Physical memory organization of 8086 memory address.

UNIT-1 8086 Architecture

Introduction :

Process :- Performs the operation on data is called Process

Processor : It is a programmable integrated device that has computing and decission making capability similar to that of CPU.

machine language : Instructions are in the form of binary patterns called machine language.

mnemonics : Binary instructions are given abbreviated names called as mnemonics.

Assembly language : If the processor has written with mnemonics that is called as assembly language.

Def of microprocessor : A microprocessor is a multipurpose, Programmable clock-driven, register based electronic device that reads binary instructional from a storage device called memory. accepts binary data as i/p & processes data according to those instructions & provides results as o/p

compiler : The compiler reads the entire prog. first & translates it into object code that is executed by the microprocessor.

Interpreter :- The interpreter reads one instruction at a time & produces its object code & executes the instruction before reading the next instruction.

difference b/w high level language & assembly language

* In high level language the instruction are in english so compilers & interpreters requires several machine codes to translate it into binary.
* On the other hand, there is one-to one correspondance b/w assembly language mnemonics and machine code.
* Thus assembly language prog. are compact & require less memory space They are more efficient the high level language programs
* The primary advantages of high level language is troubleshooting the prog.

applications of assembly language is.

Traffic control programs

application of high level language is

Video games.

The details of the different micro processor.

micro processor	data bus size	address bus	memory add. capability	No. of Pins.
4004	4 bits	10 bits	640 bytes	16
8008	8 bits	14 bits	16 k bytes	18
8080	8 bits	16 bits	64 k bytes	40
8085	8 bits	16 bits	64 k byte	40
8086	16 bits	20 bit	1 m b	40
8088	8/16 bit	20 bit	1 m b	40
80186	16 bits	20 bit	1 m b	68
80286	16 bits	24 bit	16 m b	68
80386	32 bit	32 bit	4 GB	132.
80486	32 bit	32 bit	4 GB	168.
80586	64 bits	32 bit	4 GB	273.

8086 microprocessor

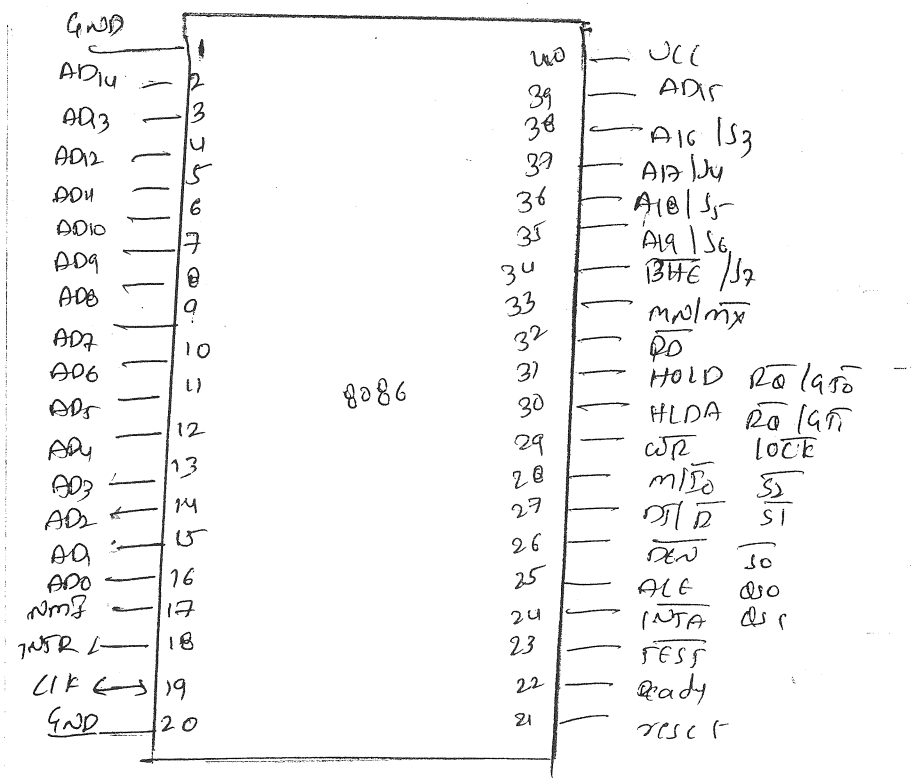
Features of 8086 :

* It is 16-bit microprocessor.
* Implemented in N-channel, depletion load, silicon gate technology & it has 40 pins
* It has 16 bit data bus. So it can read data from (or) write data to memory & ports either 16 bits (or) 8-bits at a time.
* It does not have internal clock circuit. The 8086 requires an external asymmetric clock source with 33% duty cycle. The 8284 clock generator is used to generate the required clock for 8086.
* It has 20 bit address bus. So it can access 2^{20} = 1 mbyte memory locations.

1MB addressable memory of 8086 is organized as two memory banks of 512kB each. The memory banks are called even bank and (lower bank), odd bank (upper bank). The address line A0 is used to select even bank and the control signal $\overline{BHE}$ is used to select odd bank.

* For accessing I/O mapped device, 8086 uses a seperate 16-bit address. So it can access 64k I/O ports.
* The 8086 can operate in two modes, they are "minimum mode" and "maximum mode". The mode is decided by the signal at MN/$\overline{MX}$ Pin when MN/$\overline{MX}$ is high, it works in minimum mode & the system is called "uniprocessor system". when MN/$\overline{MX}$ is low it works in maximum mode. & the system is called "multiprocessor system".
* It fetches upto six instruction bytes from memory & stores in a queue fashion, in order to speed up the instruction execution.
* It supports multiprogramming. In multiprogramming the code for two (or) more process is in memory at the same time it executes in time multiplexed fashion.

Pin diagram of 8086 :-



* 8086 signals can be catogonized in three groups. The first are the signals having common functions in minimum as well as maximum mode.
* The second are the signals which have special functions for minimum modes.
* The third are the signals having special functions for maximum mode.

Following signals description are common for the both min. & max. mode :

$AD_{15} - AD_0$: These are the time multiplexed memory I/O address and data lines. address remains on the lines during T_1 state, while the data is available on the data bus. during T_2, T_3, T_W & T_4. Here T_1, T_2, T_3, T_W & T_4 are the clock states of the machine cycle. T_W is a wait state. These lines are active high & float to a tristate during interrupt acknowledge & local bus hold acknoledge cycle.

A_{19}/S_6, A_{18}/S_5, A_{19}/S_4, A_{16}/S_3 : These are time multiplexed address & status lines. During T_1, T_1, these are the most significant address lines for memory operations. During I/O operational these lines are low.

During memory (or) I/O operational status information is available on these lines for T_2, T_3, T_W, T_4.

* S_5 is the interpt unable flag.
* These S_4 & S_3 combinely indicate which segment reg. is presently being used for memory access.

S_4	S_3	Indication
0	0	extra segment
0	1	stack.
1	0	code (or) No. segment
1	1	Data.

* A low on the S_6 indicates that 8086 is on the bus. During the hold ack. this pin is driven to high impedence state.

$\overline{BHE}/S_7$ (Bus high enable / status 7) :

The Bus high enable signal is used to indicates the transfer of data over the higher order data bus ($d_{15} - d_8$). It goes low for the data transfer over $D_{15} - d_8$. & It is used to select the odd address memory bank.

$\overline{BHE}$ is low during T_1 for read, write. & interrupt acknowledge cycles when the data is to be transferred to the higher order data bus.

The status information is available during T_2, T_3, T_4.

$\overline{BHE}$	A_0	Indication
0	0	whole word
0	1	upper byte from (or) to odd address
1	0	lower byte from (or) to even address
1	1	None.

Ground : To provide zero voltage.

Vcc: To give the power supply. (5V)

INTR: INTR is used to intrupt to the process and it is level trigger.

Ready: (ready to process) It is a aronmial signal to given to lower devile like peripheral memory. For the indication of data transfer compliation that is if ready = 1 it indicates the Processor is ready for the next data Transfer.

Reset: If reset = 1 it trimintes the current process and restart from FFFF0H. Once reset is equal to one. that must be active. atleast 4 clock sides.

TEST: This examintes the weight instruction if test is low the process continously execution. otherwise the Processor entr into the ON state. test is high the processor moderatly execution.

clk: clk is used to basic time of the operation. of the processor.

NmI: maintx & RD

This is an edge. triggered. i/p which causes a type 2 intrupt. The NmI is not maskable internally by soft ware. A transition from low to high initiates the interrupt response at the end of the current instructions. This i/p is internally synchronized.

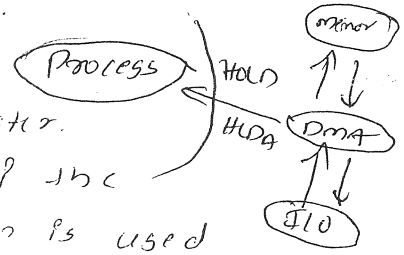
MN/mx: The logic level at this pin decides whether the processor is to oprate in either minimum or maximum mode.

RD: Read signal, when low, indicates the peripherals that processor is performing a memory or I/O read operation. RD is active low and shows the state for Bus, Two of any read cycle. The signal mains tristated during the hold acknowledge.

minimum mode pins:

HOLD: It is used to send the request from the other bus master (DMA) to get the bus access to the processor.

HLDA: Hold acunoledgment to the bus master. In order to acceptness of the bus accessing this pin is used acunoledgement to the other DMA acceptness in the controller.

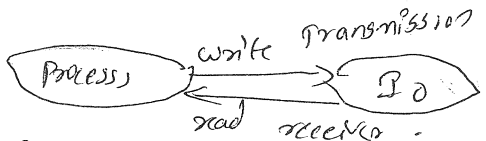


WR: It is used to performing the write. operation. It is low indirates the Processor. to writing operation. It is active.

(S2, S3, S4, S5, S6)

$M/\overline{IO}$: To know the operation is used to memory/ $\overline{IO}$. If it is equal to 1 it is involve in the memory and if it is equal to 0 it involve in the $\overline{IO}$. It is active from the Tu of the previous machine cycle. to till Tu of the current machine cycle.

