Submission Date ... Feb 2017 CSE-A+B 6 Sem

Assignment No.-2 up & mc

Q No. 1(a) State the function of AAA & AAD instruction of 8086 up

(b) What is memory segmentation? How memory segmentation is achieved in the 8086 up? State the advantage of memory segmentation.

(c) Why 8086 memory is mapped into banks? What logic levels are found with BHE & A_0 when 8086 reads a word from the address A00AH?

Q No. 2(a) Describe memory mapped I/O. State its advantage over I/O mapped I/O.

(b) Draw the timing diagram of memory read in maximum mode configuration of 8086.

Q No. 3 (a) Write a program for 8086 to arrange numbers of a series of 10 elements in ascending order.

(b) Given that BX = 6370H, SI = 2A9BH, disp. = 102H. Determine the effective address resulting from these register & the addressing mode

① immediate ② Direct ③ Register indirect using BX
④ Relative Base Index ⑤ Register relative using BX.

Q No 4 (a) Explain LEA instruction of 8086. Explain how it differs from MOV instruction with a e.g.

(b) Explain the following with eg.

① OFFSET ② RANE SCASW ③ LDS ④ ROR
⑤ CBW ⑥ PTR ⑦ DUP ⑧ ?

Q No 5 (a) Explain IVT in detail.

(b) Interface four chip. of 4K RAM with 8086 up. Give the Complete memory map of the system.

Features of 8086 µp

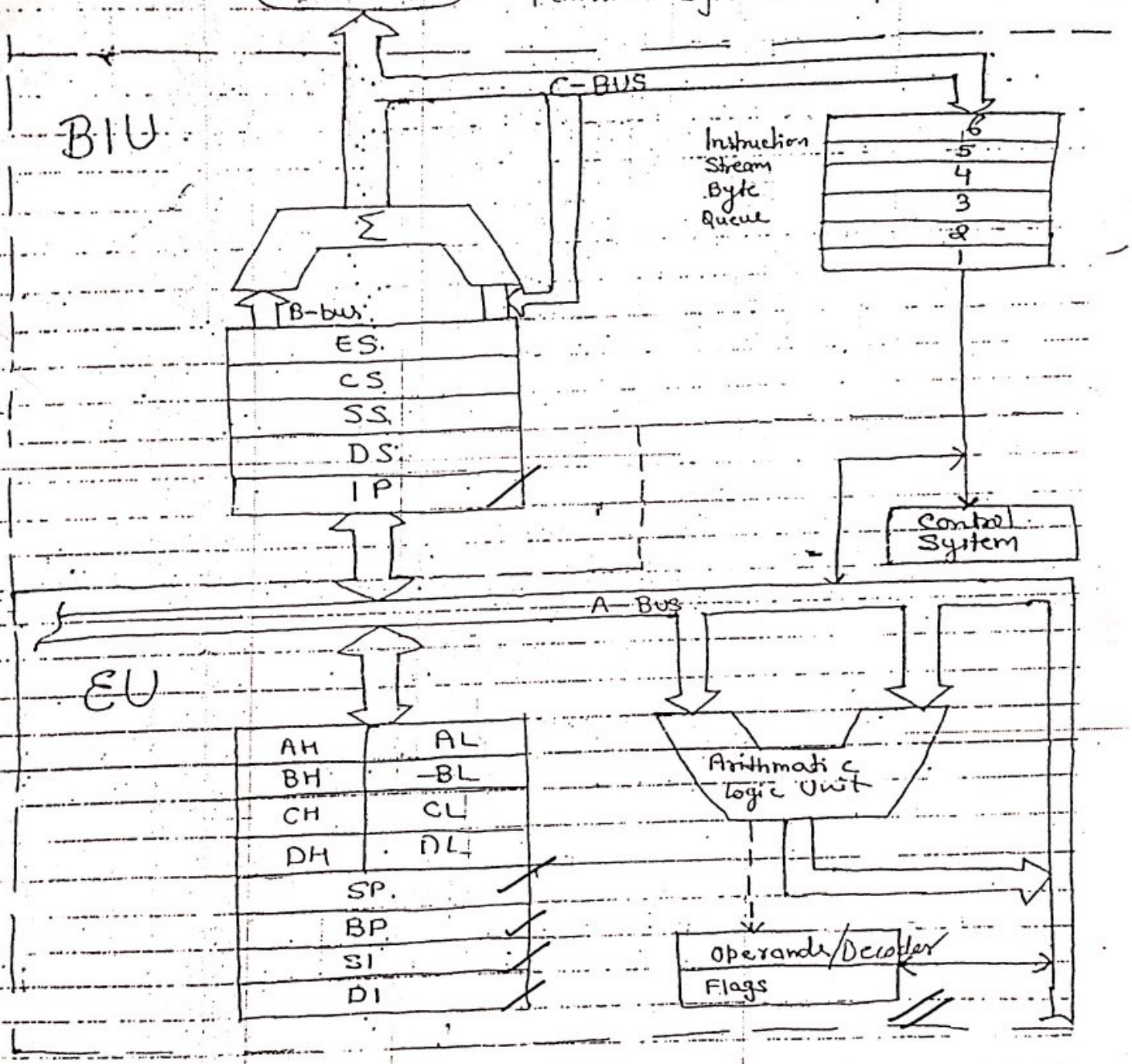
* M/M is divided in two bank ① Even ② odd bank.
* Supports multiprogramming
* 8086 is designed to operate in two modes ① max. mode ② min. mode (4)

8086

* The intel 8086 is a 16-bit µp that is intended to be used as the CPU in a µc.
* The term 16-bit means that its ALU, its internal reg. & most of the instruction are designed to work with 16-bit binary words.
* 8086 has a 16-bit data bus, so it can read data from or write data to m/m & ports either 16 bits or 8-bit at a time.
* 8086 has a 20-bit address bus, so it can address any one of 2^{20} or 1,048,576 m/m location. Each location represents a byte-wide location.
* 16-bit words will be stored in two consecutive m/m locations.
* If the 1st byte of 16-bit word is at an even address the 8086 can read the entire word in one operation.
* If the 1st byte of the word is at an odd address, the 8086 will read the 1st byte with one operation & 2nd byte with another bus operation.
* At any given time the 8086 works with only 64 KB byte segment within this 1 M-byte range.
* Powerful interrupt structure, rich instruction set, Six byte instruction queue, segment m/m addressing capability, clock freq.

8086 Internal Architecture

The internal logic design of the µp called its architecture, determine how & when various operations are performed by the µp.



8086 CPU is divided into two independent functional parts:

① Bus Interface Unit (BIU) ② Execution Unit (EU)

① BIU sends out addresses, fetches instruction from m/m, reads data from ports & m/m, & writes data to port & m/m. BIU handles all transfer of data & addresses on the buses for the EU. BIU ~~transfers~~ Unit is responsible for establishing communication with external devices & peripherals including m/m via the bus.

BIU Consists of

(a) **Queue** = While the EU is decodg an instrch or executg an instrch which doesn't require use of the buses, the BIU fetches upto six instrch byte for the following instrch. BIU stores these prefetched bytes in a FIFO register set called queue. When the EU is ready for the next instrch, it simply reads the instrch byte for the instrch from the queue in the BIU.

* Fetchg the next instrch while the current instrch executes is called pipelining.

(b) **Segment Register** are used to hold the upper 16 bit of the startg addresses of four m/m segment that the 8086 is working at a particular time.

(b.1) Code Segment (CS) register holds the upper 16 bit of the startg addres for the segment from which the BIU is currently fetchg instrch code byte.

(b.2) Stack Segment (SS) register is used to hold the upper 16 bit of the startg address for the program Stack.

(b.3 & b.4) Extra Segment (ES) & data Segment (DS) registers are used to the upper 16 bits of the startg addresses of two m/m segments that are used for the data.

(c) Instrction Pointer (IP) register contains a 16-bit offset which tells, where in that 64 KByte Code Segment the next instrch byte is to be fetched from.

(d) Adder = the BIU produces the required 20-bit physical address by adding an offset to a segment base value represented by the contents of one of the segment register.

(2) **EU** tells the BIU where to fetch instrchs or data from, decodes instrchs & executes instrchs. EU contains

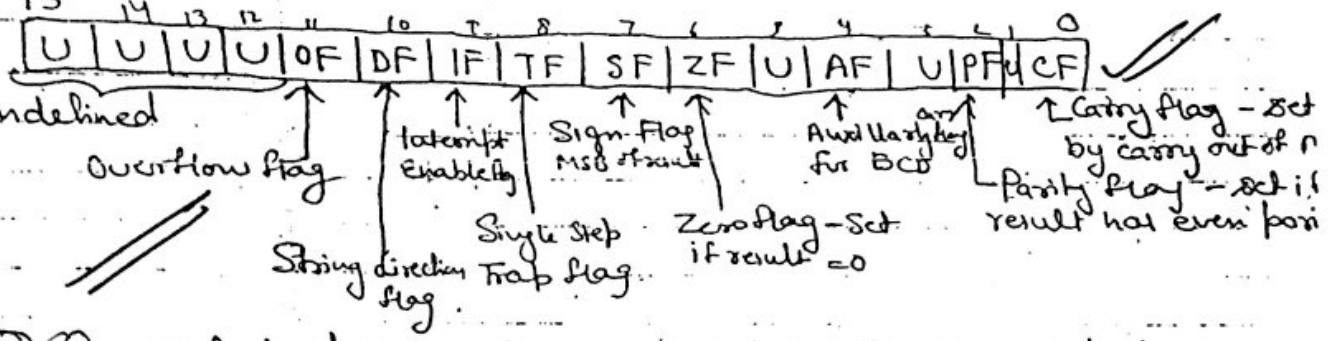
(a) Control System which directs internal operation.

(b) ALU It is a 16-bit ALU which can add, subtract, AND, OR, XOR, increment, decrement, Complement or shift binary no.

(c) decoder in the EU translates the instrchs fetched.

from m/m into a set of actions which the EU carries out.

(d) Flag register = A 16-bit flag register in the EU contains 9 active flags. Flag is a flip-flop that indicate some conditions produced by the execution of an instruction or controls certain operation of the EU.



(e) General-purpose register - EU has 8 general purpose registers AH, AL, BH, BL, CH, CL, DH, DL.
AX BX CX DX
AL reg. is also called the accumulator.

* Advantage of using the internal reg. for the temporary storage of data is that, since the data is already in the EU, it can be accessed much more quickly than it could be accessed in the external memory.

(f) Stack pointer (SP) reg. - is the EU holds the 16-bit offset from the start of the segment to the m/m location where a word was most recently stored on the stack.

* The m/m location where a word was most recently stored is called the top of stack.

(g) Base Pointer (BP) reg., Source Index (SI) reg., Destination Index (DI) reg. → These three 16-bit reg. can be used for temporary storage of data just as GPR. Their main use is to hold the 16-bit offset of a data word in one of the segment.

DS : SI
ES : DI

Special alternate function of GPR

AX is used as 16-bit accumulator, AL can be used as 8-bit accumulator
BX is used as an offset storage for forming physical address
CX is also used as a default counter in case of string & loop instructions
DX is general purpose reg. which may be used as an implicit operand or destination in case of a few instructions.

Flag Register

The description of each flag bit is as follows:

Sign Flag (SF) – This flag is set when the result of any computation is –ve. For signed computations, the sign flag equals the MSB of the result.

Zero Flag (ZF) – This flag is set if the result of the computation or comparison performed by the previous instruction is zero.

Parity Flag (PF) – This flag is set to 1 if the lower byte of the result contains even no. of 1s.

Carry Flag (CF) – This flag is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

Trap Flag (TF) – If this flag is set, processor enters the single step execution mode. A trap interrupt is generated after execution of each instruction. The processor executes the current instruction & the control is transferred to the Trap interrupt service routine.

Interrupt Flag (IF) – If this flag is set, the maskable interrupts are recognised by the CPU, otherwise they are ignored.

Direction flag (DF) – This is used by string manipulation instruction. If this flag bit is '0', the string is processed in autoincrementing mode otherwise the string is processed from highest address towards the lowest address i.e. autodecrement mode.

Auxiliary Carry Flag (AC-F) – This is set if there is a carry from the lowest nibble, i.e. bit three during addition or borrow.

Overflow Flag (OF) – This flag is set if an overflow occurs, i.e. if the result of a signed operation is large enough to be accommodated in a destination register. If the results overflows into the sign bit.

U	U	U	U	OF	DF	IF	TF	SF	ZF	V	AF	V	PF	U	CF

To communicate with a peripheral (or m/m location) the up needs to perform 3 function with the help of buses

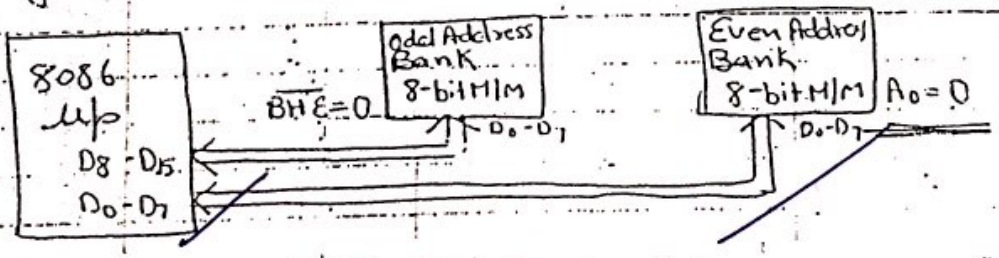
Address Bus — It is a group of 20 lines generally A_0-A_{19}. The address bus is unidirectional → bits flow in one direction → from the up to peripheral devices. The up uses the address bus to perform the 1st function → identifying a peripheral or a m/m location.

Data Bus — It is a group of 16 lines used for data flow. These lines are bidirectional → data flow in both direction b/w up & m/m & peripheral devices. The up uses the data bus to perform 2nd function → transfer binary information.

Control Bus is comprised of single line that carry synchronization signals. The up uses such lines to perform 3rd function → provide timing signal. The up generate specific control signals for every operation (M/M read or I/O write) it performs. These signals are used to identify a device type with which the up intends to communicate.

Physical M/M Organization

In an 8086 based system, the 1MByte m/m is physically organised as an odd bank & an even bank, each of 512 Kbyte, addressed in parallel by the processor.



Byte data with an even address is transferred on D_0-D_7, while the byte data with an odd address is " " D_8-D_{15} bus line. The processor provides 2 enable signals, $\overline{BHE}$ & A0 for selection of either even or odd or both the banks.

Processor: fetches a word from m/m which may be data operand, may opcode bits or one of the byte may be opcode. while the other may be data.
OPcode & Operands are identified by the internal decoder which further derives the signals these acts as I/P to the timing & Control Unit. Then timing & control unit derives all the signal required for execution of the instruction.

A map of an 8086 m/m system starts at 00000H & ends at FFFFFH. 8086 being a 16-bit processor is expected to access 16-bit data to/from 8-bit commercially available m/m chips in parallel.

To maintain an upward compatibility with 8085, 8086 must be able to implement 8-bit operation.

imp Certain location in m/m are reserved for specific CPU operation. The location from FFFF0H to FFFFFH are reserved for operations including jump to initialisation programme & I/O-processor initialisation. The location 00000H to 003FFH are reserved for interrupt vector table. The interrupt structure provides space for a total of 256 interrupt vectors. The vectors i.e. CS & IP for each interrupt routine require 4 byte for storing it in the interrupt vector table. Hence, 256 type of interrupt require 256 × 4 = 03FFH = 1KByte location for the complete interrupt vector table.

