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 perfected 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 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, AND, 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 thy 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.

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 deli - berately with specific instructions you put in the program.

These are :

- Trap Flag (TF) / Trace Flag : which is used for single stepping through a program. which causes the 8086 to generate an internal interrupt after execution of each instruction.
- 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 register 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 -percpose registers donot 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 and AL, BH and BL, CH and CL, DH and DL. It is referred to AX, BX, CX, DX respectively register

- 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 ports 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.

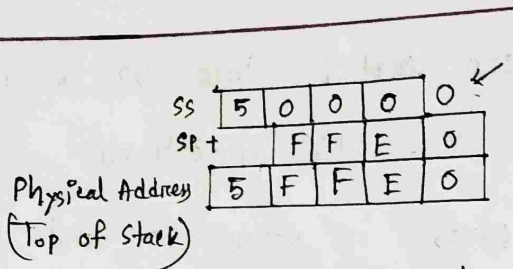


Fig: Addition of ss and SP to produce physical address of the top of the stack.

- **Pointer & 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 ~~regi~~ index register (DI).

These 3 register 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

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 together with CS segment register & it points to currently executing instruction

Addressing Modes of 8086

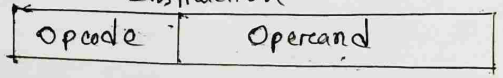
1. Immediate :

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.

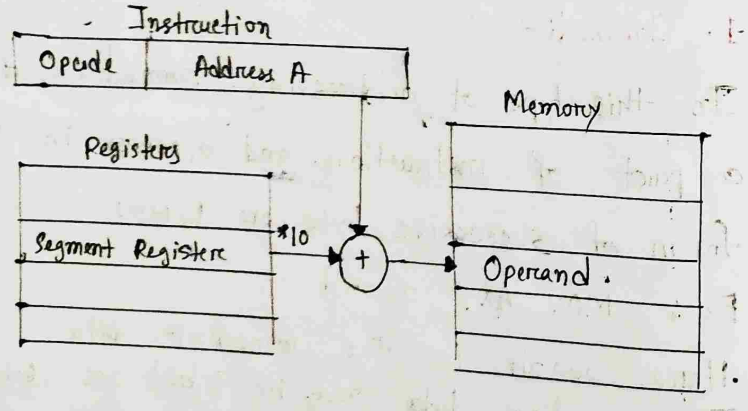
The immediate data may be 8-bit or 16-bit in size



2. Direct :

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

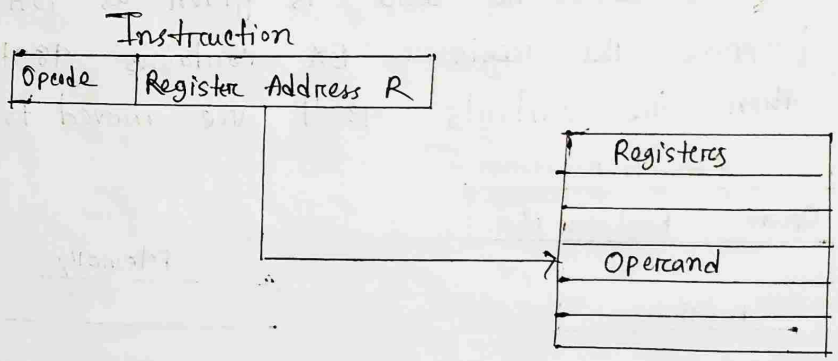
Ex: MOV AX, [5000H]. Here, data resides in a memory location in the data segment whose effective address may be computed using 5000H as the offset address and content of DS as segment address. The effective address here is 10H * DS + 5000H



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.

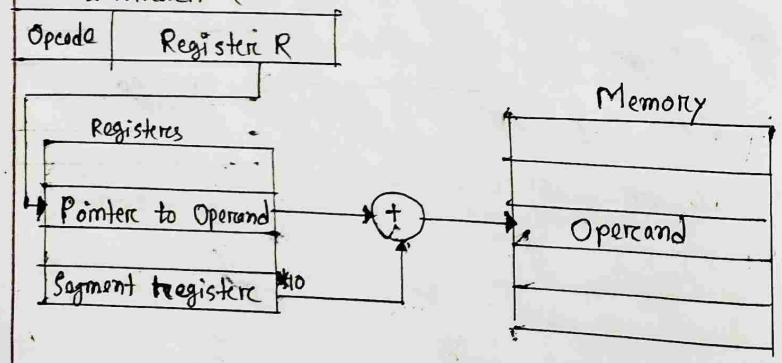
Example : MOV, BX, AX ; copies the contents of Suppose the 16-bit AX register into 16-bit CX register



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

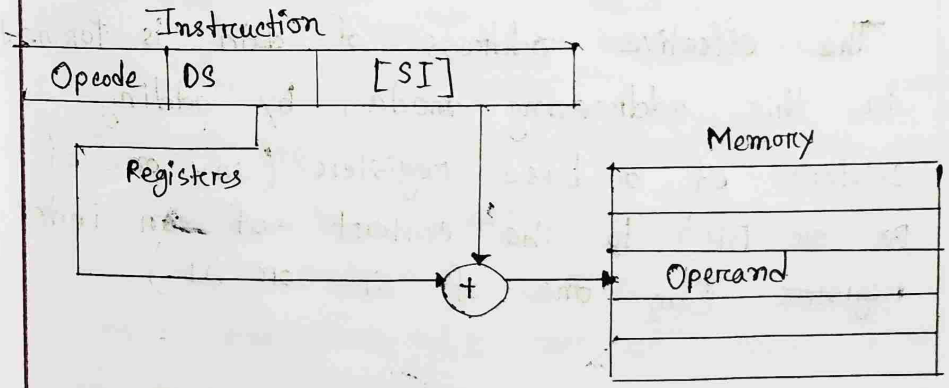
Example : MOV AX, [BX]. Here data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H*DS+[BX] [Suppose, the register BX contains 489H, then the contents 489H are moved to AX]



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]



6. Register Relative :

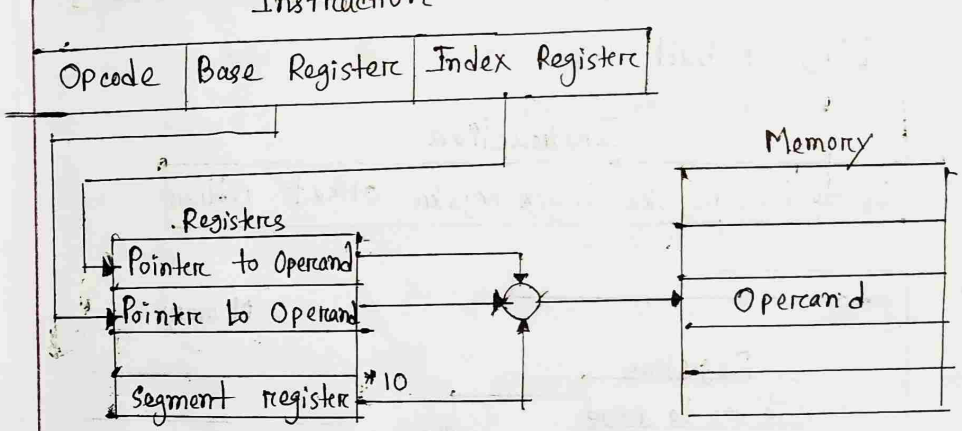
In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment. The example given before explains this mode.