$DT/\overline{R}$: data transmission or receive bar. It is indicate the type of operation. $DT/\overline{R} = 1$ it is the data transmission and $DT/\overline{R} = 0$ data receive.



$\overline{DEN}$: $\overline{DEN}$ it is used to enabled the data transmitting and simply data buffer.
* It is used to separate the data lines from the multiplexed addressed data. It is active from the middle of T2 to Tu.

ALE : address latch enable. ALE is used to enabled the latches its separate the address lines from the multiplexed data lines.

$\overline{INTA}$: Intr acknowledgement it is used. send to acknowledgement signal to the other devices.

Maximum mode : $RQ/\overline{GT_1}$ (HLDA) :

$RQ/\overline{GT_0}$ (HOLD) : $\overline{GT_0}$ is zero in maximum to the pin is used to stay bus master to reduce the bus access $RQ/\overline{GT_0}$ is highest partly than the $RQ/\overline{GT_1}$.

$\overline{LOCK}$: If it is low then the other per bus master are prevented for getting the bus access.

$\overline{S_2}, \overline{S_1}, \overline{S_0}$: The three states pin gives the what type of operation is perform memory/IO.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Indication
0	0	0	Interrupt acknowledge
0	0	1	IO read
0	1	0	IO write.
0	1	1	Halt
1	0	0	code access
1	0	1	memory read
1	1	0	memory write.
1	1	1	passive

QS0. QS1 : It is the two states operation.

QS_1	QS_0	Indication.
0	0	No operation
0	1	first byte of opcode from Q
1	0	Q is empty.
1	1	Sub sequend byte of opcode from Q.

Register organization of 8086 :

* It has a powerful set of registers. containing general perpus & special perpus registers.
* All the registers of 8086 are 16 bit registers.
* The registers set is devided into 4 groups.
  1. General data reg.
  2. Segment reg.
  3. Flags / PSW
  4. pointers & index reg.

General data register :- The reg. AX, BX, CX and DX are the general perpus registers. The general perpus register can be used as either 8 bit reg. or 16-bit reg.

* The general perpus reg. are used for holding data. variables & intermediate result tempora-rily. or for other perpus like a counter or for storing offset address for some particular addressing modes.

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL.

AX :- It is used to 16-bit accumulator with lower 8 bit of AX. It designated as AL & higher 8 bits as AH. AL can be used as 8-bit accumulator

BX :- The reg. BX is used as offset storage for forming physical address in rage certain addressing modes.

CX :- The reg. CX is used as a default counter in case of string & loop instruction.

DX :- The reg. DX is used as an implicit operand or destination in case of few instructions.

Segment register :- The physical address of 8086 is 20 bits wide to acces 1MB memory locations. However, its registers & memory locations which contain logical address are just 16 bits wide. Hence 8086 uses memory segmentation

1 MB of memory is devided into 16 logical segments. Each segment thus contains 64 KB of memory. Thus any location with in the segment can be accessed using 16 bits. The 8086 allows only four active segments at a time for the selection of from four active segments the 16 bit segment registers are used These are code segment reg. (CS). Data segment reg (DS) External segment reg. (ES). & stack segment reg (SS).

CS :- code segment reg. is used for addressing a memory location in the code segment of the memory. where the executable program is stored.

DS :- DS is used for addressing a memory location in the data segment of the memory. where the data is stored.

ES :- ES is also refers a segment which essentially another data segment of the memory. Thus Extra segment also contains data.

SS :- S.S is used for addressing stack segment of memory SI contains stack data.

physical address is calculated from two parts. first is segment address & second is offset address. The segment reg. contain 16 bit segment base addresses related to different segments. Any pointers and index reg. (or) BX may contain the offset address.

pointers & index reg: The pointers contain offset with in particular segments. The pointers IP, BP & SP usually contain offsets with in code, data, & stack & segment respectively.

The index reg. are used as genral purpose reg. as well as for offset storage in case of indexed, base indexed & relative based indexed addressing modes. The reg. SI & DI is generally used to store the offset of source data in data segment. & DI is used to store the offset of distination in data (or) extra seg. The index reg. are particularly used for string manipulation.

Flag register: The 8086 flag reg. contains indicates the results of computations in ALU It also contains some flag bits one to control the CPU operations.

* 16 bit flag reg. is devided in to two parts
i) condition code (or) status flags
ii) machine control flags

The condition code flag reg. is the lower byte of the 16-bit flag reg. along with the overflow flag. The condition code flag reg. is identical to 8085 flag reg. with an additional overflow flag. which is not present in 8085. This part of flag reg. of 8086 register holds the results of the operation performed in ALU.

The control flag reg. is the higher byte of the flag reg. of 8086. It contains three flags

1) Direction flag.
2) Interrupt flag
3) Trap flag.

X	X	X	X	O	D	I	T	S	Z	X	AC	X	P	X	CY

CY → carry flag.
P → parity flag.
AC → auxiliary carry flag.
Z → zero flag.
S → Sign flag.
T → Trap flag.
I → Interrupt flag.
D → direction flag.
O → Overflow flag.
X → not used.

Carry flag: This flag is set when there is a carry out of MSB in case of addition or a borrow is needed in case of subtraction.
Parity flag: This flag is set to 1, if the lower byte of the result contain even no. of "1"s.
auxiliary carry flag: This is set, if there is a carry from the lowest nibble i.e bit three during the addition or borrow for the lowest nibble i.e. bit three during the subtraction.
zero flag: This flag is set of the result of the result operation in ALU is zero. this flag resets if the result is non zero.
Sign flag: This flag is set when the result of any computation is -ve. for signed computations. the sign flag equally the MSB of the result.
Over flow flag: This is set to 1 if the result of a signal operation is large enough to accommodated in a destination register.

Ex:- Give the contents of the flag reg. after execution of the following additions

0110	0101	1101	0001
0010	0011	0101	1001
1000	1001	0010	1010

SF=1, ZF=0 CY=0 AF=0. OF=1 PF=0.

Give the contents of the flag reg. after executing of the following substraction

0110	0111	0010	1001
-0011	0101	0100	1010
0011	0001	1101	1111

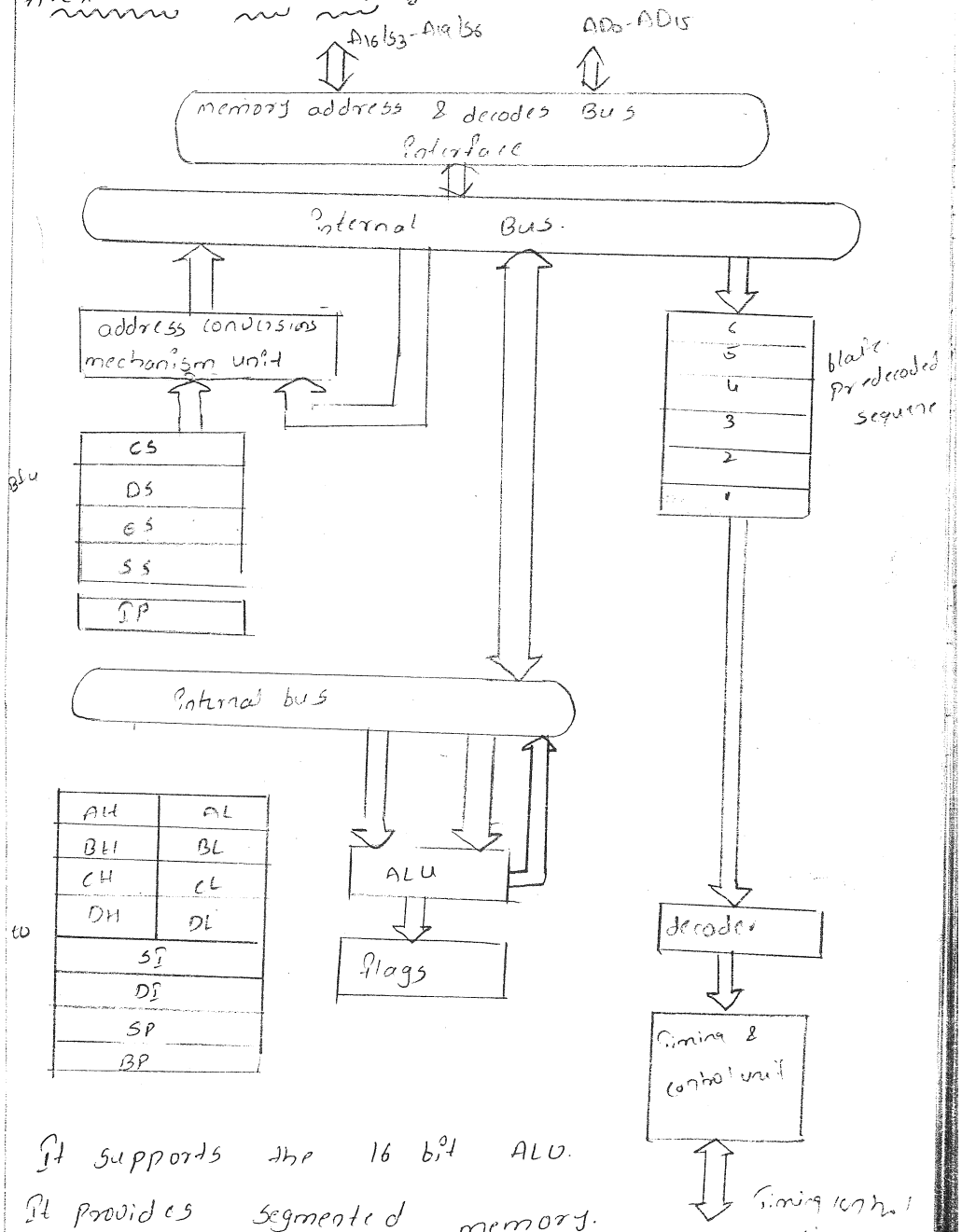
SF=0 ZF=0 CY=0 PF=0 AF=1 OF=0

TRAP flag:- In this flag is set, the processor enters the single step execution mode.