00000H

FFFFH — 0 — 003FFH

Segmentation of M/M

8086 access m/m using the segment : offset approach rather than accessing m/m directly with 20-bit address.

* Segment : offset scheme req. only a 16-bit no. to represent the base address for a segment & only a 16-bit offset to access any location in a segment.

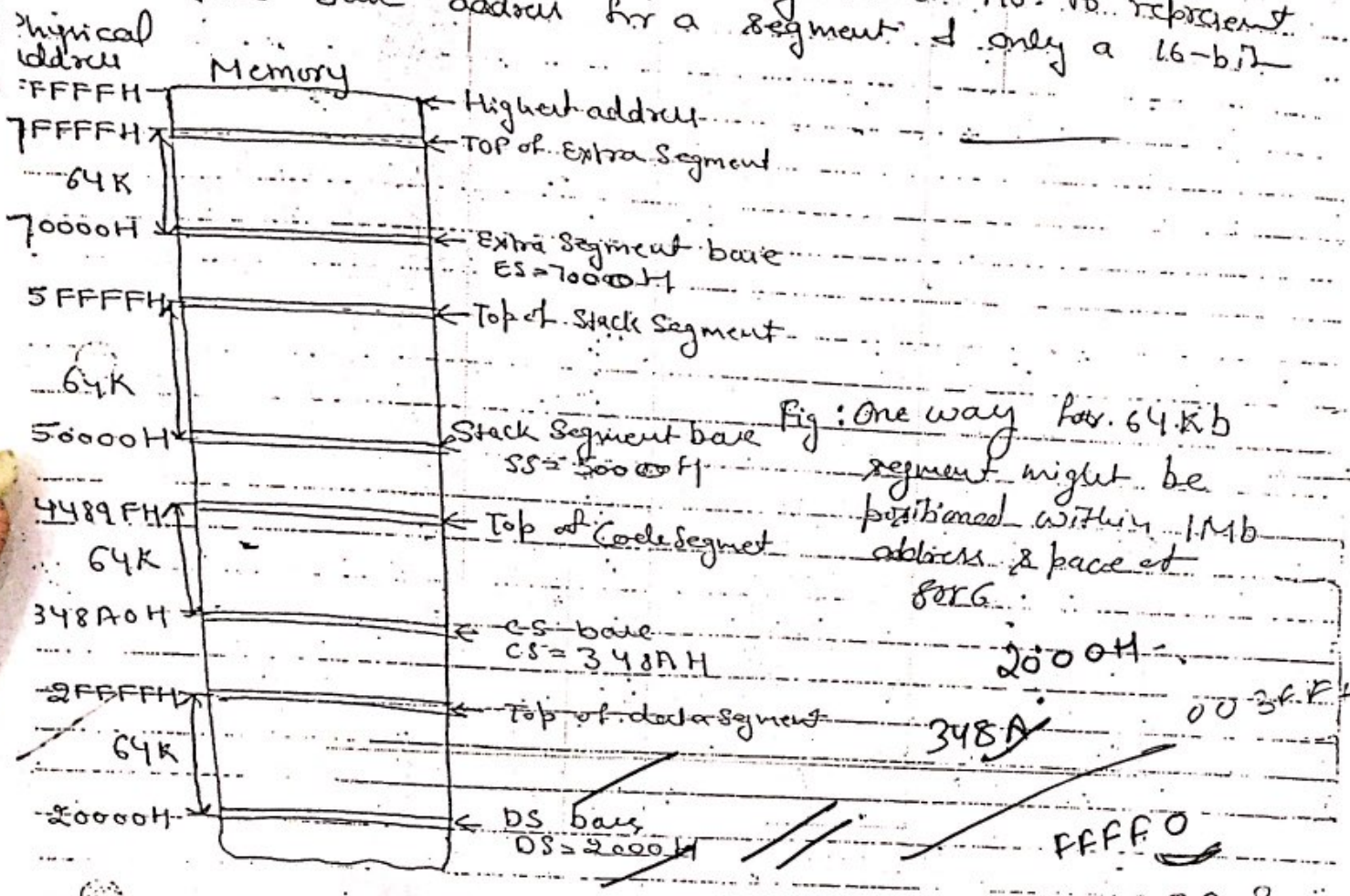


Fig : One way for 64 Kb segment might be positioned within 1 Mb address space of 8086.

To manipulate & store only 16-bit quantities instead of 20-bit quantities with the 8 & 16 bit register in 8086.

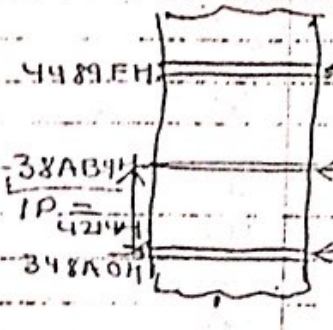
1. In a timesharing computer system segmentation makes it easy to keep users program & data separate from one another, & also make easy to switch from one user's program to another user's program.
2. Allows the m/m capacity to be 1 MByte although the actual addresses to be handled are of 16-bit size.
3. Allows the placing of code, data & stack portion of the same program in different segments of m/m for data & code protection.
4. Permit a program and/or its data to be put into different areas of m/m each time the program is executed i.e. portion for relocation is done.

Method of Generating Physical Address

The four segment can be separated or for small program which do not need all 64kbyte in each segment they can overlap.

→ Segment reg. are used to hold the upper 16 bits of the starting address for each of the segment.

E.g. ⇒ CS reg. holds the upper 16-bit of the start address for the segment from which BIU is currently fetching instruct code byte.



⇒ BIU always inserts zero for the lowest 4 bits of the 20-bit starting address for a segment.

If CS contain 348AH the CS reg will start at address 348A0H

* A 64KB segment can be located anywhere within 1MB address space but the segment will start at an address with

CS =	3	4	8	A	0 ← H/W zero
IP +		4	3	2	H
Physical address	3	8	A	B	4

zeros in the lowest 4 bits.

The IP reg. holds the 16-bit address or offset of the next code byte within this code segment. The value in the IP must be offset from (added to) the segment base address in CS to produce the req. 20 bit physical address sent out by the BIU.

Similarly (SS x 10) + SP = Physical address Default ⇒ SS : SP or SS : BP

(ES x 10) + IP = " " CS : IP

DS : BX or DI or SI or 8 or 16-bit no. ES : DI

ES : DI for string Instruct

Type of M/M reference	Default Segment	Alternate Segment	Offset/Logical
Instruct fetch	CS	none	IP
Stack op	SS	"	SP
General data	DS	CS, ES, SS	effective ad
String Source	DS	"	SI
String destination	ES	none	DI
BX used as pointer	DS	CS, ES, SS	effective ad
BP used as pointer	SS	CS, ES, DS	"

Real Mode M/M addressing

* The 1st 1MByte of M/M is called the real m/m, conventional m/m or DOS m/m system. DOS Operating system requires that the up operates in the real mode.
* Windows doesn't use the real mode.
* Real mode operation allow application S/W written for 8086/8088, which only contain 1MB of m/m, to function in 80286 & above without changing the S/W.
* A combination of a segment address & an offset address accesses a m/m location in the real mode.
* Segment address, located within one of the segment register, defines the beginning address of any 64KB m/m segment.
* Offset address selects any location within the 64KB m/m segment. It is also known as displacement.
* Because of the internally appended 0H, real mode segment can begin only at a 16-byte boundary in the m/m system.
* Once the beginning address is known, the ending address is found by adding FFFFH.

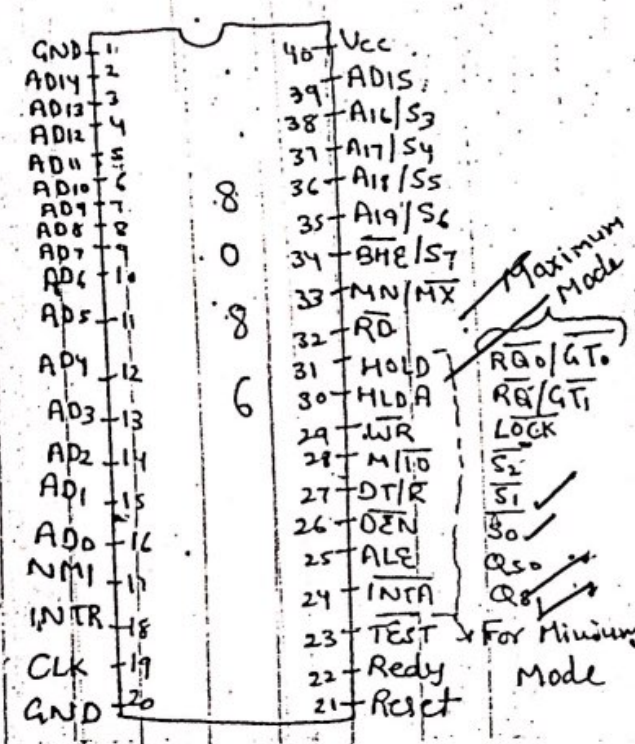
e.g.

Segment reg.	Starting address	Ending address
2000H	20000H	2FFFFH
2100H	21000H	30FFFH
1234H	12340H	2233FH

* Default Segment & offset register → (special purpose)
  - CS : IP — Instruction address
  - SS : SP or BP — Stack address
  - DS : BX, DI, SI, 8/16-bit no. — data address
  - ES : DI for string instr. — String destination address
* DOS program loader calculates & assign segment starting address.
* Segment & offset address scheme allow relocation. A relocatable program is one that can be placed into any area of m/m & executed without change. Relocatable data are data that can be placed in any area of m/m & used without any change to the program.

Pin Configuration of 8086

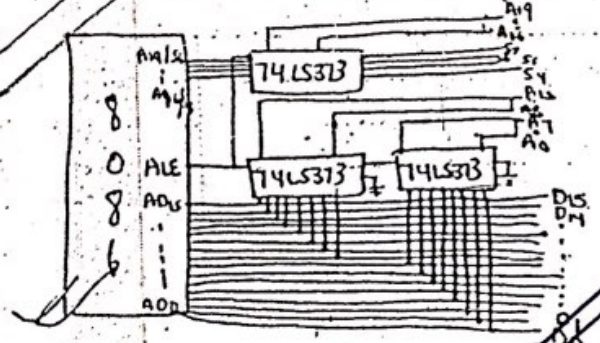
Intel 8086 consists of a total of 40 pins & some of are multiplexed



V_{cc} = +5V pin 40

V_{SS} = GND Pin 1, 20

$AD_{15} - AD_0$ (Address/data) — The data bus & least significant 16 bit of address bus are shared on the same pins. This is known as multiplexing.



$AD_{15} - AD_0$ are used for data as well as the least significant 16 bit of address.

* During T_1 state of a machine cycle the up sends out address on $AD_{15} - AD_0$.

* In later part of machine cycle the up sends out data or receive data on these pins.

NMI — Non maskable Interrupt is a positive edge triggered interrupt. It is an I/P Pin & is active high signal. NMI is a vectored interrupt & has higher priority than INTR. It can be used in emergency like power supply failure.

INTR is a maskable interrupt request which is level triggered. Whenever an external device activate

pin, the µp will be interrupted only if interrupt are enabled using a STI (set Interrupt flag) instruction.

CLK – The clock used by 8086 is 5 MHz. This clock signal is generated externally by using generator IC 8284. It is applied on the clk pin.

RESET – When a logic 1 is sent into the 8086 on the reset pin, for atleast 4 clock cycles, the µp get reset. When reset get low, 8086 initialize the system as
CS – FFFF H
IP, DS, SS, ES – 0000H.
queue = empty, Flags – clear

READY – When ready I/P = 1, then 8086 perform normal operation If ready I/P = 0, then 8086 enters in wait state b/w T_3 & T_4 clock cycle.

TEST – If 8086 get WAIT instruction then 8086 will check TEST pin. This pin is used when 8086 is operating with co-processor 8087 or any other CPU. If TEST = 1, then 8086 enters in WAIT state & it will remain in this state till TEST become zero.

RD – It is an active low output pin. Whenever the µp needs to get information from a mem location or an Yo port, it activates the RD pin.

MN/MX – In Simple system, 8086 is the only processor in the system, all the control signal can be directly generated by 8086. The 8086 enters into minimum mode if this pin is at (+5V).
1 → MIN
0 → MAX
When this pin is connected to ground, CPU enters into maximum mode. This is a Complex system & multiple processor, arithmetic co-processor are required to generate control signal for proper interaction among them.

* Pin 24 to 31 will have different function in different mode.

BHE/S7 – During the 1st clock cycle, T_1 of a machine cycle, the µp used this pin to send out Bus High Enable. In subsequent clock cycle µp sends 0 on S7 as Status information. It is used by 8087 Co-processor to determine whether CPU is 8088 or 8086.

$A_{19}/S_6 - A_{16}/S_3$ - Pins A_{19}-A_{16} provide the most significant 4 bits of m/m address during 1st clock cycle T_1 of a machine cycle. In subsequent clock cycle these pins send out S_6-S_3 status information. S_5 indicate the value of interrupt enable flag bit. S_6 is always 0 if not used.

S_4	S_3	Segment Register
0	0	ES
0	1	SS
1	0	CS (None in case of I/O port access)
1	1	DS.

Minimum Mode - Pin 24 to 31

INTA - Interrupt acknowledge → Whenever INTR I/P pin is activated by an I/O port, if interrupt are enabled & NMI is not active at the same time, the µp finishes the instruction it is executing & gives a logic 0 on INTA pin.

ALE - Address Latch Enable is an O/P pin. 8086 sends out a logic 1 on this pin during T_1 clock cycle of a machine cycle. An external latch 74LS373 use this signal to latch on to the information sent out by the 8086 during T_1. This latched information is available A_{16}-A_{19} keeps to m/m or I/O ports for the whole machine cycle. $\overline{BHE}/S_7$ is also latched using ALE & 74LS373.

DEN - Data Enable → If $\overline{DEN}=0$, it enable bidirectional tristate buffer during DMA operation.

DT/$\overline{R}$ - Data Transmit or Receive → 8086 O/P are not capable of supplying enough current drive. The bidirectional buffers, also called Transceiver, like Intel 8286 are used in the data bus to increase the current driving capability. The DT/$\overline{R}$ is an O/P pin of 8086, which is activated in T_1 of a machine cycle (If DT/$\overline{R}=1$, the buffer are used to transmit the data from 8086, if 8086 O/P a logic 0 on this pin, the buffer are set up so that 8086 receive data from the buffers.

M/$\overline{IO}$ - Memory / Input or output is an O/P pin. When µp communicates with m/m, the µp O/P a logic 1

on this pin & if up communicate with I/O ports, it o/p a logic 0 on this pin..

HOLD – Hold Request & HLDA – Hold acknowledge
In 8086, there are no instruction to transfer data directly b/w m/m & I/O Port. M/M & I/O port have to communicate via the processor which is slow.
The proces of an I/O port accessing m/m directly, without having data through the up is called Direct M/M Access (DMA). A DMA controller like Intel 8237/8257 can be used for performing DMA data transfer in a computer system.
→ The DMA controller activates the HOLD I/P of 8086, whenever DMA data transfer is requested by an I/O port. The up check this I/P pin at the end of every machine cycle in an instruction cycle. When 8086 finds that the HOLD pin is active, it tristate the address bus, data bus & control signal & enters the hold state.
→ The up indicates to the DMA controller that it has entered the hold state by sending out HLDA signal. Then DMA controller takes control over the system buses & generate necessary control signal & m/m address so that the peripheral can perform the DMA data transfer.