Ex: MOV AX, 50H [BX] . Here, 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 10 H * DS + [BX] + [SI] .

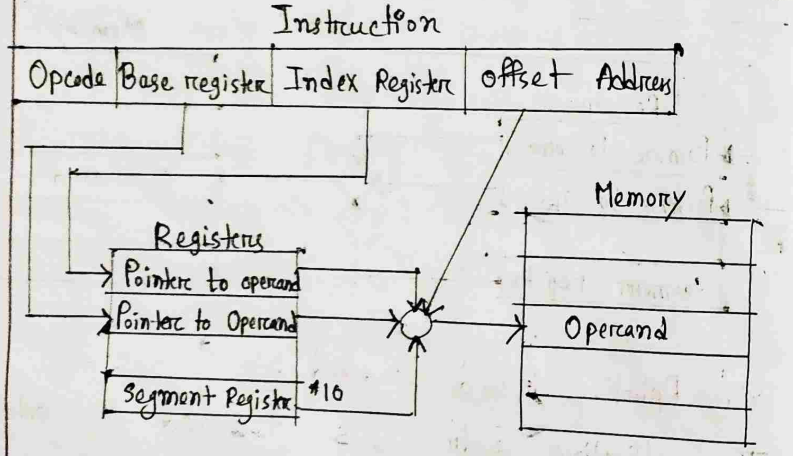


8. Relative Based Indexed :

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.

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



Physical Address = Base Address + offset

* 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 → 1005 H

offset Address → 5555 H

=> Segment address → 1005H → 0001 0000 0000 0101

Shifted by 4 bit position → 0001 0000 0000 0101 0000

offset Address → + 0101 0101 0101 0101

Physical Address → 0001 0101 0101 1010 0101

[155A5] 1 5 5 A 5

0001 0000 0000 0101 0000

0101 0101 0101 0101

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 bit × 64kB = 256 kB = 1MB

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

So, 8086 can address the locations ranging between 00000 H to FFFFF 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 :

7. ⓐ If the code segment for an 8086 program starts at address 7040 0H, what number will be in the CS register ?

⇒ The CS register will hold 7040H

70400
539
7579

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

⇒ The next code byte fetched from the physical address = 70400 + 539CH

= 7579CH

[8] What physical address is represented by · (a) 4370 : 561 EH

(b) 7A32 : 0028 H

⇒ (a) 4370 : 561 EH represents 43700H + 561 EH = 48D1EH

4370
561
48D1

(b) 7A32 : 0028H represents 7A320H + 0028H = 7A348H

7A320
0028
7A348

(10) If the stack segment register contains 3000H and the stack pointer register contains 8434H, what is the physical Address of the top of the stack ?

⇒ SS = 3000H and SP = 8434H

TOS = 30000 + 8434 H = 38434H or 3000● : 8434

(13) Describe the opercation 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.

(d) MOV BX, AX ; The contents of the AX registers are copied into the BX register. AX is unchanged.

(e) 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

(a) Load the number 7986H into the BP register => MOV BP, 7986 H

(b) Copy the BP registere contents to the SP register => MOV SP, BP

(c) Copy the contents of the AX register to the DS register => MOV DS, AX

(d) Load the number F3H into the AL register => MOV AL, 0F3H

(15) If the 8086 execution unit calculates an effective address of 14A3H and DS contains 7000H, what physical address will the BIU produce ?

= EA = 14A3H and DS = 7000H,

70000 + 14A3 = 714A3 address produced by the BIU = 714A3H

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

= DS = 4000H, 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 74B2CH

= MOV [74B2CH], DL

(18) Describe the difference between the instructions MOV AX, 2437H and MOV AX, [2437H]

>> MOV AX, 2437H = loads the AX register with the number 2437H.

MOV AX, [2437H] = copies the contents of memory location DS + 2437H into AL and the contents of memory location DS + 2437H + 1 into AH

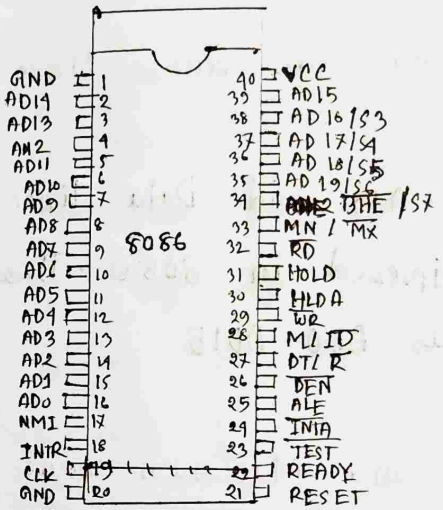
[i.e. loads the 16-bit AX register from two 8-bit memory locations]

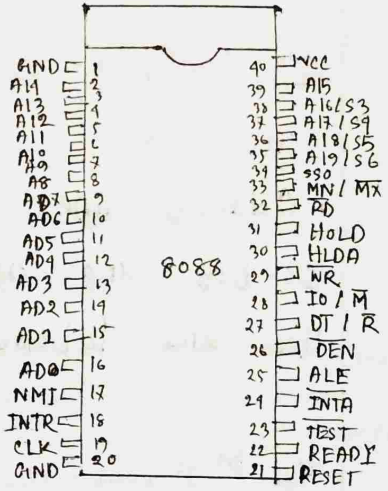
Lecture - 3

8086 / 8088 Microprocessor and it's 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 (HMOS) 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 A0-A7 and data lines D0-D7 are multiplexed in 8088. These lines are labelled as AD0 - AD7
- 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 A0 - A15 and Data lines D0 - D15 are multiplexed in 8086. These lines are labeled as AD0 - AD15

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 / $\overline{MX}$ 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 / $\overline{MX}$ 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_{cc}, pin 40 and uses ground at V_{ss}, pin 1 and 20 for its operation.