Interrupt flag:- If this flag is set the maskable interrupts are recognized by the CPU. other use they are ignored.

direction flag:- This is used by string manipulation instructions. If this flag bit is '0' the string is processed begining from the lowest address to the highest address. i.e auto increment mode. Otherwise string is processed from the highest address towards the lowest address i.e auto decrement mode.

Architecture of 8086:-



It supports the 16 bit ALU.

It provides segmented memory addressing capability.

It has 16-bit registers

It has predecoded instruction queue.

The architecture of 8086 is devided into two parts i) Bus interface unit (BIU)
ii) Execution unit (EU)

* The BIU contains the ckt for physical address calculations & prefetched instruction byte queue. (6 bytes long).
* BIU is responsible for establishing communication with external device and memory via bus.
* The complete physical address which is 20 bits long is generated using segment & offset registers each 16 bit long.

For generating the physical address from the content of these two registers, the content of the segment reg. also called segment address is shifted left bit-wise four times & to this result. content of an offset reg. also called as offset address is added to produce 20 bit physical address.

for example: segment add. 1005H
offset add. 5555H

Segment address 1005 = 0001 0000 0000 0101
shifted left by 4 bit position = 0001 0000 0000 0101 0000
offset add. + 0101 0101 0101 0101
0001 0101 0101 1010 0101
1 5 5 5 A

The segment reg. indicates the base address of a particular segment while the offset indicates the distance of the required memory location in the segment from the base add.

* BIU has a separate adder to perform this procedure for obtaining the physical address while addressing the memory. The segment address value is to be taken from an appropriate segment reg. depending upon whether code data or stack are to be accessed, while the offset may be the content of IP, BX, SP, DI, SP or an immediate 16 bit-value depending upon the addressing mode.

In case of 8085, once the opcode is fetched & decoded. the external bus remains free for some time. while the processor internally executes the instruction. This time is utilized in 8086 to achieve the overlapped fetch & execution cycles.

* while the fetched instruction is executed internally the external bus is used to fetch the machine code of the next instruction. and arrange it in queue is called predecoded. inst. byte queue.
* It has 6 bytes long. first in first out structure.
* The inst. from the queue are taken for decoding sequentially. Once a byte is decoded the queue rearranged with next instruction.
* while the opcode is fetched by BIU, this excutes the previously decoded instruction. concurrently BIU along with the execution unit the forms a pipe line.
* The execution unit contain the register set of 8086. except the segment registers & IP. It has 16

bit ALU. able to perform arithmetic & logic operations. The 16-bit flag. registers reflects the results of execution by the ALU.

* The decoding units decodes the opcode bytes issued from the instruction byte queue.
* The timing and cu. derives the necessary control signals to execute the instruction opcode received from the queue.
* The execution unit may pass the results to the bus interface unit for storing them in memory.

Memory segmentation :

The memory of 8086 is organized as segmented memory. In this scheme the physically available memory is devided into a no. of logical segments. Each segment of size is 64 kB & it is addressed by using the segment reg.

The contents of the segment register actually points to the starting address of a particular segment. To add the specific memory location with in a segment, we need an offset address. The offset add. is also 16 bit long. So that it can be access 64 k memory location.

Consider an example, a colony contains 100 houses the simplest method of numbering the houses is assign the no. es from 1 to 100 to each house sequentially. Suppose one person wants to find the house no. 75 then he will start from house no. 1 & go on till he finds the house numbered 75.

Consider another case. where the 100 houses are arranged in the 10x10. pattern. In this case to find the house no 75 one will directly go to the 7th row & then to the 5th column. In the second scheme the effort req. is too less to finding the required house. The second scheme in our ex in analogous to segmented memory. scheme where the segment add. analogous to rows and offset add. analogous to column.

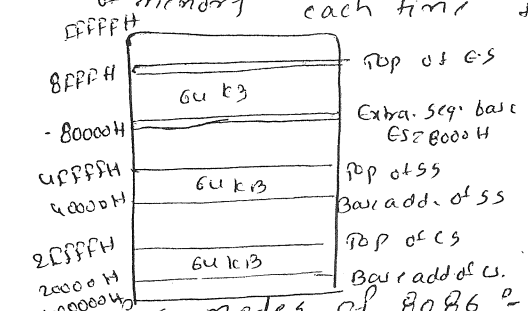
The 8086 processor. is able to add. 1MB of physical memory. The complete 1MB memory can be devided into 16 segments. each of size is 64 kB. The add. of the segments may be assigned as 0000 H to F000H. The offset add. values are from 0000H to FFFH so that the physical add. range from 00000H to FFFFF H

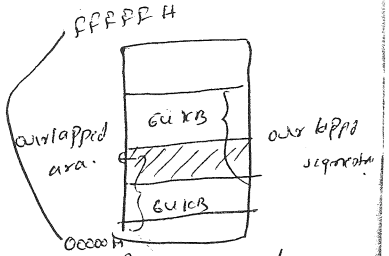
The segmentation is done in two ways one is non overlapping segment overlapped segmentation. In non overlapped. segmentation. If the one segment is ended then the other segment is initialized. In over lapped. segmentation. Suppose segment starts at a particular add. and its max. size can be 64 kB. But if another segment starts before this 64 k locat. of the first segment

* It allows the memory capacity to be 1MB

even though the actual add. to be handled of 16 bit size.

* It facilitates use of separate memory area for program. data & stacks. It is helpful to protect the data and code. * Provision of relocation is done. It permits a program (or) data to I/p into different areas of memory each time the pro. is executed





Addressing modes of 8086 :- addressing modes describes the types of operands & the way they are accessed for executing the instructions.

*According to the flow of instruction execution the inst. may be categorized as.

1) sequential control flow instruction
2) control transfer instruction

The sequential control flow instructions are the inst. which after execution transfer control to the next instruction appearing immediately after it is in the program.

Ex: the arithmetic logic data transfer and processor control instructions are sequential control flow instructions.

* The control transfer instructions are the inst. which after the execution transfer control to some predefined address.

or the address some low specified in the instr. for ex: INT. CALL. RET. JUMP.

The addressing modes for sequential control transfer instructions :-

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

MOV AX, 0005H.

direct :- In this direct addressing mode. 16 bit memory address is directly specified in the inst. as a part of it.

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 & content of DS as segment add. the effective add. here is 10H * DS + 5000H.

Register addressing mode :- In this register addressing mode, the data is stored in a register & it is referred using the particular register. all the registers. except IP may be used in this mode.

Ex: mov BX, AX.

Register Indirect addressing mode :- The address of the memory location which contains data (or) operand is determined in an indirect way. using offset registers.

mov AX, [BX]

Here data is present in a memory location in DS whose offset add. is in BX the effective address of the data is given as 10 * DS + (BX).

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

Ex: mov AX, [SI]

Register Relative addressing mode: 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 & DI in the default segment

Ex: mov AX, 50H[BX]

Based Indexed: The effective addressing of data is formed in this addressing mode by adding content of a base register to the content of an index reg. any one of SI or DI. The default segment reg. may be ES (or) DS.

Ex: mov AX, [BX][SI].

Relative Based Indexed: The effective address is formed by adding an 8 (or) 16 bit displacement with the sum of the contents of any one of the base registers and any one of the index registers in a default segment

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

For control transfer inst. the addressing modes depends upon whether the destination location is with in the same segment or a different one. It also depends upon the method of passing the destination add. to the process. Basically there are two addressing modes for the control transfer instructions. These are.

1. Inter segment addressing mode.
2. Intra segment addressing modes

If the location to which the control is to be transferred a different segment other than the current one the mode is called inter segment mode.

- Intersegment < direct, indirect
- Intra segment < direct, indirect.

Intrasegment direct: In the mode, the add. to which the control is to be transferred lies in the same segment. Here the add. is passed directly to the inst.

CALL 2020H.

Intra segment indirect: In this mode the add. to which the control to the transferred lies in the same segment in which control instructs lies. Here the the add. is passed to the instr. indirect

CALL [BX]

Intersegment direct: In this mode, the add. to which the control is to be transferred is in different segment. this addressing mode provides a means of branching from one code segment to another code segment. Here the destination add. are specified directly in the instr. CALL 2000H: 0050H

Intersegment indirect :- In this mode the add. to which the control is to be transferred is in different segment & it is passed to the instruction indirectly.

CALL [CS]:[BX]

Minimum mode 8086 :- The latches are generally offered o/p D-flipflops, like 74LS373. They are used for separating the valid address from multiplexed address / data signals & are controlled by the ALE signal generated by 8086.

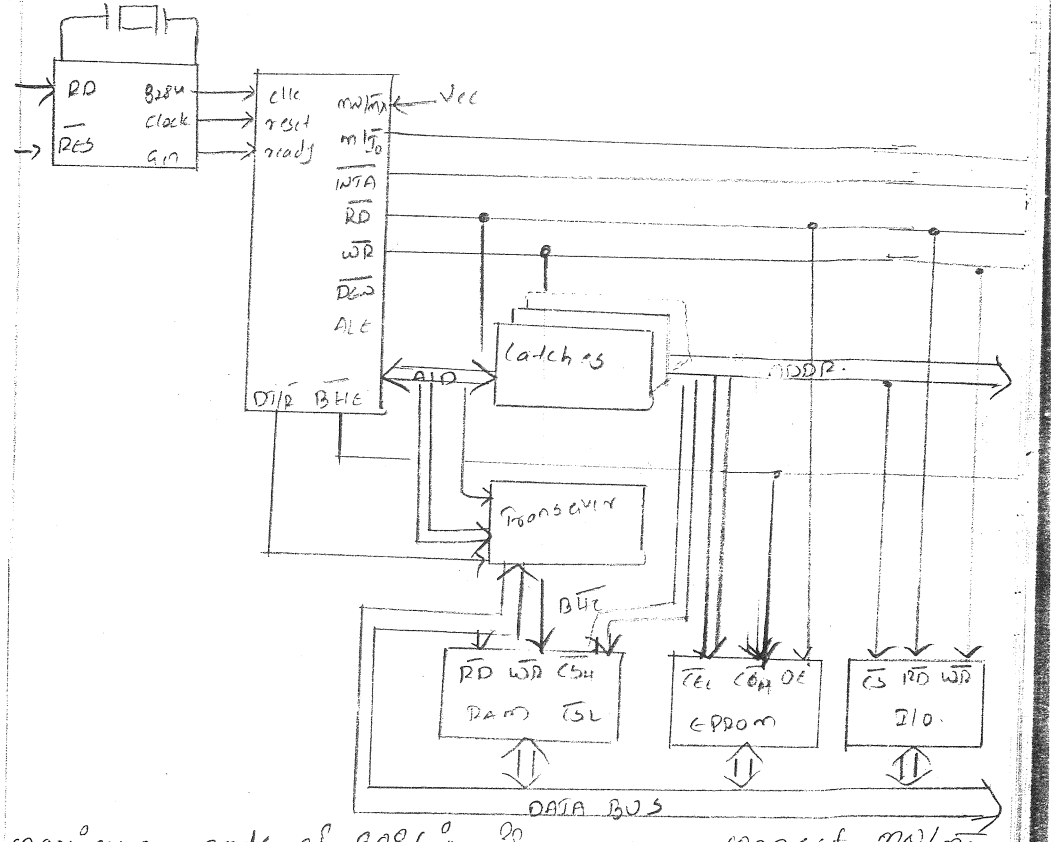
Transceivers are the bidirectional buffers and sometimes they are called as data amplifiers. they are used for separates the valid data from time multiplexed address / data signals. They are controlled by two signals $\overline{DEN}$ & DT/$\overline{R}$. The $\overline{DEN}$ signal indicates that the valid data is available on the data bus. & DT/$\overline{R}$ indicates the direction of data i.e from cpu to the processor.