Q50|:

Maximum Mode Pins

QS0, QS1 – Queue Status – In max mode, QS1 & QS0 sent out by 8086 to provide the queue status: When 8086 is in control of the local bus the 8288 bus controller generates bus control signal using these signal sent out by 8086:

QS1	QS0	Queue Status
0	0	No instruction is taken from queue, so queue is full.
0	1	1st byte (opcode) from queue.
1	0	Due to branching instruction, queue are flushed.
1	1	Next byte (opcode) from queue.

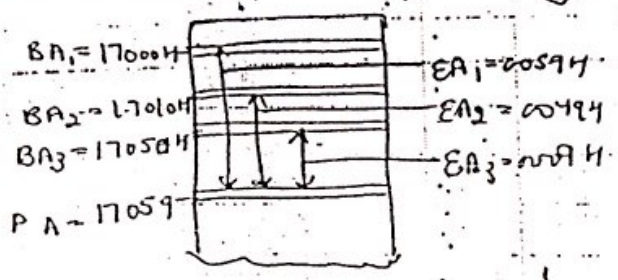
The Status information sent out by S2, S1, S0 provide the bus status

$\overline{S_2}$	S_1	S_0	Bus Cycle
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	Fetch
1	0	1	Read M/M
1	1	0	Write M/M
1	1	1	Not defined

LOCK — LOCK prefix for an instruction allows a up to make sure that another up does not take control of the system bus during execution of a critical instruction which uses the system bus. This is achieved by 8086 by activating lock signal. This signal is connected to external bus controller which prevent any other up from taking control of the system bus.

$\overline{RQ}/\overline{GT_0}$, $\overline{RQ}/\overline{GT_1}$ — Request / Grant are I/P as well as O/P. They are active low pins. If they are left without connection or logic 1 they are in inactive state. These pins are used for gaining control over the local bus in max. mode. $\overline{RQ_0}/\overline{GT_0}$ has higher priority than $\overline{RQ_1}/\overline{GT_1}$.

Relocatibility



The physical address of any m/m location will be fixed but if the base address is changed, then its effective address will change.

The same m/m location can be represented or accessed or selected by more than one effective address. This characteristic is called relocatibility.

V. Imp Addressing Modes of 8086

The different ways in which a processor can access data are referred to as its addressing mode.

* divided into 2 parts
  a) Addressing mode for data related instruction
  b) " " " branching instruct

a.1 **Immediate addressing mode** – If 8/16 bit data required for executing the instruction is given alongwith the instruction

e.g. MOV AL, 75H
MOV BX, 7506H

a.2 **Direct addressing mode** – If 8/16 bit data is present in m/m & 16 bit effective address of this m/m location is given alongwith the instruction.
It takes base address from default segment reg. DS.

e.g. MOV AL, [9106H]

DS = 1000H
Base address = 10000
EA = 9106
PA = 19106

19106 [03H] → [03H] AL

a.3 **Reg. direct addressing mode** – If 8/16 bit data required for executing the instruction, is present in register & the name of register is given alongwith the instruction.

e.g. MOV CX, BX

a.4 **Reg. Indirect addressing mode / Indirect Addressing mode** – If 8/16 bit data req. for executing the instruction is present in m/m & the 16-bit EA of this

higher address.

m/m location is present in a register & the name of this register is given alongwith the instruction.

* In this addressing mode EA can be stored in BX/SI/DI.

eg:- MOV AL, [SI]

Let DS = 1903 0
SI + 8165
PA 2 1195

21195 034H → AL

Use → To refer tabular data located in m/m.

Q.5 Register relative addressing mode — If data required for executing the instruction is present in m/m location & the EA of this m/m location is obtained by the equation

EA = BX/BP/SI/DI + 8/16 bit Displacement

e.g. MOV CX, 97H[BP] ⇒ MOV CX, [BP+97H]

Let SS = 1906 BP = 2310

19060
2310
1B407

1B407 031H → 03 CL
1B408 04H → 04 CH

Use → To address elements in a m/m array.

BX = begining address of array
DI = element to be accessed

Q.6 Base Index Addressing mode — If data required for executing the instruction is present in m/m location, then in base index addressing mode, up will calculate the EA by equation

EA = [BX]/[BP] + [SI]/[DI]

e.g. MOV CX, [BX][SI] ⇒ MOV CX, [BX+SI]

Let BX = 1573
SI = A1C2
EA = B735

BA = 17230
EA = + B735
PA = 22965H

22965H 01 → CL
22966H 02 → CH

Use → To address a 2-dimensional array of m/m data.

Q.7 Relative base Index addressing mode — Data required for executing the instruction is present in m/m & in the instruction the name of any one base reg, index reg & 8/16 bit displacement is given.

e.g. MOV AH, 1907H[BX][DI]
MOV DX, [BP+SI+9FH]

* If the displacement is of 8-bit size, then it is converted into 16-bit by extending the sign bit to upper byte. ero-

eg. ① MOV DX, [BP+SI+9FH] Sign bit of 9F is 1 (1001 1111)

Let BP = 1823, SI = 2910, SS = 8100

Sign extension of 9F is FF9F H

EA = 1823 (BP) + 2910 (SI) + FF9F (displacement) = 40D2 (Discard the carry 1)

PA = 81000 + 40D2 = 85 0D2

eg. ② MOV DX, [BP+SI+1FH] (0001 1111, sign bit 0)

Sign extension of 1F = 001F H

EA = 1823 + 2910 + 001F = 4152 H

a.8 Implicit addressing mode — If address of source of data as well as address of destination of result are fixed, then no operand is given alongwith the instruction.

e.g. CLD, STD — Clear direction flag / Set direction flag

a.b.1 Direct port mode — The port no. is an 8-bit immediate operand. This allows fixed access to port numbered 0 to 255

e.g. OUT 05H, AL

a.b.2 Indirect port mode — The port no. is taken from DX allowing 64K 8 bit port or 32K 16-bit port

e.g. IN AL, DX

DX = 7890 [01] → [01] AL

Branching Instructions

* If the location to which the control is to be transferred lies in a different segment other than the current one the mode is called Inter segment mode. (Direct / Indirect)

* If the destination location lies in the same segment the mode is called Intrasegment mode. (Direct / Indirect)

b.1 Intra-segment direct mode — In this mode, the address to which the control is to be transferred lies in the same segment in which

the control transfer instruction list of appear directly in the instruction as an immediate displacement value. The EA to which the control will be transferred is given by the sum of 8 or 16 bit displacement & current content of IP.

JMP SHORT Label ; Label lies b/w +127 & -128 of CS
JMP NEAR PTR Label ; Label lies b/w +32767 & -32768 of CS

Intra-segment indirect mode — The branch address is found at the content of a reg. or m/m location. * This addressing mode may be used in unconditional branch instruction.

e.g. Call [CX] (λ → IP indirect)
JMP FAR PTR [BX] JMP [BX]

Inter-segment direct mode — It provide a means of branching from one CS to another CS. Here the CS & IP of the destination address are specified directly in the instruction.

e.g — JMP 5000 : 2000H

Inter-segment indirect mode — The address is passed to the instruction indirectly i.e. the content of a m/m block containing 4 bytes ie IP (LSB), IP (MSB), CS (LSB) & CS (MSB) sequentially. The starting address of the m/m block may be referred using any of the addressing mode except immediate mode. The address can be generated by using one index register or by using two index registers.

eg - JMP [Zero]

Assembly Language Programming

* Assembly languages use 2, 3 or 4 letter mnemonics to represent each instruction type.
* The letters in an assembly language mnemonics are usually initials or a shortened form of the english word for the operation performed by the instruction.
e.g :- for Subtract ; ∴ SUB
* Assembly language statements are usually written in a standard form that has four fields

Label	opcode	operand	Comment
Next :	ADD	AL, 07H	; add correction factor

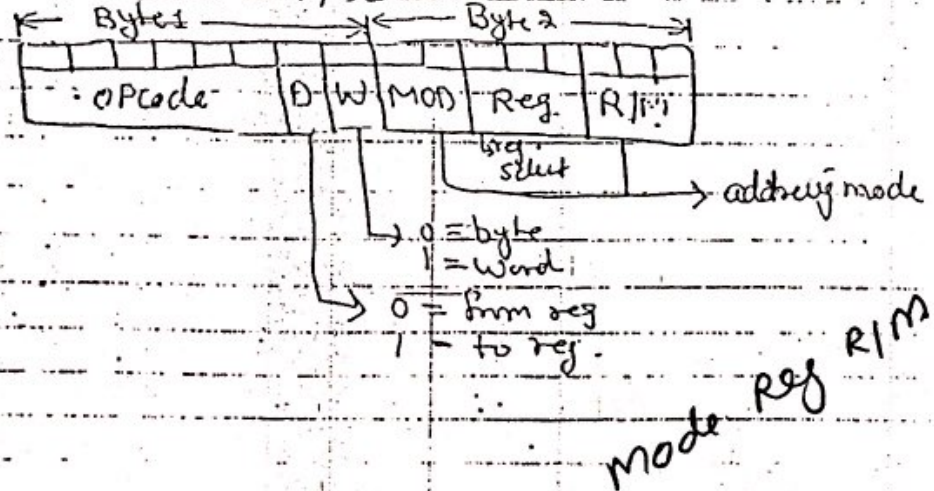
(h) Processor Control Instructions

- STC – Set CF = 1
- CLC – Clear CF = 0
- CMC – Complement the state of the CF
- STD – Set DF to 1 (decrement string pointer)
- CLD – Clear DF to 0.
- STI – Set IF to 1 (enable INTR interrupt)
- CLI – Clear IF = 0 (disable INTR ")
- HLT – Halt (do nothing) until interrupt or reset
- WAIT – Wait (do nothing) until signal on the TEST pin is low.
- ESC – Escape to external coprocessor such as 8087/8089
- LOCK – An instruction prefix. Prevents another processor from taking the bus while the adjacent instruction executes.
- NOP – No operation except fetch & decode.

Construct machine code for 8086 instructions

8086 instruction sizes vary from one to six bytes.

* One byte instruction – It takes only one byte of m/c & may have implicit data or register operands.
  e.g. – LAHF, SAHF, CBW, CWD.
* Two byte instruction – It require two bytes of m/c. 1st byte of code specifies operation code & width of operand. 2nd byte shows the reg. operands & R/M field.
  e.g. : Reg. to Reg., Reg. to M/M without displacement
  MOV CH, BL
  MOV [SI], DL



mode Reg R/M

* 3 byte Instruction requires 3 location of m/m. The additional byte for 8 bit displacement alongwith 2 byte for opcode, reg & mod.

eg = reg to m/m with displacement (8 bit)
M/M to reg " "
8 bit immediate data to register.

MOV 43H[SI], 0H

* 4 byte Instruction – requires 4 bytes of m/m location. Two additional bytes are used for storing 16 bit displacement. eg → 16 bit immediate data to reg.

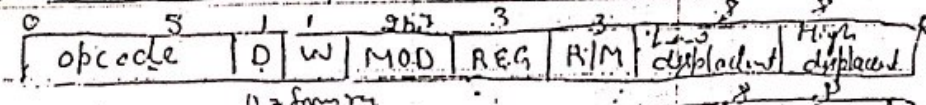
eg: MOV 1234[BP], DX

* 5 byte & 6 byte Instruction – requires 5 or 6 bytes for m/m. 1st two byte contains the information regarding OPCODE, MOD, & R/M fields. The remaining byte contains a byte of displacement & one byte of data per 2 byte of data.

eg: MOV BP[SI+500], 9298H

eg - Immediate 8/16 bit operand to m/m with 8/16 bit displacement.

Instruction Template –



MOD (2 bits)	Interpretation
00	M/M mode with no displacement follows except for 16-bit displacement when R/M = 110
01	M/M mode with 8 bit displacement follows
10	" " 16 bit "
11	Reg. mode (no displacement)

Mod 11

Reg. field occupies 3 bits. It defines the reg. for the 1st operand which is specified as source or destination by D bit.

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

Mod = 11

R/M occupies 3 bits. The R/M field alongwith the Mod field defines the second operand.

Mod 11 Reg. to Reg. Mode

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

Effective address calculation –

R/M	Mod = 00	Mod = 01	Mod = 10	Mod W=
000	[BX] + [SI]	[BX] + [SI] + D8	[BX] + [SI] + D16	AL
001	[BX] + [DI]	[BX] + [DI] + D8	[BX] + [DI] + D16	CL
010	[BP] + [SI]	[BP] + [SI] + D8	[BP] + [SI] + D16	DL
011	[BP] + [DI]	[BP] + [DI] + D8	[BP] + [DI] + D16	BL
100	[SI]	[SI] + D8	[SI] + D16	AH
101	[DI]	[DI] + D8	[DI] + D16	CH
110	direct address [D16]	[BP] + D8	[BP] + D16	DH
111	[BX]	[BX] + D8	[BX] + D16	BH

eg-1 MOV CH, BL

Case 1 opcode for MOV = 100010₂

W = 0 byte opn

D = 0 BL is a source

MOD = 11

R/M field = 101 (CH)

Reg field = 011 (BL)

[100010 00] [11 011 101]

Byte 1 Byte 2

0 - from reg.

