated. The number of times to repeat may be known in advance, or it depend on condition. There are three types of loop. And they are:

1. For loop

2. While loop

3. Repeat

#### • For loop :

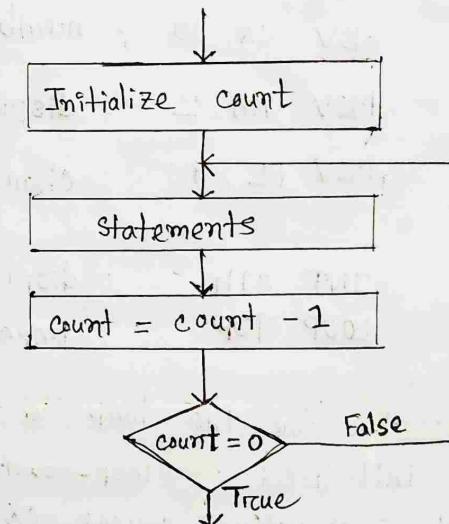
This is a loop structure in which the loop statements are repeated a number of times (a count controlled loop). In pseudo code,

For loop-count times Do

statements

END-FOR

figure of for. loop :



### Example : 6:8

Write a count-controlled loop to display a row of 80 stars.

### Solution :

```
FOR 80 times DO
    display '*'
```

END-FOR

The code is :

```
MOV CX, 80 ; number of stars to
             ; display
```

```
MOV AH, 2 ; display character function
```

```
MOV DL, '*' ; character to display
```

TOP :

```
INT 21h ; display a star
```

```
LOOP TOP ; repeat 80 times
```

\*\* The counter for the loop is the register CX which is initialized to loop-count. Execution of the LOOP instruction causes CX to be decremented automatically, and if CX is not 0, control transfers to destination-label. Must precede the LOOP instruction by no more than 126 bytes.

Using the instruction Loop, a FOR Loop can be implemented as follows:

```
TOP:           ; initialize CX to loop-count
              ; body of the loop
LOOP TOP.
```

- You may have noticed that a FOR loop, as implemented with a Loop instruction, is executed at least once. Actually, if CX contains 0 when the loop is entered the LOOP instruction causes CX to be decremented to FFFFh and the loop is then executed FFFFh = 65535 more times!

To prevent this, the instruction JCXZ (jump if CX is zero) may be used before the loop. Its syntax : JCXZ destination-label

If CX contains 0, control transfers to the destination label. So, a loop implemented as follows is bypassed if CX is 0 :

```
JCXZ SKIP
```

TOP :

```
; body of the loop
```

```
LOOP TOP
```

SKIP :

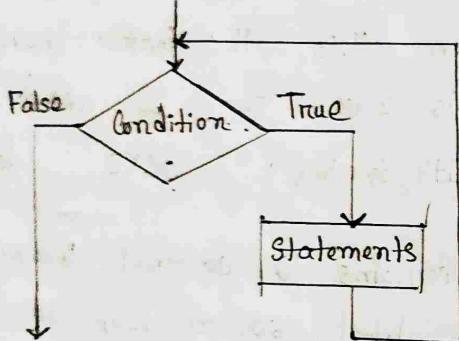
## ■ While loop :

This loop depends on a condition. In Pseudocode, WHILE condition DO

statements

END-WHILE

figure of while loop :



is

\* The condition checked at the top of the loop. If true, the statements is executed ; if false, the program goes on to whatever follows. It is possible that the condition will be false initially, in which case the loop body is not executed at all. The loop executes as long as the condition is true.

## Example : 6.9

Write some code to count the number of characters in an input line

⇒ initialize count to 0  
read a character  
WHILE character <> carriage-return DO  
    count = count + 1  
    read a character  
END-WHILE

The code is :

```
MOV DX,0 ; DX counts characters  
MOV AH,1 ; prepare to read  
INT 21H ; Characters in AL
```

WHILE :

```
CMP AL,0DH ; CR ?  
JE END WHILE ; yes, exit  
INC DX ; not CR, increment count  
INT 21H ; read a character  
JMP WHILE_ ; loop back
```

END WHILE

Note that, because a WHILE loop checks the terminating condition at the top of the loop, you must make sure that any variables involved in the condition are initialized before the loop is entered. So you read a character before entering the loop, and read another one at the bottom. The label WHILE- is used because WHILE is a reserve word.