- GND = GND (0V)
- Clock Signal :

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

- **Address / data bus :**

AD0 - AD15 . These are 16 address/ data bus. AD0 - AD7 carries low order byte data and AD8-AD15 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 :**

A16 - A19 / S3 - S6 . These are the 4 address / status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

- **$\overline{BHE}$ / S7 :**

BHE stand for Bus High Enable . It is available at pin 34 and used to indicate the transfer of data using data bus D8 - D15.

- **$\overline{INTA}$:**

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

- **MN / $\overline{MX}$:**

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 Transreceiver 8286 . The transreceiver is device used to Separate data from the address / data bus .

- **DT / $\overline{R}$:**

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

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

- **M/$\overline{IO}$:**

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

- **$\overline{WR}$:**

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/$\overline{IO}$ 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.

- **$\overline{RQ}$ / $\overline{GT1}$ and $\overline{RQ}$ / $\overline{GT0}$:**

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

- **$\overline{LOCK}$:**

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

- **S_0, S_1, S_2 :**

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

S2	S1	S0	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	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 begining 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 controll of the status bus.

- QS_1 and QS_0 :

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

QS1	QS0	Charcacterisities
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

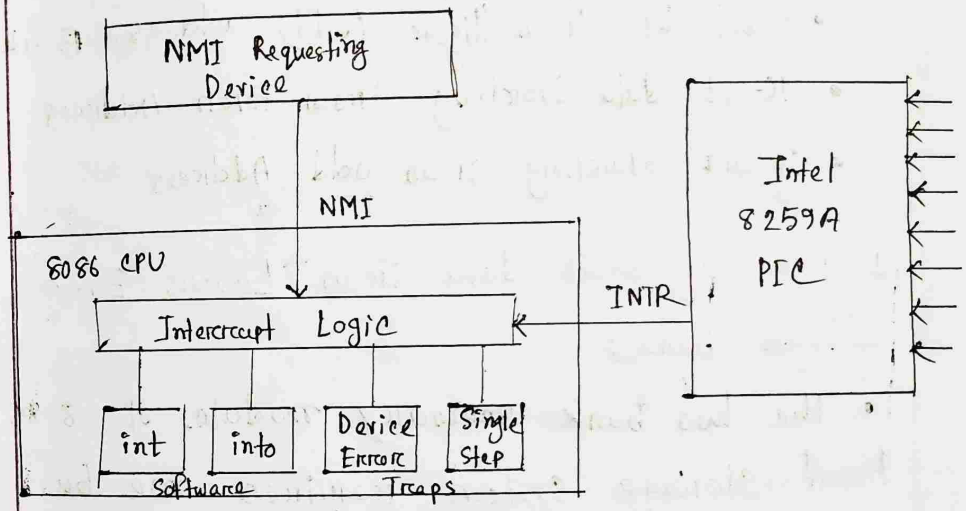
- NMI (input) : [Non-Maskable Interrupt Line]
- The Non Maskable interrupt, input is similar to INTR except that the NMI interrupt doesn't check to see if the IF flag bit is at logic 1.
- This interrupt can't be masked (or disabled) and no acknowledgment is required.
- It should be reserved for "Catastrophic" events such as power failure or memory errors.
- Test (input) :
- The $\overline{\text{Test}}$ pin is an input that is tested by the WAIT instruction
- If $\overline{\text{Test}}$ is at logic 0, the WAIT instruction functions as a NOP
- If $\overline{\text{Test}}$ is at logic 1, then the WAIT instruction causes the 8086 to idle, until $\overline{\text{Test}}$ 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

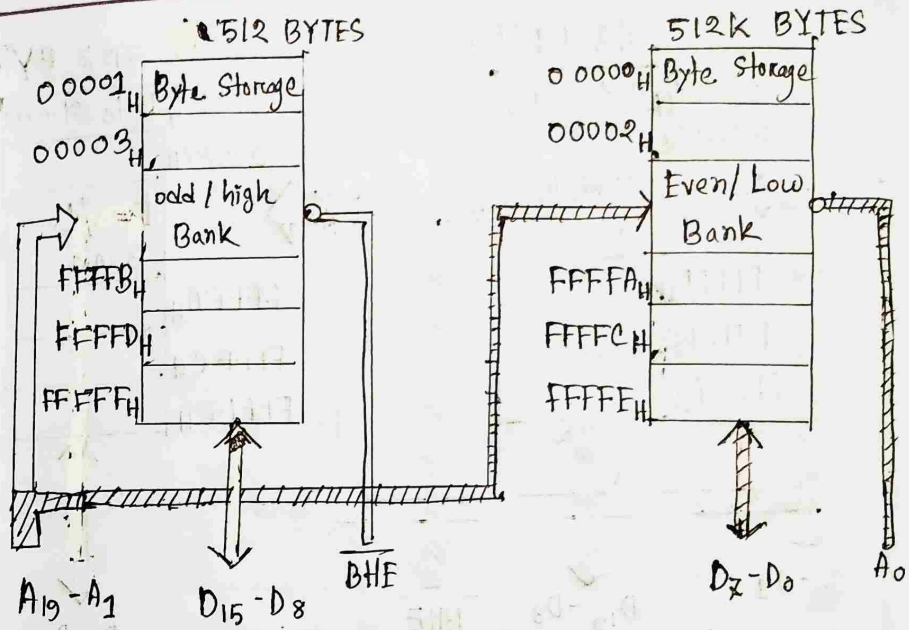


Data can be accessed from the memory in four different ways.

- 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

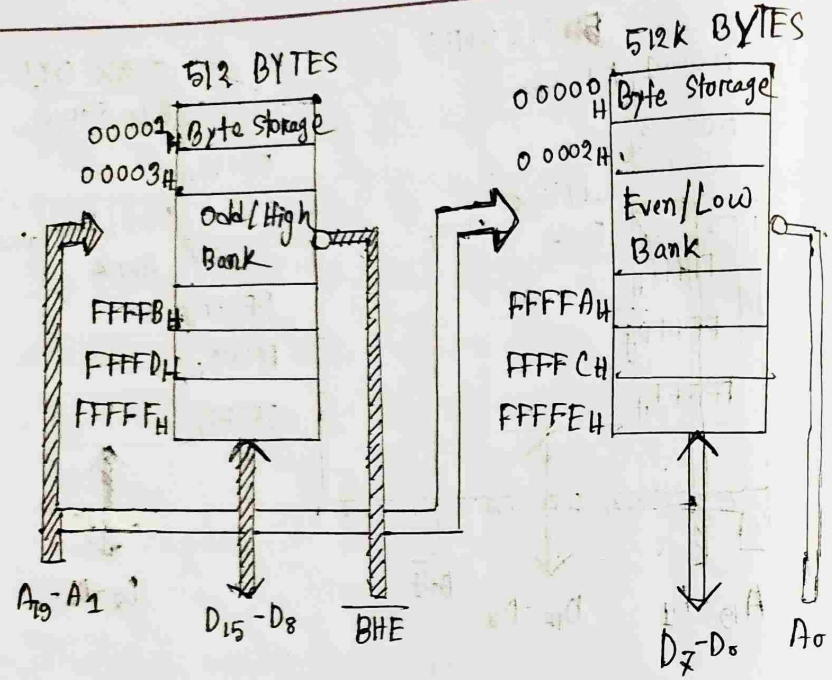
1. Accessing 8-bit data from Lower (Even) address bank:

- The two bank memory module of 8086 based storage system requires one bus cycle to read/write a data bytes.
- To access a Byte of data in Low-Bank, valid address is provided via address pins A1 to A19 together with A0 = '0' and $\overline{BHE}$ = '1'



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

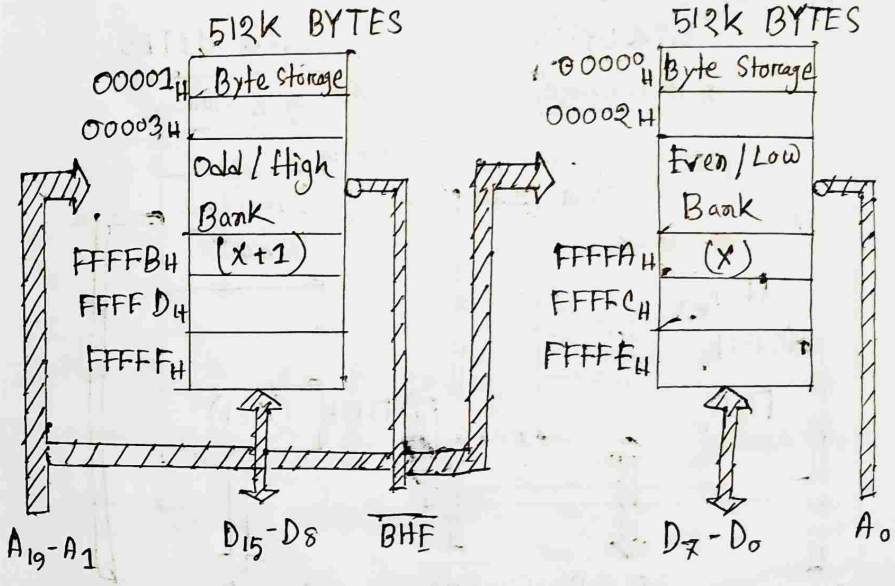
- Similarly to access a Byte of data in High-bank, valid address in pins A1 to A19, A0 = "1" and $\overline{BHE}$ = '0' are required to access the data through D8 to D15 of the data-bus.
- These signals disable the Low bank and enable the High bank to transfer (in/out) data through D8 to D15 of the data-bus.



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

- 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 A0 = '0' and $\overline{BHE}$ = '0'

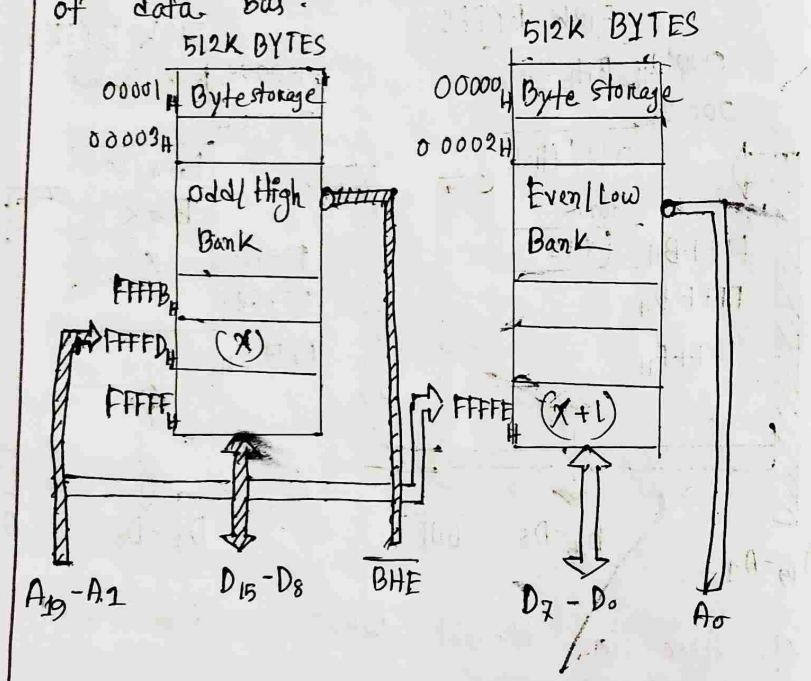
- Note that during this bus-cycle, all 16-bit data is transferred via D0 to D15 of the data bus



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

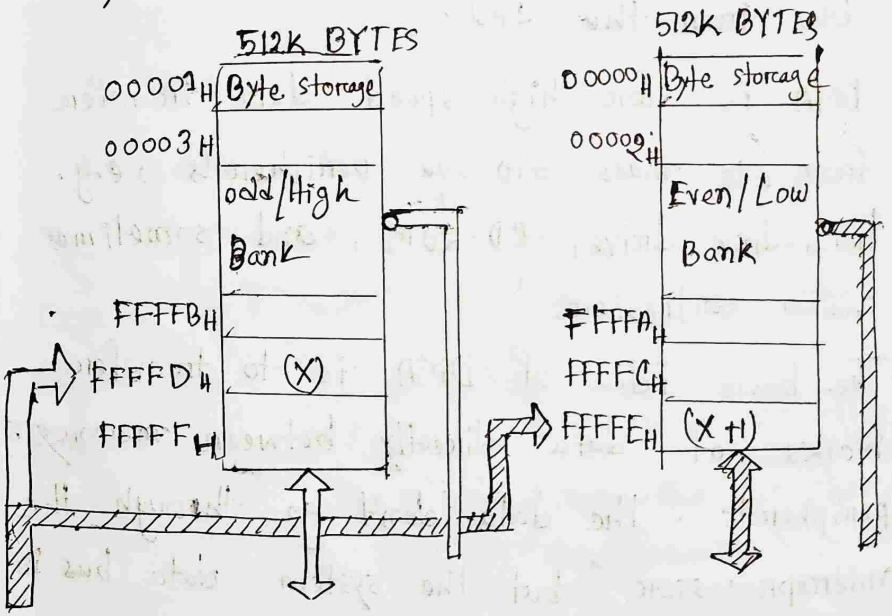
- For odd-addressed (unaligned) words (with odd P.A. of the LSB), two bus-cycles are required to access the Word-data.

• During the 1st bus-cycle, odd addressed LSB of the word is accessed from the High-memory-bank via D8 to D15 of data bus.