Case 2 MOV SP, BX

[100010 1 1] [11 100 011]

1 - big word SP BX

to from

eg-2 SUB BX, [DI]

R/M = 101 D = 1

Reg = 011 W = 1

Mod = 00

[0010 1011] [00 011 101]

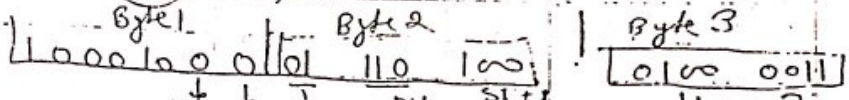
MOV CL, [BX]

[100010 1 0] [00 001 111]

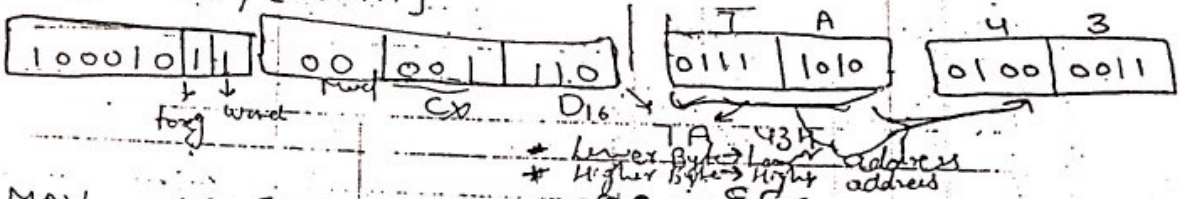
1 - byte 2 - indirect data from

Assembler Directives are the directions to the assembler

eg.3 MOV [43H][SI], DH



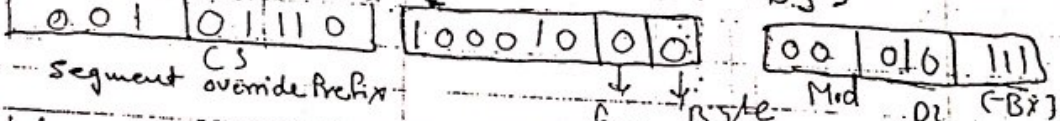
eg.4 MOV CX, [437AH]



eg.5 MOV CS:[BX], DL

Segment override
Segment Register

00 – ES
01 – CS
10 – SS
11 – DS
} code for segment register



One byte e.g –

LAHF = Load AH with flags 1001 1111
SAHF = Store AH into flags 1001 1110

S-bit is called as sign extension bit

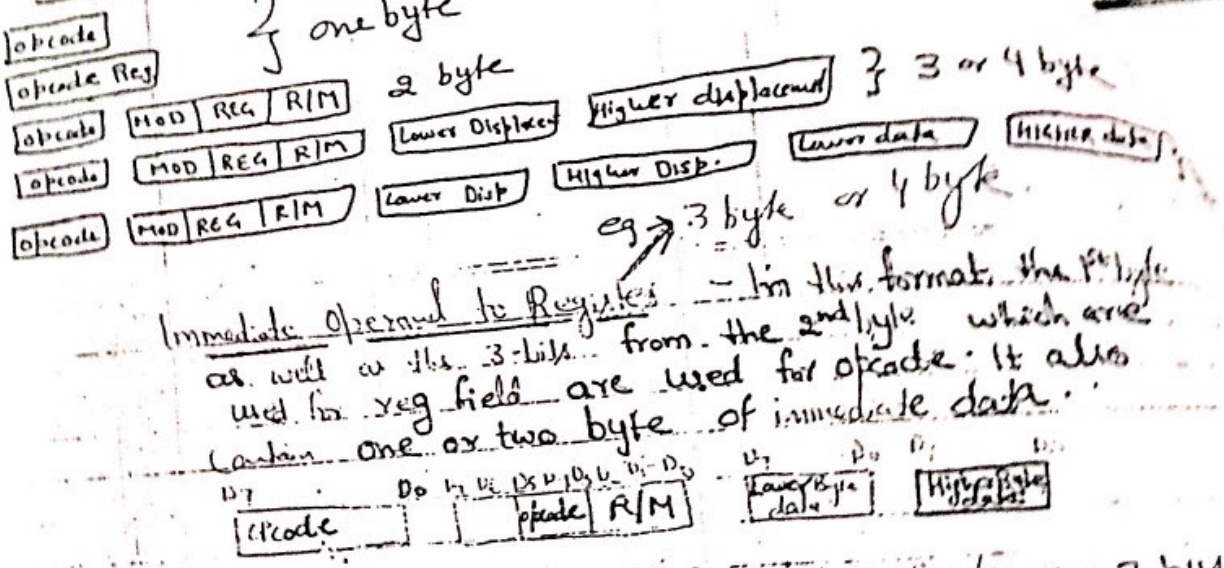
8-bit operation with 8-bit immediate operand is indicated by S=0 & W=0
16 " " " 16 " " " " " " " S=0 & W=1
16 " " " Sign extended immediate data by S=1 & W=1

V-bit is used in case of shift & rotate instruction. This bit is set to '0' if shift count = 1 & set to '1' if CL contains shift count.

V=0 if count = 1
V=1 if count = more than 1

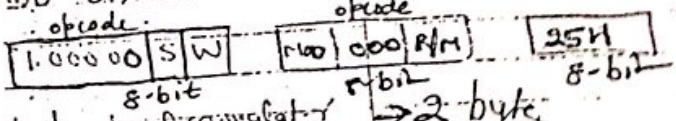
Z-bit is used by REP instruction to control the loop, if Z=1, the instruction with REP prefix is executed until the zero flag matches the Z bit.

Instruction Format



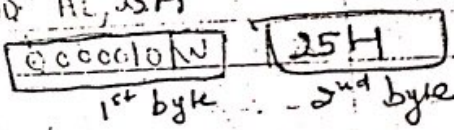
Immediate Operand to Registers — In this format the 1st byte as well as the 3 bits from the 2nd byte which are used for reg field are used for opcode. It also contain one or two byte of immediate data.

Ex:- ADD BL,25H — Immediate to Reg/Mem → 3 byte



Immediate to Accumulator → 2 byte

ADD AL,25H



SHR = 110 00 V W | MOD 101 R/M

ROL = 110 00 V W | MOD 000 R/M

REP = 1111 001 Z

CALL

① direct within Segment — 11101000 | disp Low | disp-high

② Indirect within Segment — 11111111 | mod 010 R/M

③ Indirect Intersegment — 11111111 | MOD 011 R/M

④ Direct Intersegment — 10011010 | offset Low | offset High | Seg Low | Seg High

Segment Override is required to accen data in CS. Program using this technique must pay a penalty of one extra byte of code per data accen instruction. Segment override is in-effect for the one instruction only & must be specified for each instruction to be overriden.

MOV CS: [BX], DL — This instruction copies a byte from DL reg to a m/m location: The effective address for the m/m location is contained in BX reg - Normally an effective address in BX will be added to the DS to produce physical m/m address. In this instruction CS: [BX] indicate that we want the BIU to add the effective address to the CS to produce physical address. The CS: is called a segment override prefix.

When an instruction containing a segment override prefix is coded an 8-bit code for the segment override is put in m/m before the code for the rest of the instruction.

Byte 1 | 0 | 0 | 1 | S | R | 1 | 1 | 0 |

if SR = 00 it means ES override
= 01 " " CS "
= 10 " " SS "
= 11 " " DS "

eg - MOV ES: [BP], CX

MOV CS: [BX], DL

| 0 | 0 | 1 | S | R | 1 | 1 | 0 |

MOV CS: [BX], DL

MOV CS: [BX], DL

| 0 | 0 | 1 | | | 1 | 1 | 0 |

Instruction Set of 8086

Data Copy / Transfer Instruction → Transfer data from source to destn.

① MOV → Move – This instruction transfer data from one register/m/m to another reg/m/m locations. Source may be any one of the segment reg./general purpose reg./m/m & another reg./m/m act as destination.

* In Immediate Addressing mode, segment reg. cannot be a destination register

MOV DS, 5000 ✗ not permitted

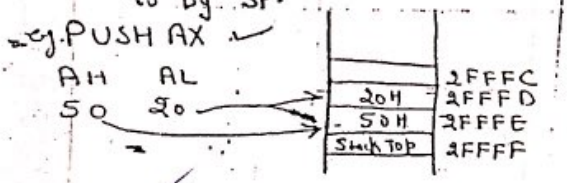
MOV AX, [SI] ✓ MOV DS, AX ✓ MOV AX, [2000]

MOV AX, 5000 ✓ MOV AX, BX ✓ MOV AX, 50H [BX]

② PUSH → Push to Stack – Pushes the contents of the specified reg./m/m location on to the stack. SP is decrement by 2, after each execution of the instruction.

* ① Current stack top is already occupied so decrement SP by one then store upper byte into the address pointed to by SP
② Decrement SP by one & store lower byte into location pointed to by SP.

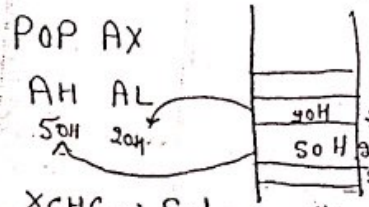
eg. PUSH AX ✓



SS = 2000 eg PUSH [2000]
SP = FFFF PUSH DS

③ POP → Pop from Stack → load the specified reg./m/m with the contents of m/m locations of which the address is formed using the current SS & SP.

* ① Contents of stack top m/m location is stored in lower byte, m/m & SP is incremented by one.
② Further contents of m/m location pointed by SP are copied to upper byte reg./m/m & SP is again incremented by 1. SP is incremented by 2 & points to next stack top



SS = 2000
SP = 2FFD

eg POP DS
POP [5000]

④ XCHG → Exchange the contents of specified source & destination operands, which may be reg. or one of them may be m/m

XCHG [5000H], AX ✓ XCHG [5000H], [SI] ✗

XCHG AX, BX ✓

(4) IN → Input the port – It is used for reading an I/P port. The address of the I/P port may be specified in the instruction directly or indirectly. * AL & AX are allowed destination.
* DX is only reg. (implicit) which is allowed to carry the port address. If the port address is of 16 bit it must be in DX.

```
IN  AL, 05H   ✓
IN  AX, DX    ✓
IN  AX, 1234H ✗
```

(5) OUT → Output to port → It is used for writing to an I/P port.
* reg. AL & AX are allowed source operands
* If port address is of 16 bit, it must be in DX.

```
OUT  03H, AL   ✓
OUT  DX, AX    ✓
OUT  1230H, AX ✗
```

(6) XLAT → Translate – It is used for finding the codes in case of code conversion problems, using look-up table technique.

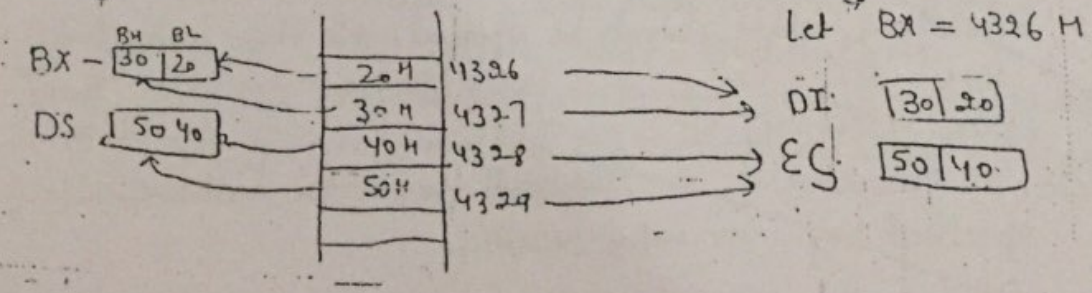
```
MOV AX, seg table   ; address of the segment containing lookup table.
MOV DS, AX
MOV AL, code        ; code of pressed key is transferred in AL.
MOV BX, offset table ; offset of the code look up table is in BX
XLAT                ; Find the equivalent code & store in AL
```

(7) LEA → ~~Load~~ Effective Address – It loads the effective address ~~formed~~ by destination operand into specified reg.

```
LEA BX, ADR       ; offset of label ADR will be transferred to BX
LEA SI, ADR[BX]   ; ..... added to content of BX to form effective address & loaded in SI.
```

(8) LDS → Load reg. & DS with words from m/m
LES → Load reg. & ES with words from m/m

```
LDS BX, [4326H]   ✓
LES DI, [BX]      ✓
```



* LDS is useful for pointing SI & DS at the start of a string before using one of the string instruction.
* LES is useful for pointing DI & ES for string instruction.

(10) LAHF :→ Load AH from lower byte of flag → It is used to observe the status of all the Condition Code flags

(11) SAHF :→ Store AH to lower byte of flag reg — It is used to set or reset the flag depending upon the corresponding bit position in AH.

(12) PUSHF → Push flags to Stack → 1st upper byte & then lower byte is pushed on to it. SP is decremented by 2.

(13) POPF → POP flags from Stack → It loads the flag reg completely from the word contents of m/m location currently addressed by SP & SS : SP is incremented by 2

Arithmetic Instruction - perform arithmetic operation.

(14) ADD → adds an immediate data or content of m/m location
* both source & destination cannot be m/m.
* " " " " may be of same type (size).
The result is in the destination operand.

```
ADD AX, 0100H ✓        ADD 25H[BX], AL ✓
ADD AL, BL    ✓        ADD [5000H], 0100H ✓
ADD DX, [SI]  ✓        ADD AX, CS.  X
```

* The content of Segment reg: Cannot be added.

(15) ADC → add with carry → It add a no. from source to a no. from destination & also adds the status of the carry flag into the result.

```
ADC AX, BX. ✓   all added abnc eg.
```

(16) INC → Increment → Increase the contents of the specified reg. or m/ms by 1.
* Immediate data cannot be operand of this instruction.

```
INC AX       ✓    INC [BX] ✓    INC 5000 X
INC [5000H]  ✓
```

(17) DEC → Decrement → Subtract the 1 from the content of specified reg. or m/m location.

```
DEC AX       ✓
DEC [5000H]  ✓
DEC 2011     X
```

(18) SUB → Subtract the source operand from destination operand & result is left in the destination operand. Source operand can be reg./m/m or immediate data. destination "

* both source & destination can not be m/m operand.

SUB AX, 0100H ✓
SUB CH, AL ✓
SUB 2000, [5000] ✗
SUB AX, [2000H] ✓
SUB [5000H], 2000H ✓

(19) SBB → Subtract with borrow — It subtract the borrow flag & source operand from the destination.

SBB AX, [5000H] ✓ all eg. of SUB

(20) CMP → Compare the source operand which may be reg./m/m or immediate data, with destination operand that may be reg./m/m location.

It subtract the source from destination operand but does not store result anywhere, Flags are affected

Source = Destination ZF = 1
Source > Destination CF = 1
Source < Destination CF = 0, ZF = 0

CMP BX, 1000H ✓
CMP [5000H], 1000H ✓
CMP BX, [SI] ✓
CMP BX, AX ✓

(21) AAA → ASCII adjust after addition — AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. AAA instruction converts the resulting content of AL to unpacked BCD form.

assume AL = 0011 0101 i.e ASCII 5
BL = 0011 1001 ie. ASCII 9
ADD AL, BL → AL = 0110 1110 = 6EH ie incorrect BCD
AAA → AL = 0000 0100 i.e. BCD 4
& CF = 1 answer is 14 decimal.

AAA Converts the result of two valid unpacked BCD digit to a valid unpacked BCD no.

* AH must be cleared before addition
AAA instruction works only on the AL reg.

(22) AAS → ASCII adjust after Subtraction
AAS instruction: Correct the result in AL reg. after subtracting two unpacked ASCII operands. The result is in unpacked decimal format.

—

Eg. 1

ASCII 9 − ASCII 5

AL = 39 H

BL = 35 H

SUB AL, BL

AAS

Result AL = 04

CF = 0

Eg. 2

ASCII 5 − ASCII 9 =

AL = 35 H

BL = 39 H

AL = 11111100 = −4

result in 2's Comp

CF = 1

after AAS

AL = 00000100 B

& CF = 1

(23) AAM → ASCII adjust after multiplication — It converts the product available in AL into unpacked BCD format. It follows a multiplication instruction that multiplies two unpacked BCD operands & store result in AX.

AAM replaces the contents of AH by tens of decimal multi & AL by ones of decimal multiplication.

If AL = 35 H BH = 39 H

MUL BH → AX = 35 × 39 = 002DH

AAM → AX = 0405H

(24) AAD → ASCII adjust before division — It converts two unpacked BCD digits in AH & AL to the equivalent binary no. in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte.

If AX = 0607 H & CH = 09H

AAD → AX = 0043 H = 67 decimal

DIV CH Quotient AL = 07 unpacked BCD

Remainder AH = 04 unpacked BCD

(25) DAA → Decimal adjust Accumulator — It converts the result of the addition of two packed BCD no. to a valid BCD no. The result has to be only in AL.

DAA instruction adds 06 to correct the BCD sum.

Let AL = 73 CL = 29

ADD AL, CL

Sum AL = 9C

DAA

+6 → greater than 9 add 6

greater than 9 ← A 2

+60

CF = 1 | 02 in AL

(26) DAS → Decimal adjust after Subtraction - It converts the result of subtraction of two packed BCD no. to a valid BCD no. The subtraction has to be in AL only.