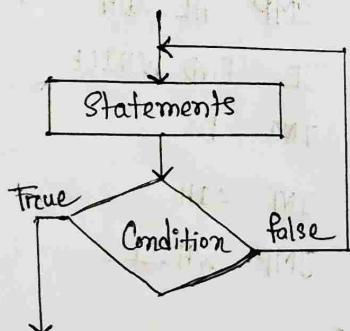
### REPEAT LOOP:

Another conditional loop is the REPEAT LOOP.

In Pseudocode :

```
REPEAT
    Statements
UNTIL condition
```

Figure of REPEAT LOOP :



In a REPEAT ... UNTIL loop, the statements are executed and then the condition is checked. If true, the loop terminates; if false, control branches to the top of the loop.

Example : 6.10

Write some code to read characters until a blank is read.

REPEAT

read a character  
UNTIL character is a blank

The code is :

MOV AH, 1 ; prepare to read

REPEAT :

INT 21H ; char in AL

; until

CMP AL, ' ' ; a blank ?

JNE REPEAT ; no, keep reading

## WHILE Versus REPEAT :

In many situations where a conditional loop is needed, use of a WHILE loop or a REPEAT loop is a matter of personal preference. The advantage of a WHILE is that the loop can be bypassed if the terminating condition is initially false, whereas the statements in a REPEAT must be done at least once. However, the code for a REPEAT loop is likely to be a little shorter because there is only a conditional jump at the end, but a WHILE loop has two jumps - a conditional jump at the top and a JMP at the bottom.

### Exercise :

1

(a)

```
CMP AX, 0
JGE END-IF
MOV BX, -1
```

END-IF

(b)

MOV

```
CMP AL, 0
JNL ELSE-
MOV AH, OFFh
JMP END-IF
```

ELSE\_:

MOV AH, 0

END-IF

```
CMP AL, 0
JL THEN-
MOV AH, 0
JMP END-IF
THEN_: MOV AH, OFFh
END-IF :
```

(c)

```
CMP DL, 'A'
JL END-IF
```

```
CMP DL, 'Z'
```

```
CMP DL, '-'
```

```
JG END-IF
```

```
MOV AH, 2
```

```
INT 21H
```

END-IF :

.; display DL

```
CMP AX, BX
JGE END-IF
CMP BX, CX
JGE ELSE-
MOV AX, 0
JMP END-IF
ELSE_: MOV BX, 0
END-IF :
```

## Lecture:7

### Logic, Shift and Rotation

#### Shift

Shift operation move the bits in a pattern, changing the position of the bits. They can move bits to the left or to the right. We can divide shift operation into two categories:

1. Logical Shift
2. Arithmetic Shift

#### Logical Shift :

A logical shift moves the bits within the cell one position to the right or to the left.

- In a logical right shift, the least significant bit LSB is discarded and the most significant bit MSB is assigned 0.
- In a logical left shift, the LSB is assigned to 0 and the msb is discarded.

**SHL:** Syntax : SHL Register, Bits to be shifted

Example : SHL AX, 2

- A shift instruction will have an operand that specifies how many times the one position shift is applied.

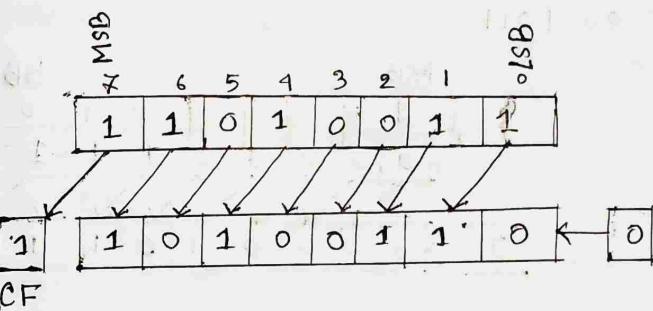
There are two kinds of Logical shift :

- Shift Left
- Shift Right

#### Left Shift: (SHL)

- A shift left logical of one position moves each bit to the left by one. The low-order bit LSB is replaced by a zero bit and the high-order bit MSB move to CF (carry flag).
- Shifting by two positions is the same as performing a one-position shift two times. Shifting by zero positions leaves the pattern unchanged. Shifting an N-bit pattern left by N or more positions changes all of the bits to zero.