• During 2nd bus-cycle, P.A. is auto-incremented to access the even address MSB of the word from the Low Bank via D0 to D7

• Note that A_0 and $\overline{BHE}$ signals are reset (violet) accordingly to enable the required memory bank.



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 event of a typical DMA process.

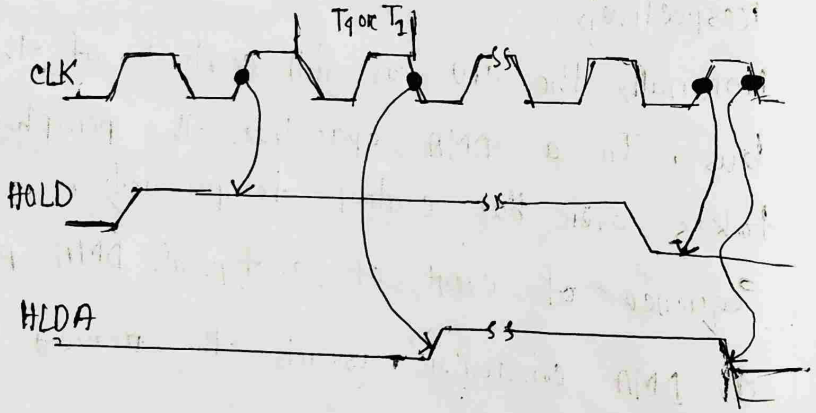
① DMA controller asserts the request on the HOLD pin.

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 HOLDA pin. 8086 pins (Address, Data, C-trol => 3-rd 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/GT1 and RQ/GT0 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/GT1 or RQ/GT0 (RQ/GT0 has higher priority)

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

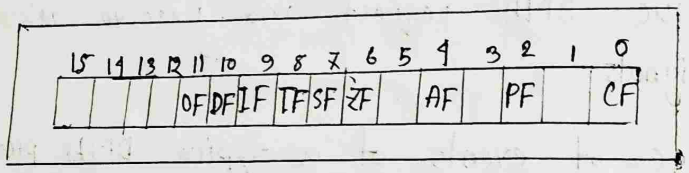
③ 8086 grants the right of bus control by asserting a grant signal via the same pin as the request signal

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 Register:



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 substraction; 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 contain 7 one bits, so PF = 0

- **Auxilary 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 binaary-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 ~~or~~ 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 -

```
    1111   1111   1111   1111
(+) 0000   0000   0000   0001
-------------------------------
  1 0000   0000   0000   0000
```

• If we are giving an unsigned interpre-
-tation, 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 ans-
-wer 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 7FFFh and 7FFFh (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 7FFFh and we execute ADD AX, BX. The binary result is:

```
  0111  1111  1111  1111
+ 0111  1111  1111  1111
-------------------------
  1111  1111  1111  1110  = FFFEh
```

- The signed and unsigned decimal interpretation of 7FFFh is 32767. Thus for both signed and unsigned addition, 7FFFh + 7FFFh = 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 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 the processor executes an instruction, the flags are altered to reflect the result. But, those instructions don't effect any of the flags, affects only some of them, or may leave them undefined.

Examples

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

Solution:

```
  FFFFh (1111 1111 1111 1111)
(+) FFFFh (1111 1111 1111 1111)
 1FFFEh (1111 1111 1111 1110b)
```

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 : 5.2 :

ADD AL, BL ; where AL contains 80h, BL contains 80h

Ans)

```
  80h
+ 80h
-----
 100h
```

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:

```
  8000h
(-) 0001h
-------
  7FFFh = 0111 1111 1111 1111
```

The result stored in AX is 7FFFh

- 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 negatives. Because the result is positive (the wrong sign), so, OF = 1.

Example : 5.4 INC AL, where AL contains FFh

Solution:

```
  FFh
+  1h
-----
 100h
```

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 afterward.

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

Example 5.5 : MOV AX, -5

Solution:

The result stored in AX. is -5 = FFFBh

None of the flags are affected by MOV.

Example : 5.6

NEG AX, where AX contains 8000h

Solution:

```
        8000h = 1000 0000 0000 0000
one's complement = 0111 1111 1111 1111
                                   +1
-------------------------------------------
                1000 0000 0000 0000 = 8000h
```

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 it's own two's complement, there is no sign change.

Lecture - 5

Flow Control Instructions

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

- Singned Jumps

Symbol	Description	Condition for Jump
JG / JNLE	jump if greater than • jump if not less than or equal to	ZF = 0 and SF = OF
JGE / JNL	jump if greater than or equal to. jump if not less than or equal to	SF = OF

JL / JNGE	jump if less than. jump if not greater than or equal	SF <> OF
JLE / JNG	jump if less than or equal. jump if not greater than	ZF = 1 or SF <> OF

- Unsigned Conditional Jumps :

Symbol	Description	Condition for Jumps
JA / JNBE	jump if above. jump if not below or ~~below~~ equal	CF = 0 and ZF = 0
JAE / JNB	Jump if above or equal. jump if not below	CF = 0
JB / JNAE	Jump if below. Jump if not above or equal	CF = 1
JBE / JNA	jump if equal. jump if not above	CF = 1 or ZF = 1

- Single - Flag Jumps :

Symbol	Description	Condition for Jumps
JE / JZ	jump if equal. Jump if equal to zero	ZF = 1
JNE / JNZ	jump if not equal. jump if not 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 parcity even	PF = 1
JNP / JPO	Jump if parcity odd	PF = 0

ইবাদাত যা করার একনিষ্ঠ করতে হবে, সময় নষ্ট করে শয়তানকে সুযোগ দেয়া যাবে না।

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 = 7FFFh, and BX = 0001. The result of CMP AX, BX is 7FFFh - 0001h = 7FFEh. Here, the jump condition for JG is satisfied, because, ZF = SF = OF = 0, so control transfers to label BELOW.