AL = 75 BH = 46

SUB AL, BH result AL = 2F & AF = 1

DAS → AL = 29

(27) NEG → Negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's C it subtract the content of destination from zero.

NEG AL ✓ NEG 25H[BP] ✗

NEG BX ✓ NEG BYTE PTR 25H[BP] ✓

(28) IMUL → Signed Multiplication Byte or Word

It multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by signed word in AX. Source can be general purpose reg. / m/m / index reg. / base reg, but it cannot be an immediate data.

* In case of 32 bit result, the higher word is stored in DX & lower word is stored in AX.
* AL & AX are implicit operands for 8/16 bit multiplication.
* The unused higher bit of the result are filled by Sign bit.

IMUL BH ✓ IMUL [SI] ✓

IMUL CX

(29) MUL → Multiply an unsigned byte or word by the content of AL or AX reg. Source can be a reg. or m/m location specified by any addressing mode. The most significant word of result in DX & lower word in AX

MUL BH AX ← AL × BH

MUL CX (DX)(AX) ← AX × CX

(Higher word, lower word)

(30) CBW → Convert signed byte to word - It copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word.

Byte to be converted must in AL & result in AX.

(31) CWD → Convert Signed Word to Double Word - It copies the sign bit of AX to all the bits of the DX reg.

* CBW & CWD must be done before a signed byte/word in AL/AX can be divided by another signed byte/word with IDIV instruction

(26) DAS → Decimal adjust after Subtraction – It converts the result of subtraction of two packed BCD no. to a Valid BCD no. The subtraction has to be in AL only.

AL= 75 BH=46

SUB AL,BH result AL = 2F f AF = 1

DAS → AL = 29

(27) NEG → Negate instruction forms its complement of the specified destination in the instruction. For obtaining 2's C it subtract the content of destination from zero.

NEG AL ✓ NEG 25H[BP] ✗

NEG BX ✓ NEG BYTE PTR 25H [BP] ✓

(28) IMUL → Signed Multiplication Byte or Word

It multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by signed word in AX. Source can be general purpose reg./m/m / index reg./ base reg, but it cannot be an immediate data.

* In case of 32 bit result, the higher word is stored in DX & lower word is stored in AX.

* AL & AX are implicit operands for 8/16 bit multiplication.

Imp * The unused higher bit of the result are filled by Sign bit.

IMUL BH ✓ IMUL [SI] ✓

IMUL CX

(29) ~~MUL → Multiply an unsigned byte or word by the Content of AL or AX reg. Source can be a reg. or m/m location specified by any addressing mode. The most significant word of result in DX & lowerword in AX~~

MUL BH AX ← AL×BH

MUL CX (DX)(AX) ← AX × CX

Higher word. Lower word.

(30) CBW → Convert signed byte to Signed-Word – It copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word. Byte to be converted must in AL & result in AX.

(31) CWD → Convert Signed Word to Double Word – It copies the sign bit of AX to all the bits of the DX reg.

* CBW & CWD must be done before a signed byte/word in AL/AX can be divided by another signed byte/word with IDIV instruction

(32) [Di the Res In

* In (u

(33) I

* Th

(34) d

*

(35)

(36)

(37)

(38)

(32) DIV → Unsigned divide → It divide an unsigned word by a byte or to divide an unsigned doubleword (2 by a word.

Dividend must be in AX & divisor can be specified by any of the addressing mode except immediate.

Result will be in AL (Quotient) & AH (remainder).

In case of doubleword dividend (DX, AX)

Result AX (Quotient) DX (Remainder)

* In case result is too big to fit in AL or AX, Divide by 0 interrupt is generated.

(33) IDIV → It divide a signed word by a signed byte or doubleword by a word.

* The sign of remainder will be same sign of dividend. The Quotient is also signed no.

```
MOV AL, DIVIDEND
CBW
IDIV DIVISOR
```

IDIV BL ✓
IDIV CX ✓

Logical Instruction – perform logical operation

(34) AND → Logical AND → This instruction bit by bit ANDs the source that may be immediate, reg., m/m to the destination that may be reg. or m/m. The result is stored in the destination operand.

* Both operand cannot be m/m or immediate data.

```
AND AX, 0008H ✓
AND AX, BX ✓
AND AX, [5000H] ✓
```

AND [5000H], [2000H] ✗
AND 20H, 30H ✗

(35) OR → Logical OR → Carry OR operation as above for 1

(36) NOT → Logical Invert → It complement the contents of an operand reg. or m/m location bit by bit.

```
NOT BX ✓
NOT 20H ✗
```

(37) XOR → Logical Exclusive OR → Similar as AND, OR Contr

(38) TEST → Logical Compare Instruction – TEST instruction perf a bit by bit logical AND operation on the two operand The result of the ANDing operation is not available fo

further use/by! flags are affected. Operand may be reg, m/m or immediate data.

TEST AX, BX ✓
TEST BP, [BX][DI] ✓
TEST AL, 80H ✓

* TEST instruction is often used to set flags before a Conditional Jump Instruction.
* PF has meaning only for lower 8 bit of destination.

```
HERE : IN  AL, 20H  ; Read the port with strobe
       TEST AL, 01H ; AND immediate AL with 01 to test AL LSB is 1 or 0.
       JZ   HERE    ; If LSB = 0 read port again.
```

(39) SHL/SAL → Shift logical/Arithmetic left - SHL & SAL are two mnemonics for the same instruction. It shifts each bit in the specified destination some no. of bit position to the left.

CF ← MSB ← ... LSB ← 0

* In case of multiple shift ① CF will contain the bit most recently shifted in from the MSB. Bit shifted into CF previously will be lost. ② The desired no. of shift is loaded into the CL Register.
* As a bit is shifted out of the LSB position, a '0' is put in the LSB position. SAL is similar to SHL.

```
SAL BX, 1 ✓        MOV CL, 02H
SAL BX, 2 X        SAL BX, CL ✓
```

(40) SHR → Shift logical Right perform bit-wise right shifts on the operand word or byte that may reside in a reg. or m/m location by the specified count in the instruction & insert zeros in the shifted position.

0 → MSB → LSB → CF

(41) SAR → Shift Arithmetic Right - It insert the most significant bit of the operand in the newly inserted position. Result is stored in the destination operand.

MSB → MSB → LSB → CF

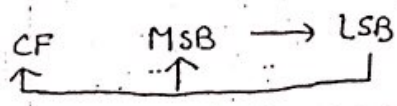
```
SAR AL, 1 ✓
SAR BH, 1 ✓
```

* SAR can be used to divide a signed byte/word by a power of 2.
* SHR can be used to divide an unsigned binary no. by a power of 2.

* SAL or SHL can be used to multiply an unsigned bina by a power of 2.

→ Shifting a binary no. one bit position to the left & putting a 0 in multiplies the no. by 2. / Shift two bit position, multiplies by

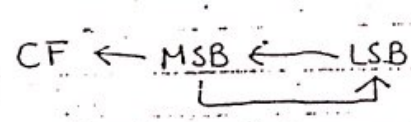
(12) ROR → Rotate right without carry → It rotates the c of the destination operand to the right (bit-wise) eith by one or by the count specified in CL, excluding carry.



ROR BL, 1 ✓

ROR BL, 2 X → MOV CL, 02
ROR BL, CL ✓

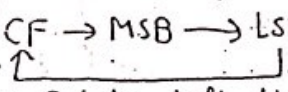
(13) ROL → Rotate left without carry



ROL AX, 1 ✓

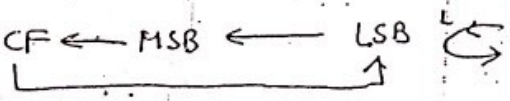
* ROR & ROL can be used to swap the nibble in a byte swap the byte in a word.
It can also be used to rotate a bit into CF, where it be checked & acted upon by conditional Jump JC & JI

(14) RCR → Rotate right through Carry



RCR BX, 1 ✓

(15) RCL → Rotate left through carry



* Count for rotating or shifting is either 1 or is specified using reg. CL.

* RCL & RCR move CF into LSB & MSB of a reg. or m/m location to save CF after addition or subtraction.

String Manipulation Instruction

A series of data bytes or words available in m/m at consecutive locations, to be referred as byte or word st

For referring a string, two parameters are required

(1) Starting or end address of the string
(2) length of the string, usually stored in CX reg.

* The incrementing or decrementing of the pointer, in case of string instruction depends upon direction flag.
* If it is a byte string operation, index reg. are updated by 1
* " " word " " " " " " 2
* The counter in both the cases, is decremented by one.

(46) REP → Repeat Instruction Prefix – REP is a prefix which is written before one of the string instruction. It will cause CX reg. to be decremented & the string instruction to be repeated until CX = 0.

(47) REPE & REPZ → Repeat if equal / zero often we use with the compare string instruction or with scan string instruction. To stop the repetition either CX = 0 or string byte/word not equal.

REPE CMPSB ...

(48) REPNE & REPNZ → Repeat if not equal & Repeat if not zero often used with scan string instruction. It cause the string instruction to be repeated until the compared byte or word are equal (ZF=1) or until CX = 0 i.e. end of string.

REPNE SCASW

(49) MOVSB / MOVSW → Move String Byte or String Word – This instruction copies a byte or a word from a location in the data segment to a location in the extra segment. The offset of source byte/word in DS is in SI register.
" " " " destination " " ES " DI "

* The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter (CX).
* After the MOVS instruction is executed once, the index reg. are automatically updated & CX is decremented.
* The incrementing of SI, DI, if DF = 0
DF = 1 then SI & DI gets decremented.

```
MOV SI, 100H
MOV DI, 200H
CLD
MOV CX, 04H
REP MOVSB
```

(50) CMPS → Compare String Byte or String Word – It is used to compare a word in one string with a word in another string. The DS:SI & ES:DI points to the two string. This byte by byte or word by word comparison continues till a mismatch is found.
If both string are equal CX become '0', ZF is 'I'

~~CMP AL, 0FH~~ .. REPE CMPSB ; Repeat the comparison of string byte until end of string or compared byte are not equal.
~~CMP BH, CL~~ ....

* Compare instructions ~~are often~~ as will cause the pointer in SI & DI to be incremented by 2 after each compare if the direction flag is cleared or decremented by 2 if the direction flag is set.

(51) SCAS → Scan String Byte or String Word for an operand byte or word specified in the reg. AL or AX. String to be scanned must be in ES & DI must contain the offset.

* It is used with a repeat prefix to find the first occurence of a specified byte or word in a string.
* Scanning is repeated as long as the bytes are not equal & the end of the string has not been reached.

```
MOV DI, offset of String
MOV AL, 0DH ; Byte to be scanned into AL
MOV CX, 10  ; counter
CLD
REPNE SCAS String ; Compare byte in String with byte in AL
```

(52) LODS → Load String byte or String Word – It loads the AL/AX reg. by the content of a string pointed to by DS:SI reg. pair. SI is automatically modified depending upon DF. If it is a byte transfer (LODSB), SI is modified by one & if it is word (LODSW). then SI is modified by two.

```
MOV AX, Seg_addres ; segment address of String
MOV DS, AX
MOV SI, offset
CLD
LODS String ; Copy the byte/word from String to AL/AX.
```

(53) STOS → Store String Byte or String Word – It copies a byte from AL or a word from AX to a m/m location in the ES pointed to by DI. DI is automatically incremented or decremented to point to the next string element in m/m. depending upon DF.

```
MOV   AX, seg
MOV   ES, AX
MOV   DI, offset
STD
STOSB     ; replace the byte in string with byte from AL.
```

Control transfer or Branching Instruction – transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly.

Unconditional Control transfer (Branch) Instruction – The execution control is transferred to the specified location independent of any status or condition. The CS & IP are unconditionally modified to new CS & IP.

(54) CALL → Unconditional Call is used to call a subroutine from a main program. Two basic type of CALL

① Near Call is a call to a procedure/subroutine which is in the same code segment as the CALL instruction.

② FAR Call is a call to a procedure which is in different segment from the one that contain the CALL instruction. On execution, this instruction stores the incremented IP (address of next instruction) & CS onto the Stack & loads the CS & IP reg. with the segment & offset address of the procedure to be called.

In Near call it pushes only IP reg. The Near & Far Call are discriminated using opcode.

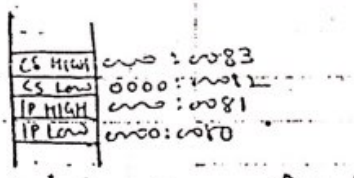
```
CALL BX              → indirect intrasegment Call.
CALL SMART_DIVIDE    → Far intersegment Call. The
                       procedure must be declared Far
                       with smart_divide PROC FAR at the start.
```

(55) RET → Return from the procedure – At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP & CS along with the flags are retrieved into the CS, IP & flag reg. from the stack & the execution of the main program continues further. The procedure may be a near or Far procedure. In case of FAR procedure, the current contents of SP points to IP & CS at the time of return.

* A RET instruction can be followed by a no. for eg. RET 6
In this case the SP will be incremented by an additional Six address after the IP or the IP & CS are popped off the Stack. This form is used to increment the SP over parameters passed to the procedure on the stack.

(56) INT N → Interrupt type N → Type refers to a no. b/w 0 – 255 which identifies the interrupt. When an INT N instruction is executed, the Type bit N is multiplied by 4 & the contents of IP & CS of interrupt service routine will be taken from the hexadecimal multiplication

INT 20H. → 20 x 4 = 80 H INT 8 → 8 x 4 = 0020H



(57) INTO → Interrupt on Overflow – this command is executed when the overflow flag OF is set. This is equivalent to Type 4 interrupt instruction.

(58) JMP → Unconditional Jump – It unconditionally transfer the control of execution to the specified address using 8/16 bit displacement.

o Near JMP – If the destination is in the same code segment as the JMP instruction, then only IP will be changed to get the destination location. It is also known as Short Jump

(59) FAR JMP – If the destination for the jump instruction is in a segment with a name different from that of the segment containing the JMP instruction, then both IP & CS reg. contents will be changed to get to the destination location.

```
JMP BX ; indirect near jump
JMP DWORD PTR [SI]  ; indirect far jump
```

IP ← word pointed by DS:SI
CS ← " " DS:SI+2

Far Jump is also known as Long Jump.

(59) IRET → Return from ISR → When an Interrupt Service Routine is to be called through Interrupt, 8086 pushes the flags, current value of CS & IP onto the stack. It then loads the CS & IP with the starting address of the procedure

Which you write for response to that interrupt.
IRET instruction is used at end of ISR to return execution of the interrupted program. To do this 8086 copies the saved value of IP from the Stack to IP & store value of CS, flag from Stack to CS & flag register.

Int

* RET instruction should not normally be used to return from the interrupt procedure. because it does not copy the flags from the stack back to flag reg.

(60) LOOP → Loop Unconditionally → It is used to repeat a series of instructions some no. of times — No. of times is loaded into CX. Each time the loop instruction executes, CX is automatically decremented by 1.

If CX ≠ 0, execution jump on label

If CX = 0, execution will simply go on to the next instruction after LOOP.

The destination address for the jump lie in range of -128byte to +127 byte from the address of the instruction after LOOP.

```
NEXT : MOV AL,[BX]
       ADD AL,07H
       DAA
       MOV [BX],AL
       INC BX
       LOOP NEXT
```

Conditional Branch Instruction — When these instructions are executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the code is satisfied. If not, the execution continue sequentially.