Usually EPROMs are used for monitor storage and RAMs are user program storage.

a system may contain I/O devices for communication with the processor.

The clock generator generates the clock from the crystal oscillator The clock generator also synchronized some external signals with the system clock.

8086 has 20 address lines & 16 data lines. So 8086 CPU requires three octal address latches & two octal data buffers for the complete address & data seperation.



maximum mode of 8086 :- In we can connect MN/$\overline{MX}$ pin to the ground. 8086 operates in maximum mode.

In this mode there may be more than one microprocessor in the system configuration In this more the processor derives the status signals $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ another chip called bus controller derives the control signal using this status information The other components in the system are the same as in the minimum mode.

The basic functions of the bus controller is to derive the control signals like $\overline{RD}$, $\overline{WR}$, $\overline{DEN}$, DT/$\overline{R}$, ALE etc using the information made available by the processor on the status lines. The bus controller chip has i/p lines $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ and clk. These i/ps to 8288 are driven by CPU then it derives the o/ps ALE, $\overline{DEN}$, DT/$\overline{R}$, $\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$, and $\overline{AIOWC}$. The $\overline{AEN}$, IOB & CEN pins are specially useful for multiprocessor systems. $\overline{AEN}$ & IOB are generally grounded CEN is usually connected to +5V.

$\overline{IORC}$, $\overline{IOWC}$ are I/O read, I/O write commands respectively. These signals enable an I/O interface to read or write the data from or to the addressed port.

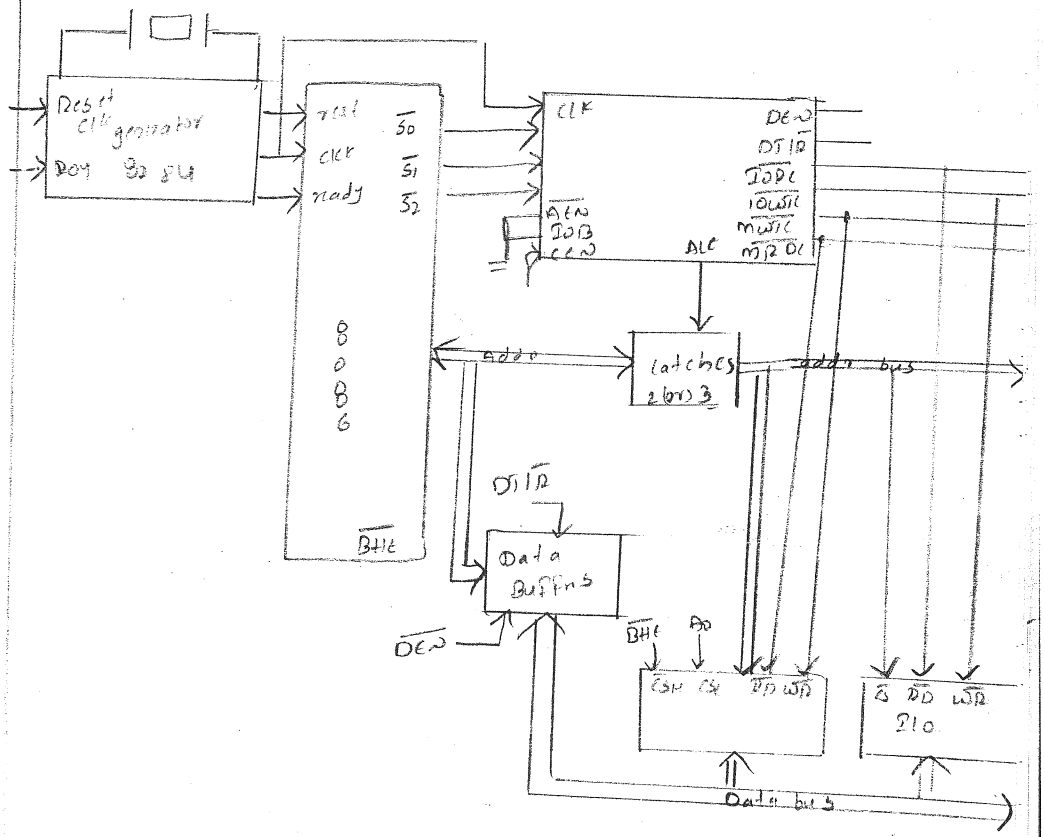
The $\overline{MRDC}$, $\overline{MWTC}$ are memory read command memory write command signals instruct the memory to accept or send data from or to the bus.

$\overline{AIOWC}$, $\overline{AMWC}$ are similar to $\overline{IOWC}$, $\overline{MWTC}$ except that they are activated one clock cycle earlier.

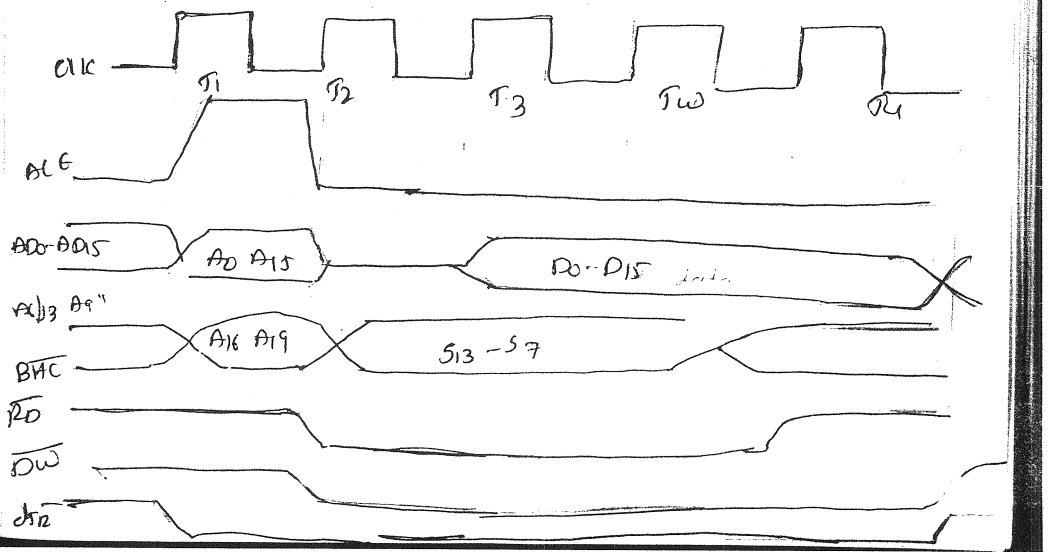
Note: $\overline{AEN}$ causes the 8288 to enable the memory control signal

$\overline{IOB}$ signal selects either I/O by mode in system bus mode operation.

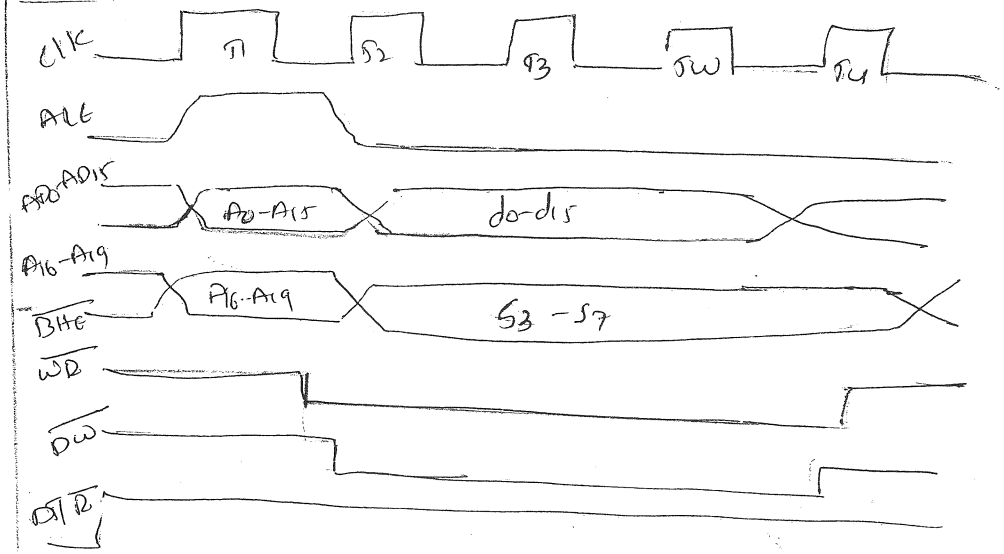
CEN i/p enables the command o/p pins on the