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 = 7FFFh, BX = 8000h, and we execute

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

then, even though 7FFFh > 8000h in a signed sense, the program does not jump to BELOW. The reason is that 7FFFh < 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
```

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.

High-Level Language Structures:

• Branching structures:

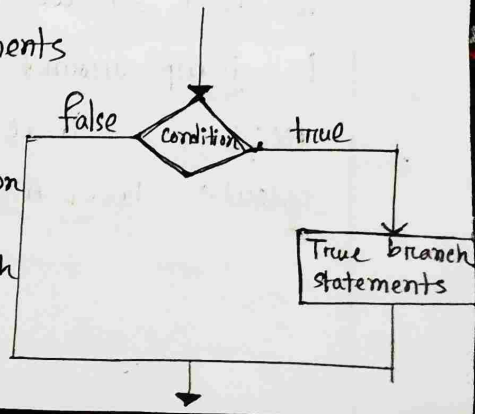
In high-level languages, branching structures enable a program to take different paths, depending on conditions. In this section, we'll look at three structures.

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 number in AX by its absolute value.

=> A pseudocode Algorithm is :

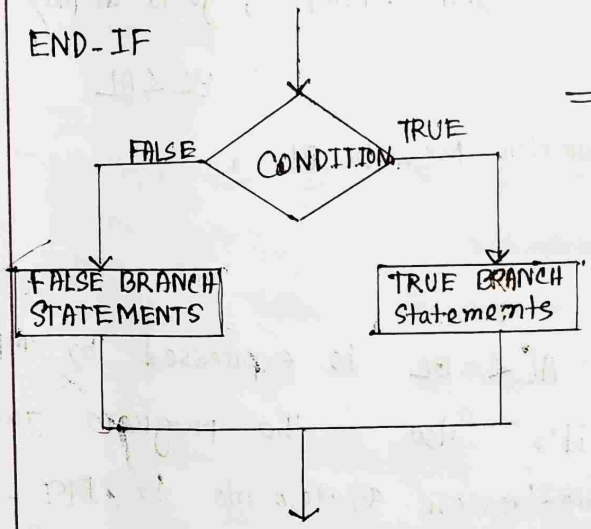
Pseudocode	Code
If AX<0	CMP AX,0 ; AX<0?
Then	JNL END-IF ; no, exit
Replace AX by -AX	NEG AX ; yes, change sign
END-IF	END-IF ;

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	Code

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

THEN display the character in AL	; AL <= BL MOV DL,AL ; move chars to be displayed JMP DISPLAY ; go to display
ELSE	; BL<AL
display the character in BL	MOV DL,BL ;
END_IF	

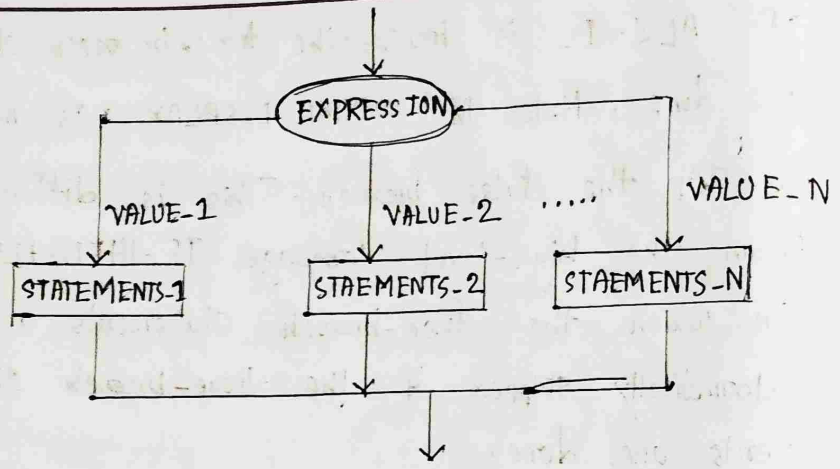
The condition AL<=BL is expressed by CMP AL,BL . If it's false , the program jumps around the true-branch statements to ELSE-. We use the unsigned jump JNBE (jump if not below or equal), because we're comparing extended characters.

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 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 $ values-1, then statements_1 are executed. We assume that sets values_1 .... 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.

=>

```
CASE AX
   <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 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 'o' ; If AL contains 2 or 4, display 'e'

Solution:

```
CASE AL
   1,3 : display 'o'
   2,4 : display 'e'
END-CASE
```

The code is :

```
; case AL
; 1,3;
        CMP  AL, 1     ; AL = 1 ?
        JE   ODD       ; yes, display 'o'
        CMP  AL,3      ; AL = '3'
        JE   ODD       ; yes, display 'o'
; 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 'o'
        MOV  DL, 'o'   ; get 'o'
        JMP  DISPLAY   ; go to display
                       ; display 'e'
EVEN :
        MOV  DL, 'e'   ; get 'e'
DISPLAY :
        MOV  AH, 2
        INT  21H       ; display ~~AH~~ 'char
END-CASE ;
```

Branches with Compound Conditions :

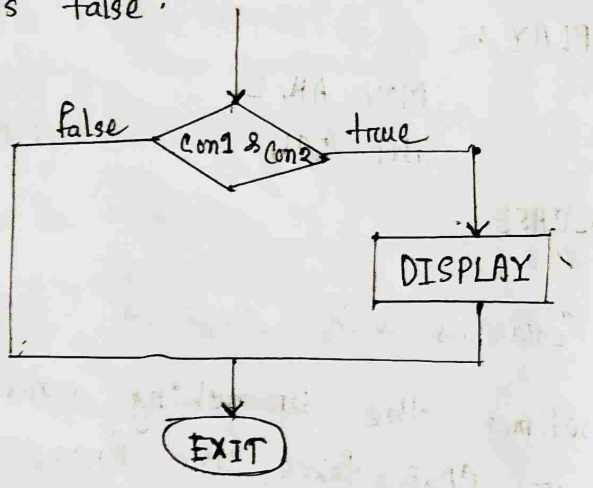
Sometimes the branching condition in an IF or CASE takes the form ..

→ 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:

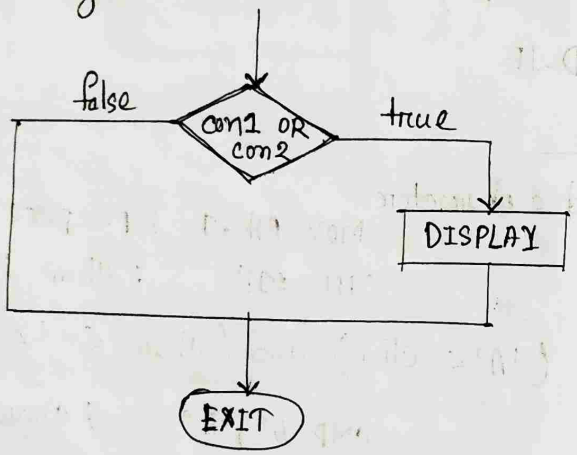
```
Read a character (into AL)
IF ('A' <= character) and (character <= 'Z')
   THEN
      display character
END-IF
```

Code

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

=>

- Pseudocode:

```
Read a character (into AL)
IF (character = 'y') OR (character = 'Y')
   THEN
      display it
   ELSE
      terminate the program
   END-IF
```