* Only short Jump (-128 to 127 byte from the address of branch instruction) can be implemented using conditional branch instruction
* Label may represent in term of relative displacement.

(61) JZ/JE → Jump if Equal / Jump if zero — Transfer execution control to address 'Label' if ZF=1.

```
JZ NEXT
```

(62) JNZ/JNE → Jump if not Equal / Jump if not zero → ZF=0

(62) JS → Jump if Signed (Jump if -ve) → if SF=1
(63) JNS → Jump if not Signed (Jump if +ve) → if SF=0
(64) JO → Jump if Overflow → OF=1
(65) JNO → Jump if no overflow → OF=0
(66) JP/JPE → Jump if parity (Even parity) ⇒ PF=1
(67) JNP/JPO → Jump if no parity (Jump if odd parity) ⇒ PF=0
* In JP/JNP → only no. of 1's in the lower 8 bit of data affects the parity flag.

(68) JB/JNAE/JC → Jump if Below / if Carry / if not above or eq
Above & Below are used when referring to the magnitude of unsigned no. ⇒ CF=1

(69) JNB/JAE/JNC → Jump if not below / if above or equal / if no carry ⇒ CF=0

(70) JBE/JNA → Jump if Below or Equal / if not above if CF=1 or ZF=1.

(71) JNBE/JA ⇒ Jump if not below or equal / if above if CF=0 or ZF=0

(73) JL/JNGE → Jump if less than / if not greater than equal
Greater & Lesser are used to refer to the relationship of two signed no. If neither SF=1, nor OF=1

```
CMP BL,39H  ; compare by subtracting 39 from BL
JL  NEXT    ; Jump to label NEXT if BL is more -ve than 39H.
```

(74) JNL/JGE → Jump if not less than / if greater than or equal
(75) JLE/JNG → Jump if less than or equal / if not greater.
(76) if ZF=1 or neither SF nor OF=1

(76) JNLE/JE → Jump if not less than or equal / Jump if Equal
if ZF=0 or atleast any one of SF & OF = 1
(Both SF & OF ≠ 0)

(77) LOOPZ/LOOPE → Loop while ZF=1 / while CX ≠ 0
Two ways to exit the loop are CX=0 OR ZF=0.
The no. of times the instruction sequence is to be repeated is loaded into CX. Each time the loop executes CX is automatically decremented by 1.

(78) LOOPNZ/LOOPNE → Loop while CX ≠ 0 & ZF=0
If CX=0 after the autodecrement or if ZF=1, execution will simply go on to the next instruction after LOOPNE/LOOPNZ

Ex:-
```
        MOV BX, offset_array
        DEC BX
        MOV CX, 100
NEXT:   INC BX
        CMP [BX], 0DH
        LOOPNE NEXT
```

Flag Manipulation & Processor Control Instruction - These instructions control the functioning of the available H/w inside the processor chip.

→ Flag manipulation instruction directly modify the some of the flags of 8086.

→ Machine Control Instructions control the bus usage & execution.

(79) CLC → Clear the Carry flag → reset the CF to 0.

(80) CMC → Complement the carry flag → invert CF

(81) STC → Set Carry flag → set the CF to 1.

(82) CLD → Clear the direction flag → if DF=0, SI & DI will automatically be incremented, when one of the strings instruction executes.

(83) STD → Set the Direction flag → if D=1, SI & DI get incremented, during string instruction.

(84) CLI → Clear the Interrupt flag - if IF=0, 8086 will not respond to an interrupt signal on its INTR I/P. CLI has no effect on NMI I/P.

(85) STI → Set the interrupt flag - if IF=1, enable the INTR interrupt. The instruction will not take effect until after the next instruction after STI.

(86) WAIT → Wait for test Signal or Interrupt Signal - When this instruction executes, 8086 enters an idle condition in which it is doing no processing. 8086 will stay in this idle state until 8086 TEST I/P pin is made low or until an interrupt signal is received on the INTR or NMI interrupt I/P pins. 8086 will return to idle state after interrupt service procedure executes. It returns to the idle state bcz the address of the WAIT instruction is the address pushed on the stack when 8086 responds to interrupt request.

* WAIT is used to synchronize the 8086 with external H/w such as 8087 maths coprocessor.

(87) HLT → Halt Processing - It causes the 8086 to stop fetching & executing instructions. The only way to get the processor out of the Halt state are with an interrupt signal on the INTR pin, NMI pin or a reset signal on reset pin.

(88) NOP → No operation - It uses three clk cycles & increment the IP to point to the next instruction.
* It can be used to increase the delay of a delay loop.

(89) ESC → Escape to external device like 8087 - It is used to pass instructions to a coprocessor such as 8087 which shares the address & data bus with an 8086. Instructions for this coprocessor are represented by 6-bit code embedded in the escape instruction. When the 8086 fetches an ESC instruction, the coprocessor decode the instruction & carries out the action specified by 6-bit code specified in the instruction. 8086 treats the ESC instruction as NOP. In some cases 8086 will access a data item in m/m for the coprocessor.

(90) LOCK → Bus lock Instruction Prefix - Microcomputer system may have several μp. Each microprocessor has its own local bus & m/m. The individual μp are connected together by a system bus so that each can access system resources such as disk drives or m/m. Each μp takes control of the system bus only when it needs to access some system resources. LOCK prefix allow a μp to make sure that another processor does not take control of the system bus while it is in the middle of critical instruction which uses the system bus.

eg → LOCK XCHG Semaphore, AL ; XCHG require two bus accesses. Lock prefix prevents another processor from taking control of the system bus b/w the two accesses.

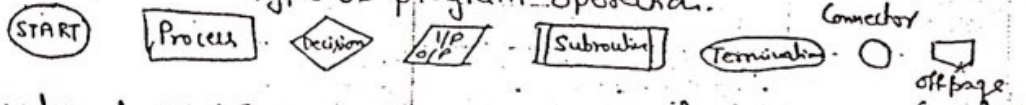
Program Development Steps

① **Defining the Problem** – Carefully think about the problem, that we want the program to solve.
Write the operation that we want the program to do it.

② **Representing program Operation** – Formula or sequence of operation used to solve a programming problem is called algorithm of the program.
Two ways to represent the algorithm

Flowchart | Structured Programming & Pseudocode overview

2.1 **Flowchart** → It uses the graphic shapes to represent different type of program operation.



disadvantages : ① We can't write much information in little boxes.
② Do not present information in a very compact form.
③ For complex problems, flowcharts tends to spread out over many pages & hard to follow back & forth b/w pages.
④ The overall structure of the program tends to get lost in the details.

2.2 **Structured Programming & Pseudocode** → For professional programmer team to work, a systematic approach & standardize tools are required.

Top-down design → large programming module is break into major module & then major module is broken down into smaller module. The division is continued until the steps in each module are clear.

Bottom-up design → Each programmer starts writing low level modules & hopes that all the pieces will eventually fit together.

Any desired program operation could be represented by 3 basic steps
① Sequence → doing a series of action.
② Selection/decision → choosing b/w two alternative action.
③ Repetition/Iteration → repeating a series of action until some condition is or is not present.

Advantage of machine language
- m/m control is directly in the hands of the programmer & can manage the m/m of system efficiently

disadvantage of machine language
- Programming is tedious
- Chances of human error are more
- Thorough technical knowledge of the processor architecture & instruction set is required.

Advantage of Assembly language
- The address values of the constants can be identified by labels.
- Documentation facility is available

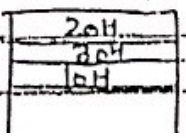
Assembler directives are direction to the assembler, not instruction for the 8086. They are also known as Pseudoinstruction. It helps the assembler to correctly understand the assembly language programs to prepare the codes.
Assembler needs some hints from the programmer, ie the required storage for a particular constant or variable, logical name of the segment, type of different routine & modules, end of file etc. These type of hints are given to the assembler using some predefined alphabetical strings called assembler directives.
Another type of hint which helps the assembler to assign a particular constant with a label or initialize particular m/m location, or label with constant is an operator. Unlike the assembler directives, the operator can perform arithmetic & logical tasks.

① DB → Define Byte is used to reserve byte or bytes of m/m locations in the available m/m. It also initializes the reserved m/m bytes with the ASCII codes of the character specified as a String.

eg. RANKS DB 01H, 02H, 03H, 04H — Declare array of 4 bytes named RANKS & initialize 01, 02, 03, 04 to them.

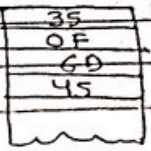
eg. Message DB 'Good Morning' — Declare array of 12 bytes & initialize with ASCII code for letters in string.

eg. Pressure DB 20H, 30H, 10H



② DD – Define Doubleword is used to declare a variable of type doubleword (4 byte or 2 word)
e.g. MEM DD 465D0342H, 2a30h012H This statement will define a 2 doubleword named MEM & initialize the specified value when the program is loaded into m/m to be run.

③ DW – Define Word (or double byte) is used to allocate one or more data in word (16-bit) format.
e.g. MEM DW 0F35H, 456DH



④ DQ – Define quadword (8 byte) is used to direct the assembler to reserve 4 words (8 byte) of m/m for the specified variable & may initialise it with the specified value.
e.g. Big-word DQ 10 2A 3B 26 12 10 01 1A H

⑤ DT → Define Ten Bytes is used to define a variable which is 10 bytes in length. It is often used when declaring data arrays for the 8087.
eg. Packed-BCD DT 10 23 A0 26 B3 40 26 839A 12 H

⑥ DUP → Duplicate is used to assign value to the locations & initialize several locations. It has following format
Name Data-Type NUM DUP(value)
Data-Type is type of data i.e. DB, DW, DD, DT
NUM is the no. of times duplication
e.g. Table DW 10 DUP(0) → assembler reserve an array of 10 words of m/m & initialize all 10 words with 0 & the array name is Table.
eg. TEMP DB 10 DUP (20H, 40H, 60H, 80H)
The data 20, 40, to, 80H are stored in a m/m & then 10 such sets are created in m/m. & M/m is identified as TEMP.

⑦ ASSUME → ~~Assume~~ Logical segment name for different segment used in the program. We can define more than one CS, DS ~~area~~ But only one of each type can be active at a time. Assume tell the assembler, which segment is to be used as an active segment at any instant & with respect to which it has to calculate the offset of the variable or instruction.

NUM1 DB 20H
RES1 DB 30H

Usha Ends

⇒ Assume DS: Usha → Then assembler will consider Usha as data segment.

Assume is must at the starting of each assembly language program.

(8) END – End program → End directive is put after the last statement of a program to tell the assembler that this is the end of the program module. Assembler will ignore any statements after an END directive.

(9) ENDP → End Procedure is used along with the name of the procedure to indicate the end of a procedure to the assembler. eg → Square PROC

=

Square EndP

(10) ENDS → End segment is used with the name of the segment to indicate the end of that logical segment.

eg – Code Segment

=

Code Ends

(11) EQU → Equate is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it will replace the name with the value or symbol you equated with that name.

eg – Correction_factor EQU 03H

In our program, if we have instruction ADD AL, correction_factor Then assembler will code it as ADD AL, 03H

advantage – If the correction_factor is used 20 times in a program & we want to change the value, then we just need to change the value with EQU statement & reassemble the program.

(12) EVen → Align on even m/m address – Even directive tell the assembler to increment the location counter to the next even address.

advantage – A series of words can be read much more

quickly if they are at even address.
When Even is used in DS the location counter will simply be incremented to the next even address if necessary.

e.g. Data Here Segment
Sales DB 9 DUP(0)
Even ; → increment location to 000AH
Records DW 100 DUP(0)
Data Here Ends

(13) ? → Uninitialize value – It indicates a value that the assembler allocate but does not initialize.

Eg. MEM-1 DB ?,?,? → allocate three uninitialized bytes.
MEM-1 DB 20 DUP(?) → allocate 20 uninitialized bytes.

(14) EXTRN – External – It is used to tell the assembler that the names or labels following the directives are in some other assembly module. Public directive is used alongwith EXTRN directive

(15) PUBLIC – In other module, where the name / procedure / labels actually appear / they must be declared using the public directives.

e.g. Module1 Segment
Public Factorial FAR
Module1 Ends
Module2 Segment
EXTRN Factorial FAR
Module2 Ends

If one wants to call a procedure Factorial appearing in Module1 from module2, in module1, it must be declared public & in module2, it must be declared external using EXTRN directives.

(16) GLOBAL directives can be used in place of a Public or in place of EXTRN directive. Global directive is used to make the symbol available to other module

eg. Global DIVISOR :Word tell the assembler that divisor is a variable of type word & it can be accessed from the other assembly module.

(17) Group directive is used to tell the assembler to group the logical segments named after the directive into one logical group segment. The assembler passes on information to the linker/loader to form the code such that

the group declared segments or operands must lie within a 64Kbyte m/m segment. Thus all such segment & label can be addressed using the same segment base.

e.g. Program Group Code, data, Stack It direct the loader/linker to prepare an EXE file such that Code, data, & Stack segment must lie within a 64Kbyte m/m segment is named as Program. Now for the assume statement, we can use the label program

Assume CS: Program, DS: Program, SS: Program

(18) INCLUDE — Include Source Code from file — This directive insert a block of source code from the named file into the current source module. This shorten the source code.

With the Include directive, we can construct text files which Contain common Macros; equates, Constant source code & other items.

eg. Include file-name

(19) Label directive is used to give a name to the current value in the location counter. It must be followed by a term which specifies the type you want associated with that name.

* If the label is going to be used as the destination for a jump or a call, then the label must be specified as type near or type far.
* If the label is going to be used to reference a data item, then the label must be specified as type byte, word, or doubleword.
* A Far Jump Cannot be made at a normal label with a colon. for that we have to use like this → Continue Label Far

eg- Data_1 Label Byte Far will assigned a label Data_1 & its type will be byte & far.

(20) Length is an operator which tells the assembler to determine the no. of elements in some named data items, such as a string or an array.

eg. MOV CX, LENGTH String1

It will determine the no. of element in string1 & load into CX reg.

(21) Name directive is used to assign a name to an assembly language program module.

eg. NAME PCB.

(22) ORG — Originate directive allows us to set the location Counter to a desired value at any point in the program.

The $ is often used in ORG statements to tell the assembler to make some change in the location counter relative to its current value.

eg. ORG $+100 tells the assembler to increment the value of location counter by 100 from its current value.

eg ORG 2000H → to set the location counter to 2000 H.

(23) PROC → Procedure – It is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive, the term near or far is used to specify the type of the procedure. eg – SMART_DIVIDE PROC FAR

(24) PTR → Pointer – PTR operator is used to assign a specific type to a variable or to a label. It is necessary to do this in any instruction where the type of the operand is not clear.
eg. INC [BX] → assembler does not know whether to increment the byte pointed by BX or word pointed to by BX.
So INC BYTE PTR [BX] is used to tell Byte

* PTR operator can be used to override the declared type of a variable.

* For near jump (Indirect) JMP Word PTR [BX] { (For direct) JMP NEAR PTR Label
For far jump (Indirect) JMP DWord PTR [BX] { JMP FAR PTR Label

(25) Offset is an operator which tells the assembler to determine the offset or displacement of a named data item or procedure from the start of the segment which contains it. This operator is usually used to load the offset of a variable into a register so that the variable can be accessed with one of the indexed addressing mode.
eg. MOV BX, OFFSET PRICES → determine the offset of the variable prices from the start of the segment in which Prices is defined & load into BX reg.
then ADD AL, [BX] can be used to add a value from prices to AL.