- The picture shows the operation performed on eight bits. The original pattern is 11010011. The resulting pattern is 10100110.



#### Right Shift: (SHR)

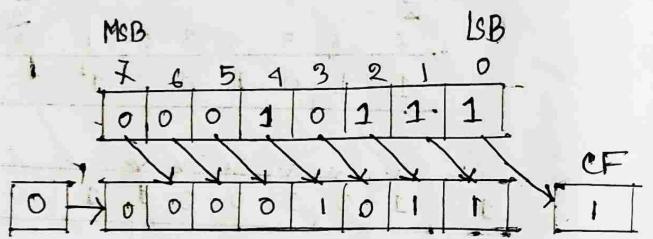
- A shift right logical of one position moves each bit to the right by one. The high-order bit MSB is replaced by a zero bit and the low-order bit LSB move to CF (carry Flag).
- Shifting by two positions is the same as performing a one-position shift two times. Shifting by zero positions leaves the pattern unchanged. Shifting an N-bit pattern right by N or more positions changes all of the bits to zero.

**SHR** → Syntax : SHR Register, Bits to be shifted

Example : SHR AX, 2

- The picture shows the operation performed on eight bits. The original pattern is 00010111. The resulting pattern is

00001011



### Arithmetic Shift :

Arithmetic shift operations assume that the bit pattern is a signed integer in two's complement format. Arithmetic right shift is used to divide an integer by two, while arithmetic left shift is used to multiply an integer by two.

There are two kinds of arithmetic shift:

1. Left arithmetic shift
2. Right arithmetic shift

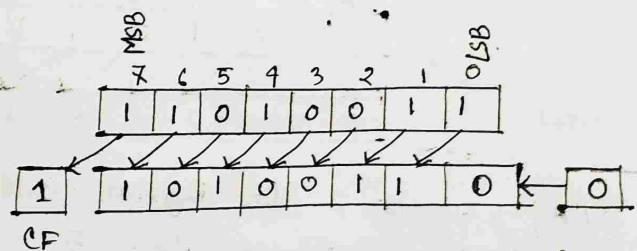
जटाते

### Left Arithmetic Shift : (SAL)

- An shift arithmetic left is the same as a logical left shift.
- A shift arithmetic left of one position moves each bit to the left by one. The low-order bit LSB is replaced by a zero bit and the high-order bit MSB move to CF (Carry Flag).

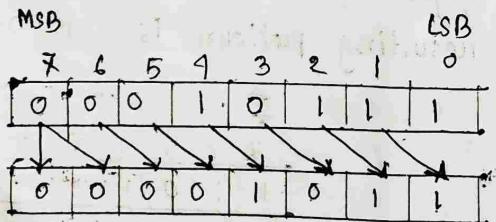
SAL Register, Bits to be shifted  
syntax : SAL CL, 2  
Example : SAL CL, 2

- A left arithmetic shift by n is equivalent to multiplying by  $2^n$ . In 2's complement, positive or negative, a logical left shift, is equivalent to multiplication by two.
- The picture shows the operation performed on eight bits. The original pattern is 11010011. The resulting pattern is 10100110.



## Shift Arithmetic Right : (SAR)

- A shift arithmetic right of one position moves each bit to the right by one. The high-order bit MSB is replaced by sign bit and the low-order bit LSB move to CF (Carry Flag).
- A shift arithmetic right is equivalent to integer division by two.
- In 2's complement, positive or negative division by two is accomplished via an shift arithmetic right.
- The picture shows the operation performed on eight bits. This original pattern is 00010111. The resulting pattern is 00001011.



SAR → Syntax : SAR Register, Bits to be shifted

Example : SAR BX, 5

## Rotation :

ROTATE is a logical operation.

1-byte instruction. This instruction does not require any operand after the opcode. It operates the content of accumulator and the result is also stored in the accumulator. The Rotate instruction is used to rotate the bits of accumulator.