Code:

```
; read a character
        MOV  AH, 1        ; prepare to read
        INT  21H          ; char in AL
; If (character = 'y') or ( character = 'Y')
        CMP  AL, 'y'      ; char = 'y' ?
        JE   THEN         ; yes, go to display it
        CMP  AL, 'Y'      ; char = 'Y' ?
        JE   THEN         ; yes, go to display it
        JMP  ELSE-        ; no, terminate

THEN:
        MOV  AH, 2        ; Prepare to display
        MOV  DL, AL       ; get char
        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 :

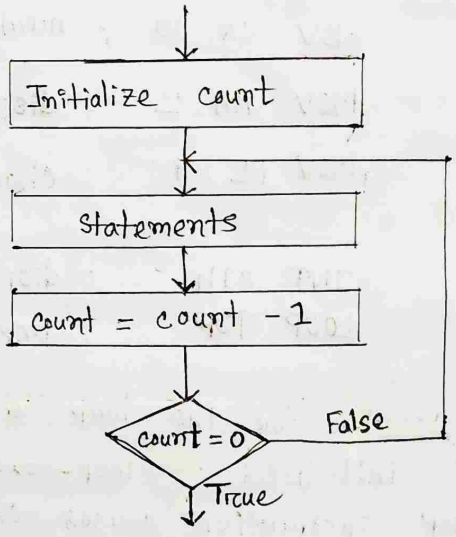
1. For loop
2. While loop
3. Repeat.

- 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 decrement 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:

```
        ; initialize CX to loop-count
TOP :
        ; 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. It's 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 :

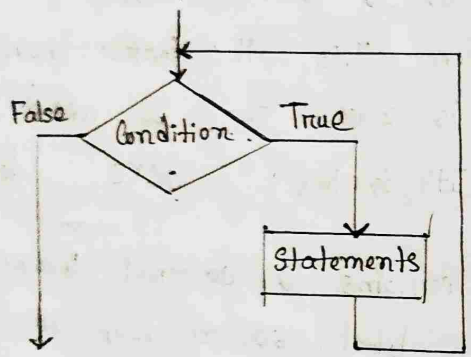
```
        JCXZ SKIP
TOP :
        ; body of the loop
        LOOP TOP
SKIP :
```

While loop :

This loop depends on a condition. In Pseudocode,

```
WHILE condition DO
    statements
END-WHILE
```

figure of while loop :



*The condition is 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      ; Character 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.

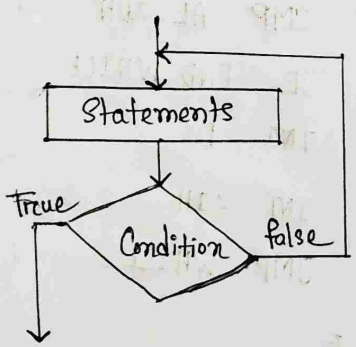
REPEAT LOOP :

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)

```
    CMP   AL,0
    JNL   ELSE_
    MOV   AH,0FFh
    JMP   END_IF
ELSE_: MOV  AH,0
END_IF
```

or,

```
    CMP   AL,0
    JL    THEN_
    MOV   AH,0
    JMP   END_IF
THEN_: MOV  AH,0FFh
END_IF:
```

(c)

```
    CMP   DL,'A'
    JL    END_IF
    CMP   DL,'Z'
    JG    END_IF
    MOV   AH,2      ; display DL
    INT   21H
END_IF;
```

(d)

```
    CMP   AX,BX
    JGE   END_IF
    CMP   BX,CX
    JGE   ELSE_
    MOV   AX,0      either
    JMP   END_IF
ELSE_: MOV  BX,0
END_IF;
```

(e)

```
    CMP   AX, BX
    JL    THEN-
    CMP   BX, CX
    JL    THEN-
    MOV   DX, 1    ; else
    JMP   END-IF
THEN-: MOV  DX, 0
END-IF
```

(f)

```
        CMP AX,BX
        JNL ELSE-
        MOV AX,0
        JMP END-IF
ELSE-:  CMP BX,CX
        JNL ELSE-2
        MOV BX,0
        JMP END-IF
ELSE-2: MOV CX,0
END-IF:
```

Lecture: 7

Logic, Shif and Rotation

Shift

Shift opercation move the bits in a patterm, Changing the position of the bits. They can move bits to the left or to the right. We can divide shift opercation into two categories : 1. Logical Shift
2. Arcithmetic 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 signi-ficant bit MSB is assigned 0.
- In a logical left shif, the LSB is assig-ned to 0 and the msb is dicarded.

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

There are two kinds of Logical shift:

1. Shift Left.
2. Shift Right.

Left Shift: (SHL)

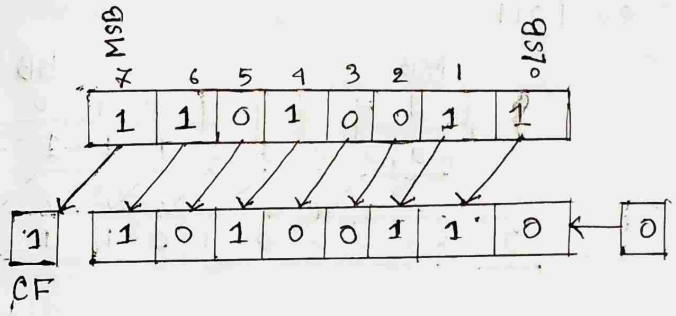
• A shift left logical of one position moves each bit to the left by one. The low-order bit LSB is replaced by a zero bit and the high-order bit MSB move to CF (carry flag).

• 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 0 zero.

SHL: Syntax: SHL Register, Bits to be shifted

Example: SHL AX, 2

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



• Right Shift: (SHR)

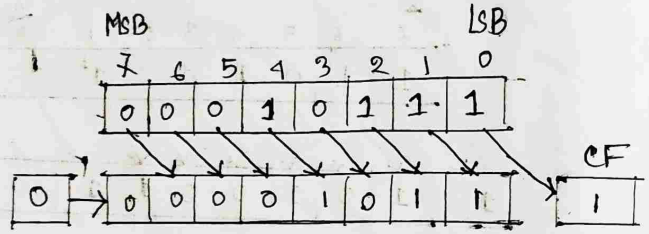
• A shift right logical of one position moves each bit to the right by one. The high-order bit MSB is replaced by a zero bit and the low-order bit LSB move to CF (carry Flag)

• 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

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

তারিখ

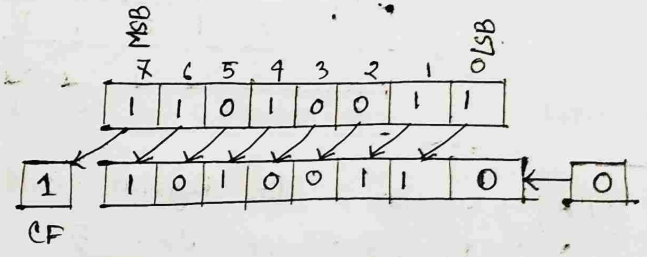
• Left Arithmetic Shift : (SAL)

• 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).