(26) Segment directive is used to indicate the start of a logical segment. Preceding the Segment directive is the name of the segment.
e.g. Code_HERE Segment
Segment may be assigned a type like PUBLIC (can be

used by other module of program while linking) or GLOBAL (can be accessed by any other module).

(27) SHORT operator is used to tell the assembler that only 1-byte displacement is needed to code a Jump instruction. Using the short operator saves 1 byte of m/m by telling the assembler that it needs to reserve only 1 byte for this particular Jump. The destination must be in the range of -128 byte to +127 byte from the address of the instruction after the jump.

eg. JMP SHORT HERE

(28) TYPE operator tells the assembler to determine the type of a specified variable. Assembler actually determines the no. of bytes in the type of the variable.

Type operator directs the assembler to decide the data type of the specified label & replace the TYPE label by the decided data type. For eg - Word type variable, data type is 2. for byte type, it is 1.

eg. MOV AX, Type STRING

If STRING is a word array then move 0002H in AX.

eg. Write a program to find out whether a given byte is in the string or not. If it is in the string, find out the relative address of the byte from the starting location of the string.

```
Assume CS: Code, DS: Data
Code Segment
        MOV AX, Data
        MOV DS, AX
        MOV ES, AX
        MOV CX, Count
        MOV DI, offset String
        MOV BL, 00H
        MOV AL, Byte1
SCAN :  NOP
        SCASB
        JZ Below1
```

display, hard disk, floppy disk controller ...

INT 21H. The routine required to refer to these resources are written as ISR for 21H interrupt. Under this interrupt, specific resources is selected depending upon the value in AH reg. For eg 09 for CRT display, 0A for keyboard, 4C for DOS prompt.

```
        INC BL
        LOOP SCAN1
Below:  MOV AH,4CH
        INT 21H
        Code Ends
Data Segment
        Byte1 EQU 25H
        Count EQU 06H
        String DB 12H,13H,20H,20H,25H,21H
Data Ends
```

Eg. WAP to Convert BCD no. 0 to 9 to their equivalent 7segment. Code using the look-up table technique. Assume the codes are stored sequentially in Codelist at the relative addresses from 0 to 9. The BCD no. (CHAR) is taken in AL.

```
Assume CS:Code, DS:Data
Data Segment
        Codelist DB 34,25,56,45,23,12,19,24,21,00
        CHAR EQU 05
        CODEC DB 0H DUP(?)
Data Ends
Code Segment
        MOV AX,Data
        MOV DS,AX
        MOV BX,OFFSET Codelist
        MOV AL,CHAR
        XLAT
        MOV BYTE PTR CODEC,AL
        MOV AH,4CH
        INT 21H
Code Ends
        End
```

* No. of clk cycle required to read/write from/to m/m depends on whether the 1st byte of the word is at even or at an odd address.
* No. of clk cycle depends on the addressing mode used to access that byte.
* In a microcomputer system, the wait states can be inserted during each m/m access, this will increase the no. of clk cycle required.

eg. Write a program to compare two string data byte

```
Data Segment
        String1 DB 'EMPTY'
        SL  EQU  String1
        NEQU DB 'STRING ARE NEQUAL'
        EQUA DB 'STRING ARE EQUAL'
Data Ends
Extra Segment
        String2 DB 'EMPTY'
Extra Ends
Code Segment
        Assume CS: Code, DS: Data, ES: Extra
        MOV AX, data
        MOV DS, AX
        MOV AX, Extra
        MOV ES, AX
        MOV SI, OFFSET String1
        MOV DI, OFFSET String2
        CLD
        MOV CX, Length SL
        REP CMPSB          ; until CX=0
        JZ LOOP
        MOV DX, OFFSET NEQU
        INT 21H
        JMP EXIT
LOOP :  MOV DX, OFFSET EQUA
        INT 21H
EXIT :  INT 03H
```

at INT 21 → a subroutine to display the string is mentioned.

Write a program to find out whether the string is a palindrome

```
Data Segment
    STRING1 DB 'NITCBATIN'
Data Ends
Code Segment
    Assume CS:Code, DS:Data
    MOV AX,Data
    MOV DS,AX
    LEA SI, string1
    MOV BX,08H
    XOR BP,BP
    MOV CL,05H
    XOR CH,CH
Loop: MOV AL,[SI+BP]
    MOV AH,[SI+BX]
    CMP AL,AH
    JE Dum
    DEC CH
    JMP EXIT
Dum: INC CH
    INC BP
    DEC BX
    DEC CL
    JNZ Loop
EXIT: INC 03H
```

If CH = FF, i.e. NO
CH = 05 i.e. YES

Q. Write a program to generate a delay of 100 ms using 8086 that runs on 10 MHz frequency

```
PROC Delay Local
Assume CS:Codep
Codep Segment
    MOV CX,BA03H
Wait: DEC CX
    NOP
    JNZ Wait
    RET
Delay EndP
```

Req. delay $T_d = 100$ ms

$T = \frac{1}{F} = \frac{1}{10 \times 10^6} = 0.1\ \mu s$

	State
MOV CX, count	4
Here: DEC CX	2
NOP	3
JNZ Here	16

No. of clk cycle for execution of the loop once = 2+3+16
$n = 21$

Time required to execute the loop once $= nT$
$= 21 \times 0.1 \approx 2.1\ \mu s$

$N = \frac{T_d}{DT} = \frac{100 \times 10^{-3}}{2.1 \times 10^{-6}} = 47.619 \times 10^3$
$= 47613$
$\approx$
BA03H

A Basic 8086 Microcomputer System (Minimum Mode)

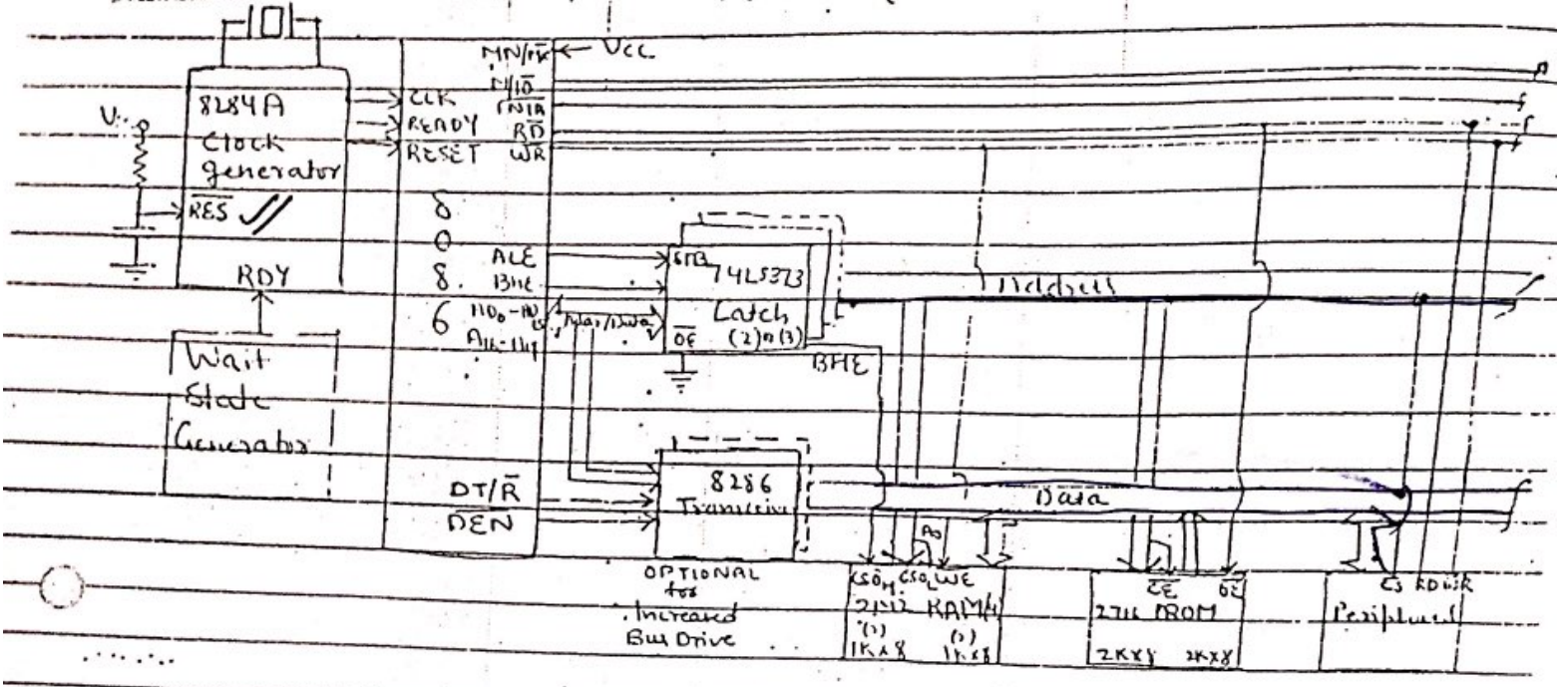


Fig: Block diagram of a simple 8086 microcomputer.

In a minimum mode 8086 system, the up MN/MX pin is connected to Vcc (logic 1). In this mode, all the control signal are given out by the up chip itself. There is a single up in the minimum mode system. The remaining components in the system are latches, transceiver, clock generator, memory & I/O devices. Some type of chip selection logic may be required for selecting m/m or I/O devices, depending upon the address map of the system.

Latches – External latches 74LS373 octal devices are used to separate the valid address from the multiplexed address/data signal & hold it during the rest of operation. Generally they are buffered O/P D-type flip flops & are controlled by the ALE signal generated by 8086. Once the address is stored on the O/P of the latches, 8086 can use the bus for reading or writing data.

Transceivers – are the bidirectional buffer & sometimes they are called data amplifiers. They are required to separate the valid data from the time multiplexed address/data signal. They are controlled by two signals, DEN & DT/R. The DEN signal indicates that the valid data is available on the data bus, DT/R indicates the direction of data i.e. from/to the processor. When DT/R is asserted

high, the buffers will be set up to transmit data from the 8086 to ROM, RAM or ports. When DT/$\overline{R}$ is asserted low, the buffers will be set up to allow data to come into the 8086 from ROM, RAM or ports.

* The buffers used on the data bus must have 3-state o/p so the o/p can be floated when the bus is being used for other operation. The 8086 asserts the $\overline{DEN}$ signal to enable three state o/p on data bus buffers at the appropriate time in an operation.

Clock Generator – uses a crystal to produce the stable frequency clock signal which steps the 8086 through execution of its instruction in an orderly manner. The 8284A also synchronizes the RESET signal & the READY signal with the clock so that these signals are applied to the 8086 at the proper times. When the RESET I/P is asserted, 8086 goes to address FFFF0H to get its next instruction.

Memory – The system contains the m/m for the monitor & user program storage. Usually, EPROM are used for monitor storage, while RAM for user's program storage.

I/O devices – A system may contain I/O devices for communication with the processor as well as some special purpose I/O devices.

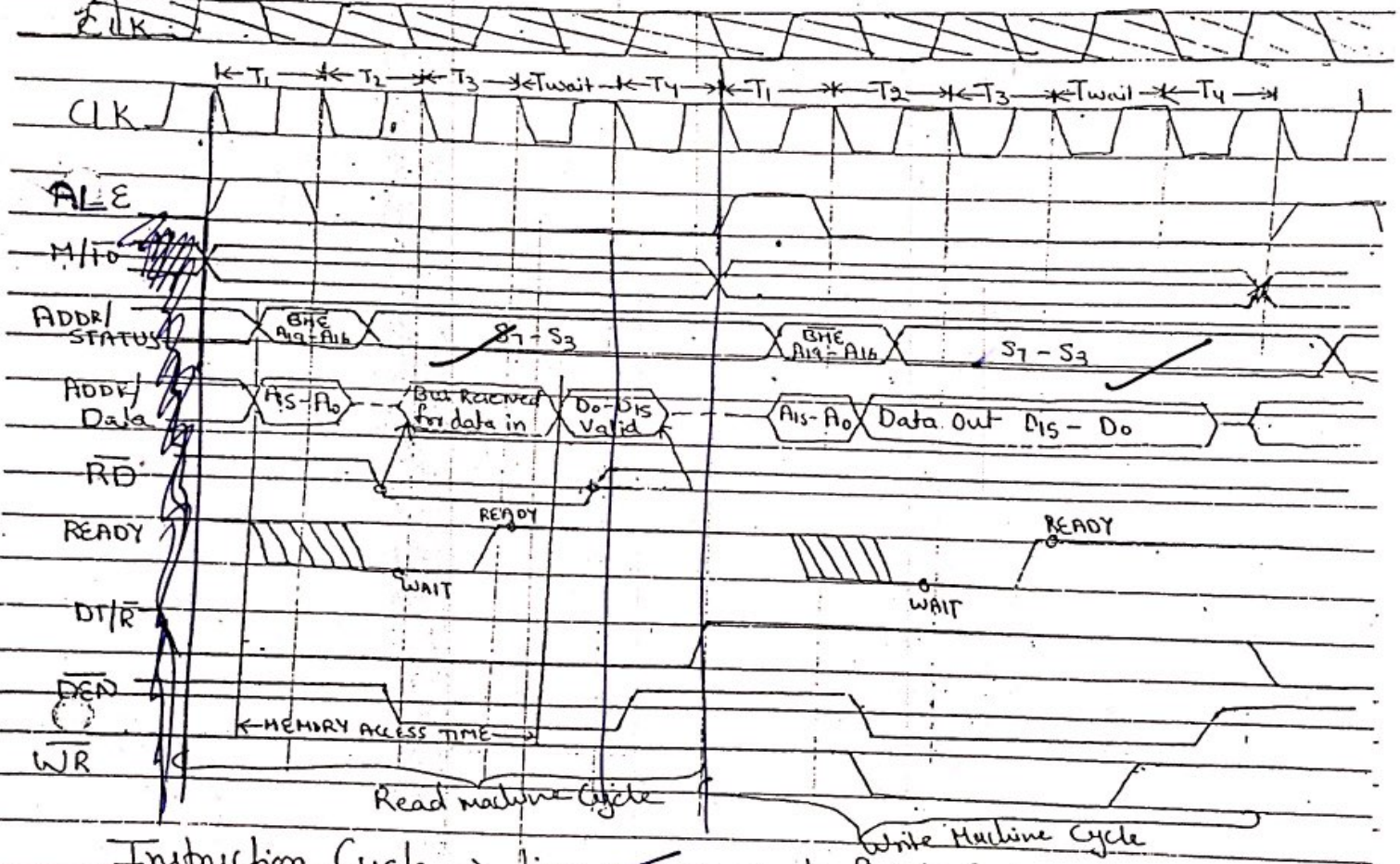
8286 Transceiver are bidirectional 3 state buffers. For a very small system these buffers are not needed, but as more devices are added to a system, they become necessary.

Reason → Most of the devices – such as ROMs, RAMs & ports – connected on up buses have MOS I/P, so on a dc basis they don't require much current. However each I/P or o/p added to the system data bus, for eg – acts like a capacitor of a few picofarads connected to ground. In order to change the logic state on these signal lines from low to high, all this added capacitance must be charged. To change the logic state to a low, the capacitance must be discharged. If we connect more than a few user devices on the data bus lines, the 8086 o/p cannot supply enough current drive to charge & discharge the circuit capacitance fast enough. Therefore, we add external high-current drive buffer to do this job.