There are 4 types of the ROTATE instruction:

1. ROL : Rotate Left

2. ROR : Rotate Right

3. RCL : Rotate Carry Left

4. RCR : Rotate Carry Right

• ROL : Rotate Left :

ROL (rotate) shifts each bit to the left.

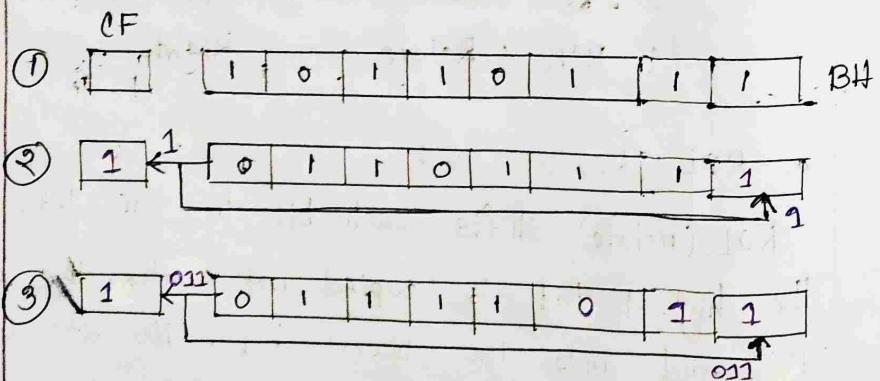
The highest bit is copied into both the carry flag and into the lowest bit. No bits are lost. ROL is used for unsigned data.

[label:]	ROL	register/memory, CL/immediate
----------	-----	-------------------------------



Below are instances of the ROL instruction:

Instruction	Comment	Binary	CF
① MOV BL, 10110111B	; Initialize BH	10110111	1
② ROL BL, 01	; Rotate Left 1	01101111	1
③ MOV CL, 03	; Set rotate value		-
④ ROL BL, CL	; Rotate left three more	01110111	1



- ROR : Rotate Right :

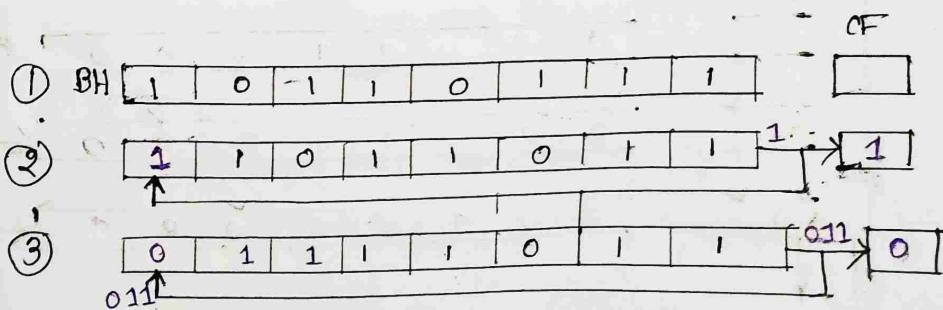
ROr (rotate right) shifts each bit to the right. The lowest bit is copied into both the Carry flag and into the highest bit. No bits are lost. ROR is for unsigned data.

[label:]	ROr	register/memory, CL/immediate
----------	-----	-------------------------------



- A few examples on ROR :

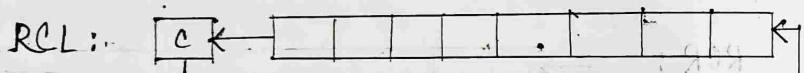
Instruction	Comment	Binary	CF
① MOV BL, 10110111B	; Initialize BH	10110111	
② ROR BL, 01	; Rotate right 1	11011011	1
③ MOV CL, 03	; Set rotate value		-
④ ROR BL, CL	; Rotate right three more	01110111	0



- RCL : Rotate Carry Left :