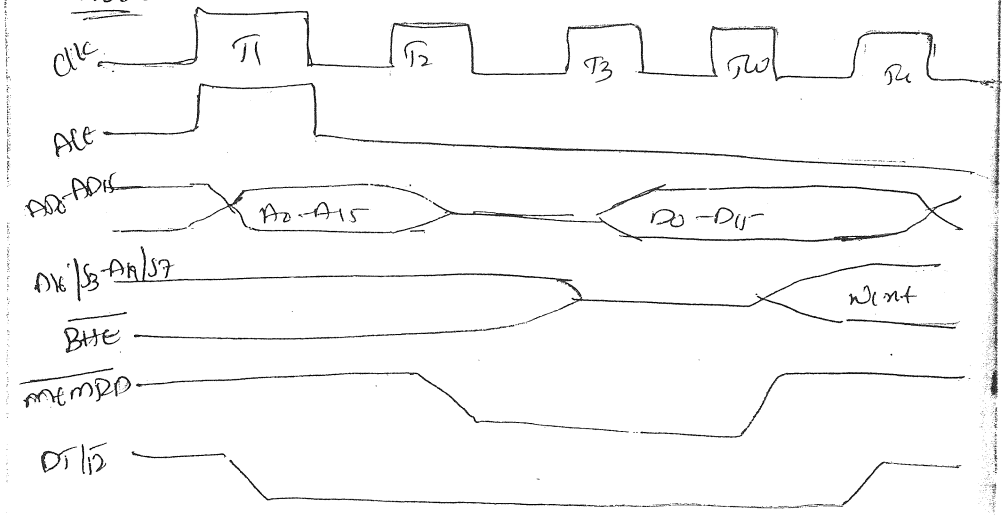
Timing diagram of minimum mode read operation.



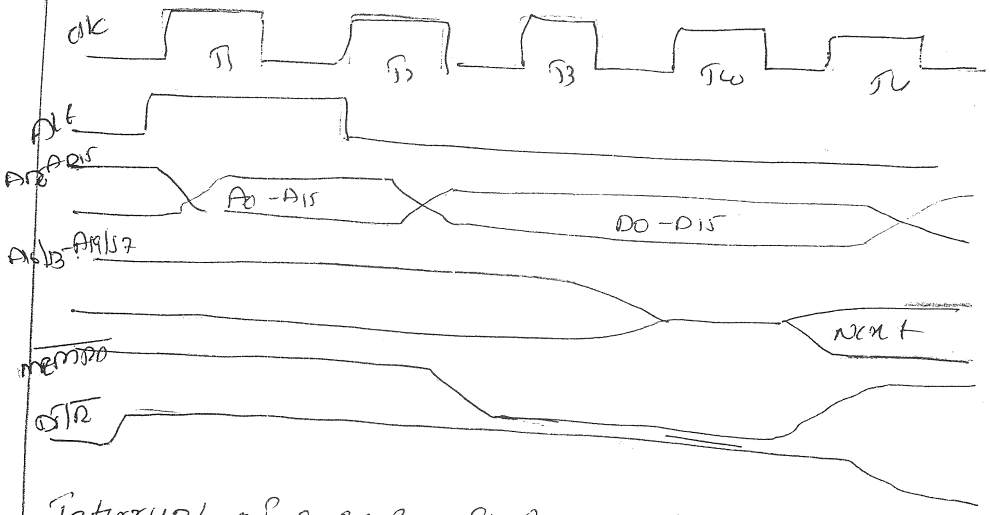
Timing diagram for write operation in minimum mode:



Timing diagram for read operation at maximum mode.



Timing diagram for write operation at maximum mode:



Interrupt of 8086:- It is a kind of signals received to processor either from software or hardware of connected source to stop the current execution in order to attached an urgent work

Interrupts

H/w Interrupt (2) → NMI (non maskable interrupt)

INTR (Interrupt request)

S/w (256) → INT0 H, INT1 H, ... INTFF H.

memory address:-

memory size → unit

The no. of address bits required to address 1kB memory is = 12 address bits

$2^{10} = 1024 = 1K$

$2^{11} = 2048 = 2K$

$2^{12} = 4096 = 4K$

inriment mode

↑ 4FFF H
:
4000 H

add. last no.

dicrriment mode

↓ 4000
3001 } 4K

Substract (last no)

Physical memory organization :

UNIT-2

Instruction set and assembly language programming of 8086

data copy / transfer instruction:

mov: This data transfer instruction transfer data from one register / memory location to another register / memory location. The source may be any one of the segment register or other general purpose reg. or special perpus reg. or a memory location and another reg. (or) memory location may act as destination

Ex:
```
mov AX, 5000H
mov AX, BX
mov AX, [BX]
```

Note: direct loading of the segment reg. with immediate data is not permitted. To load the segment reg. with immediate data. first we have to load any general perpose reg. with data and then it well have to be transfered to that particular segment register.

for ex:
```
mov DS, 5000H  → Not permitted

mov AX, 5000H
mov DS, AX     } → permitted.
```

PUSH: Thus instructions pushes the contents of the specified reg/ memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction.

Note: The higher byte is pushed first & then the lower byte.

Ex:
```
PUSH AX
PUSH DS
PUSH [5000H]
```
The contents of location 5000H & 5000LH in DS are pushed onto the stack.

POP: loads the specified register / memory location with the contents of the stack which is present in the stack memory. The stack pointer is incremented by 2.

Note: Pop instruction serves exactly opposite. to the PUSH instruction.

Ex:
```
Pop AX
Pop DS
Pop [5000H]
```

XCHG: This inst. exchanges the contents specified source and destination operands which may be. one of them may be memory location.

Note: Exchange of data contents. of two memory location is not permitted.

Ex:
```
XCHG [5000H], AX
XCHG BX   → This inst. exchanges the
```
data b/w AX & BX.

IN (Input the port): This inst. is used for reading the i/p port. The address of the

i/p port may be specified in the instruction directly or indirectly. AL & AX are the allowed destination for 8 bit or 16 bit i/p operations. DX is the only register implicit which is allowed to carry the port address.

Ex: IN AL, 3000H
This inst. reads the data from an 8-bit port whose address is 3000H and stores it in AL.

IN AX.
This inst. reads the data from an 16 bit port whose address is given by DX & stores it in AX.

OUT (o/p to the port): This instruction is used for writing to an o/p port. The address of the o/p port may be specified in the inst. directly or implicity.

Ex: OUT 3000H, AL.
This sends the data in AL to a port whose add. is 3000H.

OUT AX, This sends the data in AX to a port where address is specified implicitly in DX.

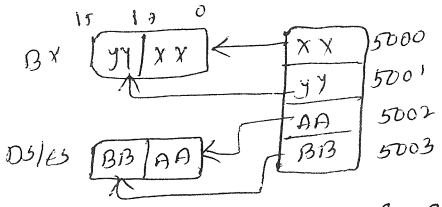
XLAT (Translate): The translate instruction is used for finding out the codes in case of code conversion problems.

XLAT inst. is used to translate to the code of the key pressed to the corresponding of segment code.

LEA (load effective address): This instruction loads the effect of an operand in the specified reg.

LDS/LES (load pointer to DS/ES): This instruction load DS/ES with pointer. loads the DS or ES register & the specified destination register in the instru. with the content of memory location specified as source in the instruction.

LDS BX, 5000H / LES BX, 5000H



LAHF load all from lower Byte of flag: This instruction loads the AH register with the lower byte of the flag reg. This instruction may be used to observe the status of all the condition code flags at a time.

SAHF store AH to lower byte of flag reg: This inst. sets or resets the condition code flags in the lower byte of the flag reg. depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set else it is reset.

POSHF Push flags to stack: This inst. pushes the flag reg. contents on to the stack. first the upper byte & then the lower byte will be pushed on to the stack. The SP is decremented by 2 for each operation.

POPF :- pop flags from stack :- This instruction loads the flag reg. completely from the stack. stack pointer is incremented by 2 for each operation.

arithmetic instructions :-

ADD :- This instruction adds as an immediate data or contents of a memory location specified in the instruction register. to the contents of another reg. or memory locations. Ex:- ADD AX, 0100H

ADC :- add with carry :- This inst. performs the same operation as add inst but adds the carry flag bit to the result. Ex:- ADC 0100H

INC :- It increase the contents of the specified reg. or memory location by 1

Ex: INC AX.

INC [BX]

DEC :- decrement :- It substracts 1 from the contents of the specified memory location (or) register

ex: DEC AX

SUB :- The sub instruction substracts the source operand from the destination operand and the result is left in the destination operand

SUB AX, 0100H

SBB: substract with Barrow :- The substract with barrow inst. sub the source operand and the borrow flag which may reflect be the previous calculation from the destination operan. SBB AX, 0100H

CBW convert signed Byte toword :- This instruction converts a signed byte to a signed word. It copies the sign bit of the byte to all the bits in the higher byte of the result word. It doesnot effect any flag.

CWD (convert signed word to double word) :- This instru. convert a signed word to a signed double word. It copies the sign bit of AX to all the bits of the DX reg. It doesnot effect any flag.

DIV (unsigned division) :- This instruction prforms unsigned division It divides an unsigned word (or) double word by a 16 bit (or) 8-bit operand The dividend must be in AX for 16 bit operation. the division specified using any one of the addr. mode. except immediate.

The result will be in AL while AH contains the remainder. If the result is too big to fit in AL. type '0' interrupt is generated.

In case of a double word dividend the higher word should be in DX & lower word should be in AX. The Quotient and the remainder in AX & DX resp. This inst. does not effect any flag.

IDIV (signed division) :- This inst. prforms same op. of DIV. but with signed operands.

CMP :- compare :- This instruction compares the source operand which may be register or immediate data.

Ex:- CMP BX, 0100H

AAA : ASCII adjust after addition :- It converts the resulting contents AL to unpacked decimal digits. after the addition AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD (0 to 9).

If it is b/w 0-9 & AF=0, AAA sets the 4 higher bits of AL to 0 & AH must be cleared before the addition

If it is b/w 0-9 & AF=1 06 is added to AL The upper 4 bits by AL are cleared & AH is incremented by one.