• 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.



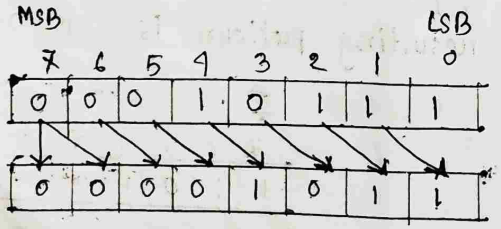
• Shift Arithmetic Right : (SAR)

• 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 doesnot 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.

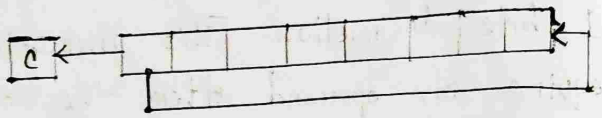
There are 4 types of the ROTATE instruction:

1. ROL : Rotate Left
2. ROR : Rotate Right
3. RCL : Rotate Carry Left
4. RCR : Rotate Carry Right

• ROL : Rotate Left :

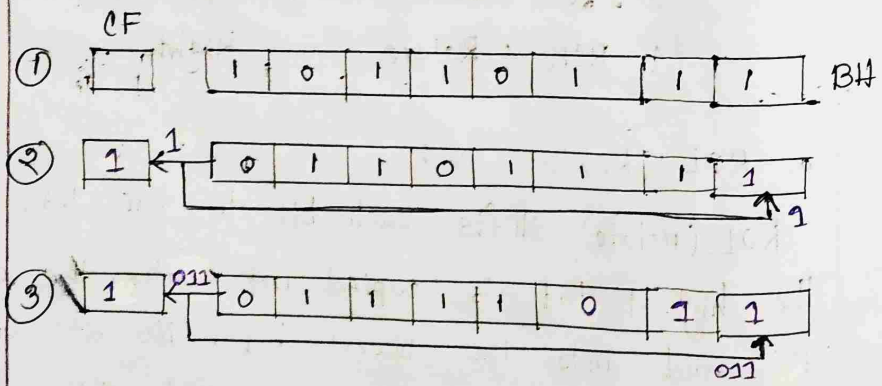
ROL (rotate) shifts each bit to the left. The highest bit is copied into both the carry flag and into the lowest bit. No bits are lost. ROL is used for unsigned data.

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

ROL : 

Below are instances of the ROL instruction :

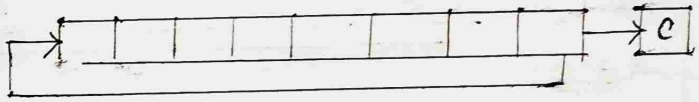
	Instruction	Comment	Binary	CF
①	MOV BL, 10110111B	; Initialize BH	10110111	1
②	ROL BL, 01	; Rotate Left 1	01101111	1
	MOV CL, 03	; Set rotate value		
③	ROL BL, CL	; Rotate left three more	01111011	1



• ROR : Rotate Right :

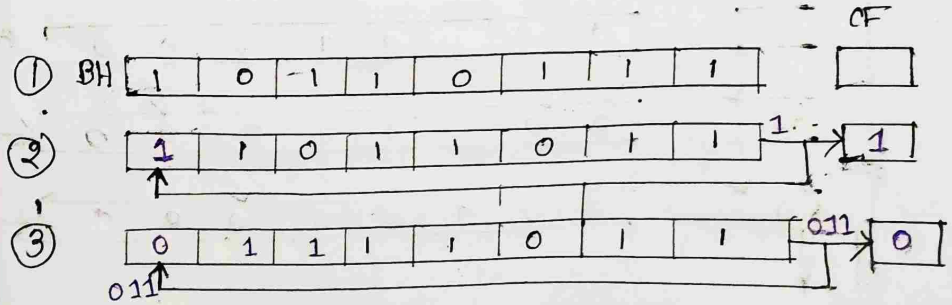
ROR (rotate right) shifts each bit to the right. The lowest bit is copied into both the Carry flag and into the highest bit. No bits are lost. ROR is for unsigned data.

[label:]	ROR	register/memory, CL/immediate

ROR : 

• 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
	MOV CL, 03	; Set rotate value		
③	ROR BL, CL	; Rotate right three more	01111011	0

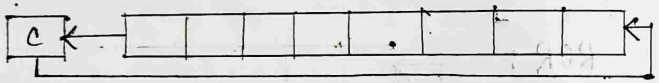


• RCL : Rotate Carry Left :

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

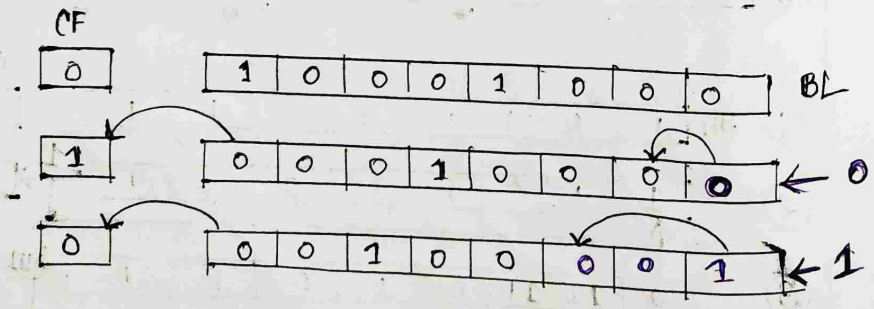
[label:]	RCL	register/memory, CL/immediate

RCL :



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
```

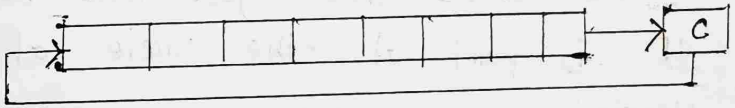


RCR : Rotate Carry right :

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.

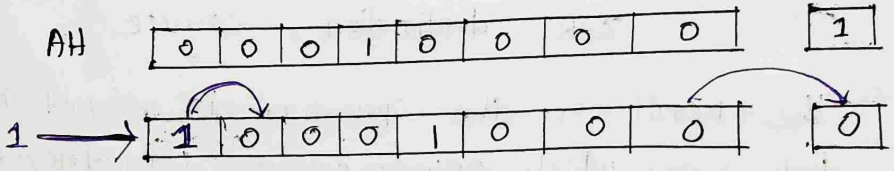
[label:]	RCR	register/memory, CL/immediate

RCR :



• 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 ANR, 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) :

b AND 1 = b , b OR 0 = b , b XOR 0 = b

b AND 0 = 0 , b OR 1 = 1 , b XOR 1 = $\bar{b}$ (complement of b)

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