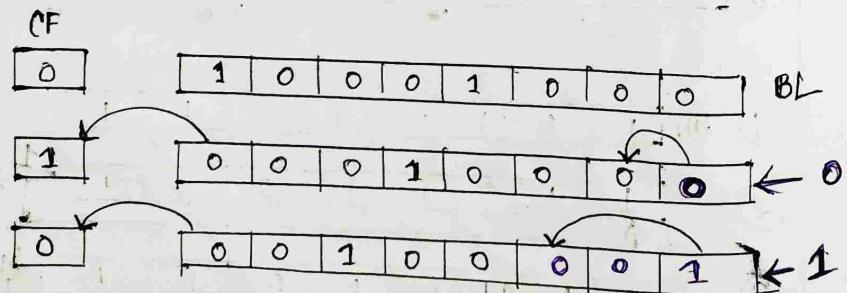
RCL (rotate carry left) shifts each bit to the left. Copies the carry flag to the least significant bit. Copies the most significant bit to the carry flag. RCL is for signed data.

[Label:] RCL register/memory, CL/immediate



Example:

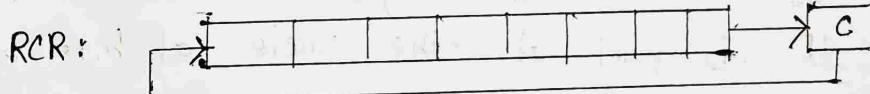
CLC	; CF = 0
MOV BL, 88H	; CF, BL = 0 10001000 B
RCL BL, 1	; CF, BL = 1 00010000 B
RCL BL, 1	; CF, BL = 0 00100001 b



- RCR : Rotate Carry right :

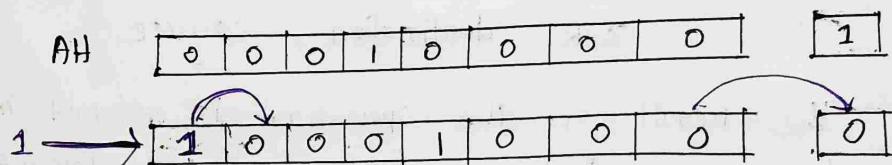
RCR ( Rotate Carry right) shifts each bit to the right. Copies the carry flag to the most significant bit. Copies the least significant bit to the carry flag.

[Label:] RCR register/memory, CL/immediate



• Example:

STC	; CF = 1
MOV AH, 10H	; CF, AH = 00010000 1
RCR AH, 1	; CF, AH = 10001000 0



- Difference:

- The difference between ROR and RCR is only the way of operation. In RCR, every bit that is rotated will enter the carry flag before entering the leftmost bit.
- RCL works like just like ROL except that CF is part of the circle of bits being rotated.

### AND, OR and XOR Instructions :

The AND, OR and XOR instructions perform the named logic operations. The formats are:

- AND destination, source
- OR destination, source
- XOR destination, source

The result of the operation is stored in the destination, which must be a register, or

memory location. The source may be a constant, register or memory location. However, memory-to-memory operations are not allowed.

#### Effect on flags:

SF, ZF, PF reflect the result

AF is undefined

CF, OF = 0

One use of AND, OR and XOR is to selectively modify the bits in the destination. To do this, we construct a source bit pattern known as a mask.

The mask bits are chosen so that the corresponding destination bits are modified in the desired manner when the instruction is executed.

To choose the mask bits, we make use of the following properties of AND, OR, XOR.

If  $b$  represents a bit (0 or 1) :

$$\begin{aligned} b \text{ AND } 1 &= b, \quad b \text{ OR } 0 = b, \quad b \text{ XOR } 0 = b \\ b \text{ AND } 0 &= 0, \quad b \text{ OR } 1 = 1, \quad b \text{ XOR } 1 = \text{complement of } b \end{aligned}$$

From these, we may conclude that :

1. The AND instruction can be used to clear specific destination bits while preserving the others. A 0 mask bit clears the corresponding destination bit; a 1 mask bit preserves the corresponding destination bit.
2. The OR instruction can be used to set specific destination bits while preserving the others. A 1 mask bit sets the corresponding destination bit; a 0 mask bit preserves the corresponding destination bit.
3. The XOR instruction can be used to complement specific destination bits while preserving

the others. A 1 mask bit complements the corresponding destination bit; a 0 mask bit preserves the corresponding destination bit.

Example : 7.2