If it is greater than 9, then lower side is incremented by '06' & AH is incremented by '1'. AF & CF flags set to '1'

AAS ASCII adjust after substraction :- This converts the resulting contents of AL to unpacked decimal digits.

If the data lower 4 bits of AL reg. are greater than 9 if AF=1 the AL is decremented by 6 & AH reg. is decremented by 1. the CF & AF are set to 1. Otherwise the CR & AF Set to '0' & the upper nibble of AL is '0' & the lower nibble may be any from 0 to 9. AH is modified as difference of the previous contents of AH.

AAM :- this instruction after execution converts the product available in AL into unpacked BCD format This follows a multiplication instruction. for AL=5D. AAM instruction will from unpacked BCD result. In AX D is greater than 9. So add 06 to it D+6 = 13 H. LSD of 13 H is the lower unpacked byte. For the result. '1' is added to '5' So 5+1=6 will be the upper unpacked byte of the result. Thus after execution AH=06 & AL=03.

AAD :- ASCII adjust after division ADD instruction converts two unpacked BCD digits in AH & AL to the equivalent binary number. in AL. This adjustment must be made before the dividing the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OR are unmodified.

Ex :- AX | 05 | 08. AX contains 0508 unpacked BCD for 58 decimal

AAD result in AL | 8 | 3A 58D = 3AH in AL.

DAA decimal adjust accumulator :- This instr. is used to convert the result of the addition of two packed BCD no to a valid BCD no. The result has to be only in AL.

If the lower nibble is greater than 9. after addition or if AF is set. It will add 06 to the lower nibble in AL after adding 06 to the lower nibble of AL. If the upper nibble

of AL is greater than 9 or if carry flag set, DAA instruction adds 60H to AL.

AL=53 CL=29

ADD AL, CL.

AL ← AL+CL

AL ← 07C

② AL=73, CL=29

ADD, AL, CL AL ← AL+CL 9C
AL ← 9C. 6.
AL
60
DAA AL ← 02 & CF=1 CF=1 02

This instruction affects AF, CF, PF, & ZF the OF is undefined.

DAS decimal adjust after instruction :- This instn. converts the result of subtraction of two packed BCD numbers to a valid BCD no. The subtraction has to be in AL only. If the lower nibble of AL is greater than 9. this instruction will substract 06 from lower nibble of AL. If the result of the subtraction sets the carry flag (or) if upper nibble is greater than 9. it subtracts 60 from AL.

① AL 75, BH=28.

ADD AL, BH ; AL=2F

DAS AL ← 29

② AL=38. CL=61

SUB AL, CL+1

ALF=D7 OF=1

AL ← 77

CF=1

logical instructions :-

AND :- This inst. bit ANDs the source operand that may be an immediate a register or memory location. to the destination operand that may be register or memory location. The results stored in the destination operand.

Ex: AND AX, 0008 H

OR :- The OR inst. carries out the OR operation. in the same way as described in case of the AND operation. The limitations on source and destinations operands are also same as in case of AND operation.

Ex :- OR AX, 0098 H.

NOT logical :- The not instructions complements the contents of an operand register or a memory location. bit by bit.

Ex: Not A X

NOT [5000 H]

XOR :- The XOR operation is again carried out in a similar way to the AND and OR operation. The constrains on the operands are also similar. It give high o/p when the 2 i/p bits are dis similar. Ex:- XOR AX, 0098 H.

TEST :- The test instruction performs a bit by bit logical and operation on the two operands Each bit of the result is then set to 1.

if the corresponding bits of both operands are 1. else the bit is reset to 0.

Ex:- TEST AX, BX.

Shift and rotate instructions :-

SHL/SAL :- Shift logical left / shift arithmetic left. It is used to perform shift the contains of the operand towards left side bit by bit. The count specified in the instruction as one or CL newly use position at LSB side. inserted with zeros with the shift operation.

SHL AX, 01H

if AX = 4545 H

MSB

CY 0 100 0101 0100 0101 LSB

0 1000 1010 1000 1010 ← inserted with 0

SHR :- Shift logical right operation :- It is used to shift the operands of the right side bit by bit in the shift operation LSB bit goes to carry flag newly vacant position at MSB is inserted with zero. and the count is specified as one. (or) CL/CH.

SHR AX 01 H

AX = 4545 H

MSB LSB

0 100 → 0101 0100 0101 CY

0 0 1 0 → 0 0 1 0 1 0 1 0 0 0 1 0 1

inserted.

SAR :- Shift arithmetic right :- This inst. performs right shift on the operand word or byte, that may be a register or a memory location of the specified count in the instruction.

Ent mov CL 02H

SAR AX, CL

AX = C545 H

1100 0101 0100 0101 CY

Inserted 1 1 1 1 0 0 0 1 0 1 0 0 0 1 0 1

1 1 1 0 0 0 1 0 1 0 0 0 1 0 1 0

Inserted

ROR :- Rotate without carry :- Rotate right without without carry is used rotate right without carry. the LSB goes to carry bag and LSB now vacant is the MSB.

ROR AX 01 H

AX = 4576

MSB LSB CY

0 100 0101 0111 0110 ×

0 0 1 0 0 0 1 0 1 0 1 1 1 0 1 1 0

ROL :- Rotate left without carry :- It used towards with out carry. rotate the process the MSB bit goes to carry bag. simultaneously goes to the newly vacant at LSB the count as above specified. ROL AX, 01 H

AX = 5575 H

CY MSB LSB

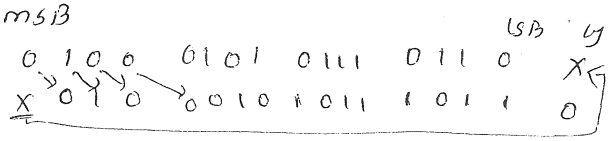
0 ← 0101 0101 0111 0101

1010 1010 1100 1010 ← 0 → vacant

RCR :- Rotate with carry :- It rotates the contents of the destination operand right by the specified count through carry flag.

ROR AX, 01, H

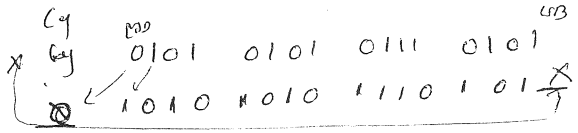
if AX = 0576 H



RCL :- Rotate left with carry :- It is used to left by the specified count through carry flag.

RCL AX, 01 H

AX = 5575



control transfer (or) Branching instructions :-

control transfer inst. transfer the flow of execution of the program to a new address specified in the instruction directly (or) indirectly.

① unconditional control transfer instructions :-

In case of unconditional control transfer instr. the execution control is transferred to specified location independent of any status (or) condition.

CALL :- This instruction is used to call a subroutine from a main program.

RET :- At each call inst. the IP & CS of the next instruction is pushed on to the stack before the control is transferred to the procedure at the end of two procedure. the RET substraction must be executed, when it is executed. the previously stored content of IP & CS along with flags are retrieved into the CS, IP & flag reg from the stack & the execution of the main program continues further.

INTN :- Interrupt type N. In 8086 256 interrupt are defined corresponding to the types of from 00H to FFH.

When an INT N inst. is executed. the type byte N is multiplied by 4 & this values gives a offset address & 0000 as segment address, which contains the IP & CS values of the interrupt service routine.

INTO :- Interrupt on over flow :- when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000 : 0010 as explained in INT type instruction. This is equivalent to a type 4 interrupt instruction.

Jmp :- This instruction unconditionally transfers the control of execution to the specified address using 8 bit (or) 16 bit displacement.

Loop :- This instruction executes the part of the program from the lable (or) address specified

on the instruction up to the loop instruction. CX no. of times at each iteration. CX is decremented automatically.

IRET :- when an interrupt service routine is to be called. before transferring control to it the IP, CS and flag register are stored onto the stack to indicate the location from where the execution is to be continued after the ISR is executed.

Conditional control transfer instructions :- In the conditional control transfer instruction the execution control is transferred to specified location provided the results of the previous operation satisfies the particular condition. other wise the execution continues in normal flow sequence.

JZ/JE	label	transfer execution control add. label if ZF=1
JNZ/JNE	"	" " " " " ZF=0
JS	"	" " " " " SF=1
JNS	"	" " " " " SF=0
JO	"	" " " " " OF=1
JNO	"	" " " " " OF=0
JP/JPE	"	" " " " " PF=1
JNP	"	" " " " " PF=0
JB/JNAE	"	" " " " " CF=1
JNB	"	" " " " " CF=0
JBE/JNA	label	transfer execution control add label CF=0 & ZF=1
JNBE	"	" " " " " CF=0 & ZF=0
JL	"	" " " " " SF=1 neither nor OF=1
JNL	"	" " " " " OF=0 neither nor SF=0.
JLE	"	" " " " " neither ZF=1 & nor OF=1
JNLE	"	" " " " " LE ZF=0 nor

Flag manipulation and processor control Inst :- SF=1 or OF=1

These instructions control the functioning of the available H/w inside the processor chip.

CLC	clear carry flag	cy=0
STC	set carry flag	cy=1
CMC	complement carry flag	cy=cy'
STI	set Interrupt flag.	I=1
CLI	clear " "	I=0
CLD	clear directional flag.	DF=0
STD	set directional	DF=1

machine control instructions :-

WAIT :- It is used executed when the processor is till $\overline{test}$ becomes low.

NOP :- No operation the processor can't perform any op except incrementing Inst. pointer

ESC :- when this Inst. is used to the processor pass the bus to the other bus master.

HLT :- Enter into the Halt state. It is executed to enter into the Halt.

LOCK :- When this is executed forward bus access to the other bus master.

loop instruction :- when this is executed to part of the program from the label or address specified in the inst. upto the loop instruction is repeated CX no. of times.

```
mov CL, 04H
mov AL, 10H
mov BL, 20H
go ADD AL,BL
  loop go
  int 3H
```

```
AL←30
→CL=3
AL←50
CL=2
AL← 70
  ←CL=1
AL←90
  CL=0
```

loope / loop2 : loop while ZF=1.

```
mov CL, AL
mov AL, 10H
mov BL, 20H
go: ADD AL,BL
    loop3 go
    int 3H
```

O/P 30

LOOPNE / LOOPNZ :-

```
ZF=0.
mov CL, AL
mov AL, 10H
mov BL, 20H
go: add AL,BL
    loopNZ go
    INT 3H
```

O/P 90

String manipulation instructions :-

REP :- Prefix to some other inst. this instruction is executed the inst. to which the prefix REP is provided is repeated is CX no. of times.

MOVSB / MOVSW :- move string byte or move string word. It is used to transfer a string bytes / words from source location to destination the source string is pointed by DS:SI reg. pair and the destination string is pointed by ES:DI reg. pair that source string starting memory location is SI reg. so that must be store in data segment and the destination string storing memory location DI so the must be stored in extra segment. String length is stored in CH reg. and to repeat this transfer of a byte source to destination. we repeat fix. reg. we used the repeat instn.

Ex:

```
mov AX, 5000 H
mov DS, AX
mov AX, 6000 H
mov ES, AX
mov CX, 0FFH
mov SI, 1000 H
mov DI, 2000H
CLD
REP MOVSB
```

CMPSB :- Compared string byte- It is used to compare the two strings one string is pointed by DS: and other string is pointed ES and the length of the string is CX reg. If ZF=1 two strings are equal. i.e zero flag is zero the two strings are not equal.

```
mov AX, SEG1
mov DS, AX
mov AX, SEG2
mov ES, AX
mov SI, offset str1
mov DI, offset str2
mov CX, 0101H
CLD
REPE CMPSW.
```

SCASB :- It is used scan a string from operand having byte (or) word in ALR or AX and the string is loaded into ES:DI pair.

If Z=1 indicates that the byte or word is present in a given string.

Z=0 is indicates the byte or word not present in a given string.

```
Ex: mov AX, SEG
    mov ES, AX
    mov DI, offset
    mov CX, 0101H
    mov AX, word
    CLD
    REPNE SCASW.
```

LODSB :- load string byte:- It is used to loads the reg. AL AX from the string memory location. pointed by DS:SI pair. the contain of AL or AX into string memory location pointed by ES:DI In all string is controlled by direction flag. If D=0 string is. parchue lower add. to higher add. enabled the auto increment mode. If D=1 string is transfer higher address to lower address. enabled the auto direment load.

STOS :- store string :- The STOS inst. stores the AL/AX register contents to a location in the string pointed by ES:DI register pair. The DI is modified accordingly. No flags are affected by this instructions.

Assembler directives :-

DB:- define Byte :- The dB directive is used to reserve byte or bytes of memory location in the available memory.

DW:- define word:- The DW directive serves the same purpose as the dB directive. but it now makes the assembler reserve the no. of memory words instead of bytes.

DQ :- define Quadword :- It is used to direct the assembler to reserve 4 words of memory for the specified variable.

DT define Ten bytes :- The DT values directs the assembler to define the specified variable requiring 10-bytes for its storage and initialse the 10-bytes with specified values.

Assume :- The assume directive is used to inform the assembler the names of the logical segment to be assumed for different segments used in the prog. For example : code segment may be given the name code. data seg. may given the name data etc : The statemen.

assume cs: code directs the assembler that the machine codes are available in a segment named code & hence the cs reg. is to be loaded with the add. alloted by the os for the lable code while loading.

END :- The end directive marks the end of an assembly language prog. when the assembler comes across the END directive. It ignores the source lines available later on. Hence it should be ensured that the end statement should be the last statement in file.

ENDP :- end of procedure. In assembly language prog. the subroutines are called Procedures. Thus Procedures may be indipedent program modules which return particular results or values to the calling programs.

The endp directive is used to indicate the end of procedure.

Sytax :- Procedure star

:

star endp

ENDS :- end of the segment This directive marks the end of logical segment. The names appear with the ENDS directive as prefix as to mark the end of these particular segments.

data segment.

:

data ends.

The above structure represents data segment. The data related to the program must the b/w data segment & data ends

EVEN :- align on even memory add.

The even directive updates the location counter to the next even add. If the current location counter contents are not even & assigns the following routine to that add.

even
proedure ROOT
:
ROOT endP.

The above structure shows a proiedure ROOT that is to be aligned at an even address. where the assembler comes across the directive even it checks the content of the location counter if it is odd, it is updated to the next even value & then the ROOT Procedure is assigned to that add.

If the content of the location counter is already even then the ROOT Proiedure will be assigned with same add.

EQU : equate. the directive EQU is used to assign a lable with a value or symbol. using EQU directive even an instruction mnemonic can be assigned with a lable & the lable can then be used in the program in plaie of that mnemonic.

lable EQU 0500H
Addition EQU ADD

The first statement assigns the location 500H with the lable and the second statement assigns another lable addition with mnemonic ADD.

GROUP : This directive is used to inform the assembler to form a logical group of the following segment names.

The assembler passes the information to the linker form the code such that the group declared segments must be with in a 64kB memory segment. Thus all such segments lables can be addressed using the same segment base.

Program Group code, Data, stack.

The above statment directs the loader linker. prepare an exe file such that CODE DATA STACK segment must lie with in a 64k byte memory segment. that is named as program. now for assume statment use lable program rather than code, data stack

local : The lables, variables, constants or proiedure declard local in a module are to be used only that module.

local a:b, DATA.

NAME : The name directive is used to assign a name to an assembly language program module.

offset : when the assembler comes across the offset operator along with lable. it fetch the comp. 16-bit displacement of the particular label & replaces the string offset label by the

computed displacement.

ORG :- ORG directive directs the assembler to start the memory allotment for the particular segment block for code from the declared address in the ORG statements

If the ORG statment is not written in the program, memory is initialised from 0000 if ORG 200H statments is present at the starting of the code segment of that module then the code will start from 200H add.

PROC Procedure :- The PROC directive marks the start of the named procedure in the statmn also the types Near or FAR specify the type of the procedure. For ex. the statment Result PROC NEAR marks the start of a routine Result which is to be called by a prog located in the same segment of memory. The statment Result PRO FAR marks the mark of a routine Result, which is to be called by a program located in diffrn segments of mimory

```
Result   PROC  NEAR
Result   PRO   FAR.
```

EXTRN :- The directive EXTRN informs the assembler that the names procedure & labls declared after the directive have already been defined in some other assembly language modules while in the other module where the names procedur & labls actually appear they must be declard public using Public directive.

Ex:-
```
module 1   segment
Public     factorial FAR
module 1   ENDS
module 2   segment
EXTRN      factorial FAR
module 2   ENDS.
```

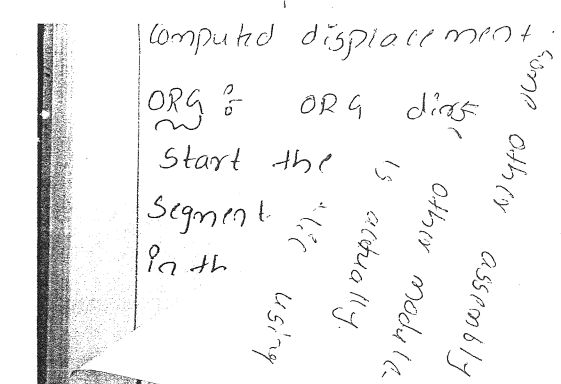
PTR :- The pointer operation is used to declare the type of a lable variable (or) memory operand The operation PTR is prefixed by either byte or word. If the prefix is Byte then the particular lable & variable (or) memory operand is treated as 8 bit quantity, while if WORD is the prefix then it is treated as 16 bit quantity.

```
mov AL, Byte PTR [SI]
INC Byte PTR [BX]
```

Public :- This directive informs the assembler that the lables variables const. in procedure declared Public many be accessed by other assembly modules to form their codes But while using the public declared lables, variables const. procedure the user must declare them

computed displacement. in externals using EXTRN

ORG :- ORG directive is used to decide the Start the segment. its is used to decide the In the variable, label (or) procedure. base address in place of



MOV DS, AX

segment :- The segment directive marks the starting of the logical segment.

In some cases the segment may be assigned type like public or global.

Eg: exe. code segment global

:

exe. code ENDS.

SHORT :- The short operator indicates to the assembler that only one byte is req. to code the displacement for a jump. otherwise the assembler may reserve two bytes for the displacement

Jmp short label.

Global :- The label variables const. declared (procedure.) global may be used by other modules of the prog. one a variable is declared global. it can be used by any module in the program.

Type :- the type operator directs the assembler to decides the data type of the specified label & replaces the type label by the decided data type.

for word type variable the data type is 2 for double word type it is 4 & for byte type it is 1. suppose string is a word array. the inst. mov AX types string moves the values 0002H.

MACROS :- suppose no. of inst are appearing again and again in the main program. then the program becomes lengthy so a label is assigned with the repeatedly appearing string of inst. The process a assigning a label. (or) macro name to the string is called defining macro.

The difference b/w macro and a subroutine that, in case of macro the completely code of the instructions string is inserted at each place where the macro name appears. hence exe. file becomes lengthy. macro doesn't utilise the stack service.

On the other hand, subroutine is called whenever necessary then the control of execution is transferred to the subroutine, every time it is called The executable code in case of the subroutines becomes smaller as the

subroutine appears only once in the complete code.

Thus the exe. file is smaller as compared to the prog. using macro.

Subroutine utilizes the stack struct. The program using subroutine requires less memory space for execution than that using macro. macro requires less time for execution. as it doesnot contain call and RET inst. as subroutines do.

* A macro can be defined any where in a prog using the directive macro and ENDM. the label paid to macro is the macro name. which should be used in the actual prog. The ENDM directive marks the end of the inst. sequence. assigned with macro name.

```
display macro
mov AX, seg. msg
mov DS, AX
mov DX, offset msg
mov AX, 09H
INT 21H
```

display to the inst. sequence. b/w the directive macro & ENDM while assembly the above sequence of inst. will replace the label display. whenever it appear in the prog

The macro may also be used in a data segment. string macro.

```
msg1 DB 0AH, 0DH Prog. terminated normally 0AH
0DH, $ : msg2 DB 0DH Retry, abort fail.
0AH, 0DH, $ ENDM.
```

A macro may be called by quoting its name along with any values to be passed to the macro.

passing parameters to a macro :-

using parameters in a definition the programmer specifies the parameter of the macro. those are likely to be changed each time the macro is called.

```
Display macro msg
mov AX, SEG msg
mov DS, AX
mov DX, offset msg
mov AH, 09H
INT 21H
ENDM
```

This parameter msg can be replaced by msg1 while calling the macro as shown

```
display msg1
display msg2.
msg1 DB 0A, 0DH Prog terminated normally
msg2 DB 0A, 0DH Retry. Abort fail.
```

machine language.	assembly language.
binary coded.	mnemonic code.
less memory	less memory.
less execution time	less execution time.
difficult	simple.

assembly language programs :-

addition 8-bit.

```
Data segment.
a db 0AH
b db 72H
Data endd
code segment
assume cs: code : ds: data
start: mov AX, data
       mov DS, AX
       mov AL, a
       mov BL, b
       add AL, BL
       int 3H
       code ends
end start
```

O/P. AL: 0000 BL = 0000
AL 0AH BL = 00
0A H = 72H
76 H = 72H

adding two 16-bit data

```
data segment
a dw 1212H
b dw 5678H
data endd
code segment
assume cs: code, ds: data
start
       mov AX, data
       mov DS, AX
       mov AX, a
       mov BX, b
       add AX, BX
       int 3H
       code end
end start
```

AX: 0000 BX: 0000
= 1212H = 5678H
= 1212H = 5678H
= 8778H =

subtraction: (simply replace "ADD" with "SUB")

multiplication :- two 8-bit :-

```
data segment.
a db 01H
a db 02H
data endd
code segment
assume cs: code ds: data
start: mov AX, data
       mov DS, AX
       mov AL, a
       mov BL, b
       mov BC
       int 3H
       code endd
       end start.
```

O/P: AX=0001H BX=0000H
AX = 0001H BX = 0002H
AX = 1002H BX = 0002H

multiplication of two 16-bit :-

```
data segment
a dw 8126H
b dw 2102H
data endd
code segment
assume cs: code DS: data
start: mov AX, data
       mov DS, ax
       mov AX, a
       mov BX, b
       mul BX
       int 3H
endstart. code endd
```

O/P: AX=8126H BX=0000H
DX=0000H

AX: 8126H BX: 2102H DX: 0000H
AX: CUCC BX: 2102H DX: 11074H

division of 8-bit data :-

```
data segment
a db 97H
b db 10H
date endd
code segment
assume cs: code. ds: data.
start:
        mov Ax. data
        mov DS, AX
        mov Al, a
        mov AH, 00H
        mov BL, b
        DIV BL
        int 3H
code end
endstart.
```

Quotient : AL.
remainder AH

O/P DX : 0097H BX : 0000H
AX : 0097H BX : 0010H
AX : 0709H BX : 0010H

division of two 16 bit data :-

```
data segment
a dd 10155123 H
b dw 1000 H
data end
code segment
assume cs: code. Ds: data
start  mov AX. data
       mov DS, AX
       mov AX, [55]
       mov DX, [55+2]
       mov Bx, b
       div BX
       int 3H
code endd
end start.
```

O/P :- AX = 5123 BX: 0000 H DX: 0000
AX : 5123H BX: 0000 H DX: 1015H
A : 5123H BX: 1000H DX: 1015H

Quet : AX
Remainder : DX

conversion of Packed BCD to ASCII :-

```
data segment
a db 36 H
count EQU 0U H
data end
code segment.
assume cs: code ds: data.
start: mov AX. data
       mov DS, AX          ; CL = 0UH
       mov CL, count       ; AL = 36 H   0011 0110
       mov AL, a.          ; AL = 03 H   0000 0011
       SHR AL, CL.         BH = 03H
       mov BH, AL.         AL = 36 H     0011 0110
       mov AL, a           AL = 60H      0110 0000
       mov BL, AL.         AL = 06       0000 0110
       SHR AL, CL.         BL = 06       0000 0110
       ROR AL, CL.
       XOR BX, 3030H       BX ; 3336
       int 3H
code endd
end start
```

conversion of ASCII to BCD :-

```
data segment
a dw 3936 H
count EQU 0UH
dataendd
code segment
assume cs: CODE DS: data.
```

```
start : mov  AX,data.
        mov DS, AX
        mov AX, a.          ;AX: 39 36
        SHL AH, CL.         AH: 90
        mov BL, AH          BL : 90
        mov AH, 00H
        SHL AL, CL.         AL : 60
        ROR AL, CL.         AL: 06
        XOR AL, BL.         AL: 96
        int 3H
        code endd
endstart
```

Classification of no of +ve and -ve no ?

```
data segment.
list dw  2579H ,8605 1H ,50791H ,0890H, 6107H
count EQU 05H
data end
code segment
assume CS: code. DS: data.
start : mov AX, data.
        mov DS, AX
        mov CL, count
        mov SI, offset list
AGAIN :  mov AX, [SI]
        SHL  AX, 01
        JC  L1
        INC  BX.
        JMP  NEXT
L1 : JNC DX.
NEXT :  ADD SI, 02H
        DEL . CL.
    JNZ  again
    int 3H
    code endd
 end starts.
```

o/p : BX: 0004
DX = 0001

SORTING ?

Ascending order ?

Ex: 53 25 19 02 (n = 4)

25	53	19	02	1
25	19	53	02	2
25	19	02	53	3
				4.

} op. 1, comparision 3

=) n = 3.

largest no.

19	25	02	53
19	02	25	53

} operation 2, compare = 2.

=) n = 2.

largest no.

02	19	25	53

} operation 3 compare = 1

largest no.

So, comparision = comp - 1 = 3.

operation = count - 1 = 3

program :

```
Data segment
list dw  53H, 25H, 19H, 02H
count EQU 04
data ends
assume CS: code  DS: data
code segment.
start :  mov AX, data.
```

```
        mov DS, AX
        mov DX, count-1
again:  mov CX, dx
        mov SI, offset list
again1: mov AX, [SI]
        cmp AX, [SI+2]
        JL  PR1
        XCHG [SI+2], [AX]
        XCHG [SI], [AX]
PR1:    ADD SI, 02
        loop again1
        DEC DX
        JNZ again0
        int 3H
code endd
end start.
```

Identification of even or odd :

```
data segment
list dw 2357H, 0A579H, 0C322H, 0C91EH, 0C000H,
        0957H.
count   EQU 006H
data ends
code segment
assume: CS: code DS: data
start:  XOR BX, BX
        XOR DX, DX
        mov AX, data
        mov DS, AX
        mov CL, count
        mov SI, offset list
again:  mov AX, [SI]
        ROR AX, 01
        JC  odd
        INC BX
        jmp next
ODD:    INC DX
NEXT:   ADD SI, 02
        DEC CL
        JNZ again
        int 3H
code endd
end start.
```

Introduction to PTR : The pointer operator is an directive used to declare the type of label variable or memory operand. The operator PTR is prefixed by either Byte or word. If the prefix is Byte/word then the particular level variable or memory operand is treated as an 8-bit / 16 bit quantity.

Ex: mov AL, Byte PTR [SI]

An alp to add two multibyte no's

```
data segment
a dB 05, 5AH, 6CH, 55H, 66H, 77H
b dB 0A, 5BH, 0AH, 57H, 32H, 19H
c dB 09H DUP(000)
count equ.
data end
code segment.
assume: CS: code. DS: data.
start : mov AX. data
        mov DS, AX
        mov CX, count
        mov SI, offset a
        mov DI, offset b
        mov BX, offset c
        XOR AX, AX
again : mov AL, [SI]
        ADC AL, [DI]
        mov Byte PTR [BX], AL
        INC SI
        INC DI
        INC BX
        DEC CX
        JNE again
        JNC exit
        mov byte PTR [BX], 01
exit : int 3H
    code endd
    end start.
```

An alp for addition of two 3x3 matrices

```
data segment
MAT1 DB 01H. 02. 03, 0A, 05, 06. 07. 08 09 H
MAT2. DB 01, 02, 03. 0A. 05, 06 07H 08H 09H.
RMAT3 DW 09H DUP(?)
count EQU 09H
data end
code segment
assume cs: code, ds: data
start: mov AX. data
       mov ds. AX
       mov CX, count
       mov SI, offset MAT1
       mov DI, offset MAT2
       mov BX. offset RMAT3
Next : mov. AL [SI]
       mov AL. [DI]
       mov word PTR [BX], AX
       INC SI
       INC DI
       ADD BX, 02.
       loop Next
       code endd
end start.
```

An alp for finding out the largest no. from given unordered array :-

```
assume cs: code . ds: data.
data segment
list dB 52H , 23H, 56H , 45H.
count EQU 0FH
largest DB 01H DUP(?)
```

An alp for finding length of a string :-

```
data segment
adh "ECE A", "$"
data end
code segment
assume cs: code ds: data.
start: mov AX.data
       mov DS, AX
       mov BL.00H
       mov SI. offset a
       mov AL.[SI]
again: cmp AL.$.
       JZ found
       INC BL
       INC SI
       Jmp again
found : Int 3H
code endd
end start.
```

An alp for reverse a string :-

```
data segment
a db "ABCDE".$'
strlen EQU 05H
data endd
code segment.
assume cs: code. DS: data
start : mov AX.data
        mov DS, AX
        mov SI offset a
        mov DI offset b
        add DI. strlen - 1
again: mov AL.[SI]
       XCHG AL [DI]
       mov [SI], AL.
       INC SI
       DEC DI
       Cmp SI, DI
       JB again
       Int 3H
code endd
end start.
```

An alp for finding the length of string :-

```
data segment
str db "ECE B". "$".
```

```
data end
code segment
assume cs: code, ds: data
start : mov ax, data
        mov ds, ax
        mov bl, 00H
        mov si, offset str
again : mov al, [si]
        cmp al, '$'
        jz found
        inc bl
        inc si
        jmp again
found: int 3H
       code endd
end start.
```