OPTIONAL

The working of the minimum mode configuration system can be better described in terms of the Timing Diagram.
The Opcode fetch & Read cycles are similar, Hence the timing diagram can be categorized in two parts, (1) timing diagram for read cycle (2) timing diagram for write cycle.

Timing Diagram illustrate graphically how the signal changes in response to other signals with time.



Instruction Cycle → Time requires to fetch & execute an entire instruction by µP is referred as an instruction cycle. Instruction cycle is made up of machine cycle & machine cycle is made up of T-states.

Machine Cycle → Time required to execute a basic µP operation such as reading a byte from m/m or writing a byte to a port.

T-state → Time period of one clock cycle & is determined by the frequency of the clock signal.

timing diagram, think of timing as a vertical line moving from left to right across the diagram.

8086 Bus Activity During a Read Machine Cycle

During T_1 of a read machine cycle the 8086 1st asserts the M/$\overline{IO}$ signal. It will assert this signal high if it is going to do a read from m/m during this cycle & it will assert M/$\overline{IO}$ low if it is going to do a read from a port during this cycle.

* Crossed waveform is used because the signal may be going low or high for a read cycle. The pt. where the two waveform cross indicates the time at which the signal become valid for the machine cycle.

* For rest of the timing diagram, crossed line X are used to represent the time when information on a line or group of line is changed.

After asserting M/$\overline{IO}$ 8086 sends out a high on the ALE. This signal is connected to the enable I/P (strobe STB) of TLS373 octal latches to enable them. After ALE gets high, it sends out the addresses (on AD0-AD15, A16-A19) of the M/M location that it wants to read. This information also passes through the latches to their O/P. Then 8086 makes ALE low, which disable the latches. The address held on the latch O/P travels along the address bus to m/m & port devices.

ADDR/DATA waveforms cross represent the time at which the 8086 has put a valid address on these lines.

After ALE goes low, the address information is held on the latches, so the 8086 no longer needs to send out the addresses. The 8086 floats the AD0-AD15 lines so that they can be used to I/P data from m/m or port. It is shown by dashed lines. At the same time, 8086 also removes the BHE & A16-A19 information from the upper lines & sends out some status information on these lines.

8086 is now ready to read data from the addressed m/m location or port, so near the end of state T_3, the 8086 asserts its $\overline{RD}$ signal low. $\overline{RD}$ signal is used to enable the addressed m/m devices or port devices.

When enabled, the addressed device will put a byte or word of data on the data bus. This cause effect relation is shown by an arrow going from falling edge of RD.

Memory access time — The time it takes for the m/m to o/p valid data after it receive an address & RD signal. If the m/m access time for a m/m device is too long, the m/m will not have valid data on its o/p, then 8086 will treat whatever garbage happens to be on the data bus as valid data & go on with the next machine cycle.

If READY pin is high, 8086 operates normally. If the READY I/P is made low at the right time in a machine cycle, the 8086 will insert one or more WAIT states b/w T_3 & T_4 in that machine cycle. An external H/w device is set up to pulse READY low before the rising edge of the clock in T_2. After the 8086 finishes T_3 of the machine cycle, it enters a WAIT state. M/$\overline{IO}$ & $\overline{RD}$, addresses on AD_0-AD_{15}, A_{16}-A_{19} do not change during the WAIT state, T_{WAIT}. If the READY I/P is made high again during T_3 or during T_{WAIT} then after one WAIT state the 8086 will go on with the regular T_4 of machine cycle. If the READY I/P is still low at the end of a WAIT state, 8086 will continue inserting WAIT states until the READY I/P is high again. During T_1 of machine cycle the 8086 asserts DT/$\overline{R}$ low to put the data buffers in the receive mode. Then after the 8086 finishes using the data bus to send out the lower 16 address bits, it asserts $\overline{DEN}$ low to enable the data bus buffers. The data put on the data bus by an addressed port or m/m will then be able to come in through the buffers to the 8086 on the data bus.

8086 bus activities during a Write machine cycle

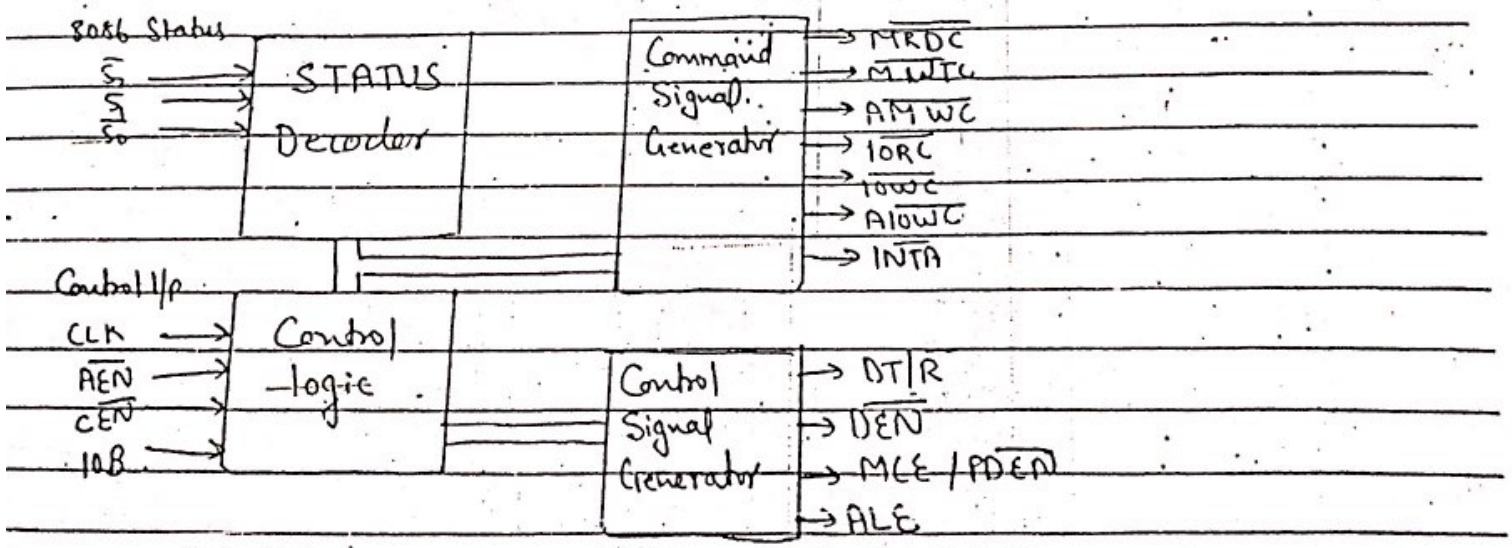
During T_1 of a write machine cycle the 8086 asserts M/$\overline{IO}$ if the write is going to be to a port, & it asserts M/$\overline{IO}$ high if the write is going to be to a m/m. 8086 raises ALE high to enable address latches & $\overline{BHE}$ get low & the address will be writing to on AD_0-AD_{19}. When writing to port, lines A_{16}-A_{19} will always be low because 8086 only sends out 16-bit port addresses. After the address information is latched, the 8086 removes the address information from AD_0-AD_{15} & o/p the desired data on the data bus, while ALE is low.

then assert its $\overline{WR}$ signal low. The $\overline{WR}$ signal is used to him on the m/m or port that the data is to be written to. After the addressed m/m or port has had time to accept the data from the data bus, 8086 raises the $\overline{WR}$ signal line high again & floats the

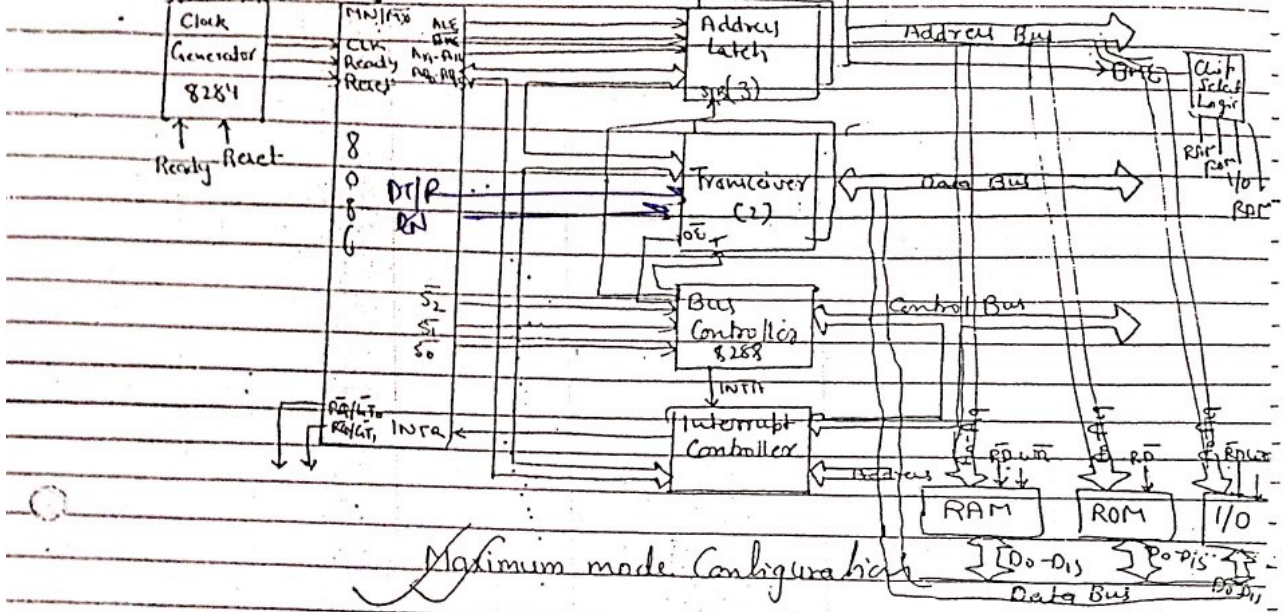
data bus.
If the m/m or port device cannot accept the data word within a normal machine cycle, external H/W can be set up to pulse the READY I/P low each time that m/m or port is addressed.
If the READY I/P is low before the end of wait state, 8086 will insert another wait state. So the addressed device has one or more extra clock cycles to accept the data from the data bus.
If the system is large enough to need buffers on the data bus, then DT/$\overline{R}$ will be connected to the direction I/P on the buffers. During Write, 8086 asserts DT/$\overline{R}$ high to put the buffers in the transmit mode.
When 8086 asserts $\overline{DEN}$ low to enable the buffers, data o/p from 8086 will pass through the buffers to the addressed port or m/m location.

Maximum Mode 8086 System

When MN/$\overline{MX}$ pin is connected to ground, 8086 is operated in maximum mode. In the maximum mode, there may be more than one µp in the system. Some pins of 8086 IC have special functions in maximum mode. Processor derives status signal $\overline{S_2}$, $\overline{S_1}$ & $\overline{S_0}$. Another chip called bus controller derives the control signal using the status information. Other components in the system are the same as in the minimum mode system.



Block diagram of Bus Controller 8288



Maximum mode Configuration

The basic function of 8288 chip is to derive control signal like $\overline{RD}$ & $\overline{WR}$, DT/$\overline{R}$, $\overline{DEN}$, ALE etc using the information made available by the processor on the status line.
Bus controller chip has I/P line $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ & CLK which are driven by CPU.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Processor State	Command
0	0	0	Interrupt acknowledge	$\overline{INTA}$
0	0	1	Read I/O port	$\overline{IORC}$ (I/O Read Command)
0	1	0	Write I/O port	$\overline{IOWC}$ (I/O write Command), Advanced $\overline{AIOWC}$
0	1	1	Halt	None
1	0	0	Code access	$\overline{MRDC}$ (m/m read command)
1	0	1	Read M/m	$\overline{MRDC}$
1	1	0	Write m/m	$\overline{MWTC}$ (m/m write Command), $\overline{AMWC}$
1	1	1	Passive	None

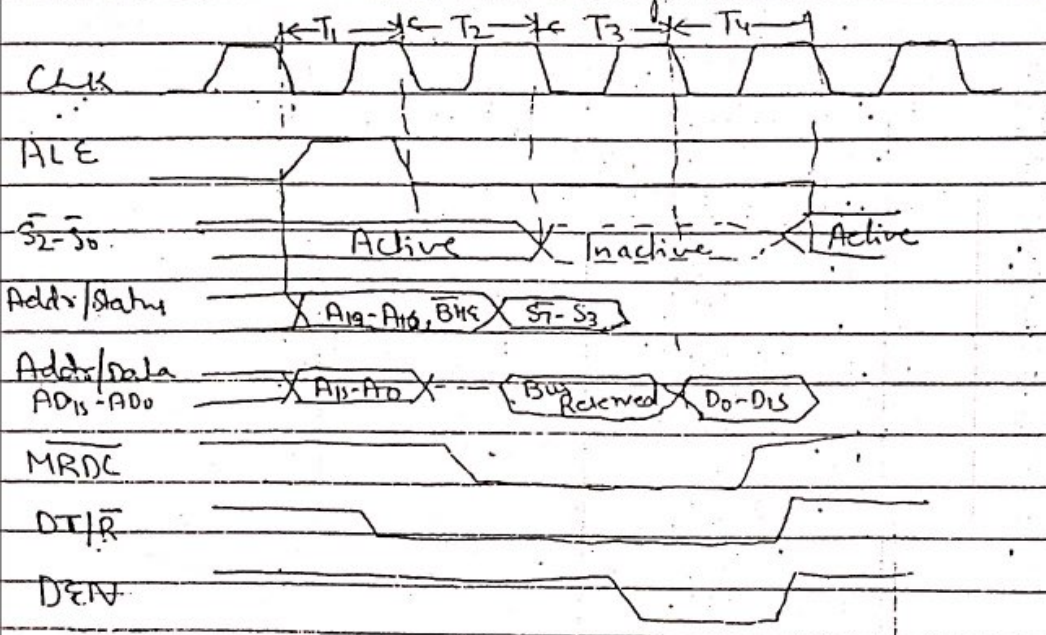
8288 works in two operating mode.

① I/O Bus mode — When the IOB pin is high, 8288 work in I/O Bus mode. It is used in multiprocessor system, when processor has its own dedicated I/O or peripheral. In this mode, all I/O command line are always enabled & independent of AEN signal. Transceiver are enabled by PDEN & DT/$\overline{R}$

2) System bus mode — When IOB pin is low, then 8288 works in System bus mode. It is used in a multiprocessor environment when only one bus exists & both I/O & m/m are shared by more than one processor. Both m/m & I/O command wait for bus arbitration through AEN signal.

MCE/PDEN → If IOB is grounded, it acts as a Master Cascade Enable to control 8259A (Interrupt Controller), else it acts as Peripheral data enable used in the multiple bus configuration.

AIOWC & AMWTC serve same purpose as IOWC & MWTC respectively, but are activated one clock cycle earlier than the IOWC & MWTC signals.



M/M Read in maximum mode

CEN	IOB	AEN	Description
1	1	X	I/O Bus mode
1	0	1	System Bus mode, but control signals disabled
1	0	0	System Bus mode, all control signal enabled
0	X	X	Open circuited, all command & PDEN are disabled.

8086 Interrupts & Interrupt Response

Interrupt means to break the sequence of operation. While the CPU is executing a program, an interrupt break the normal sequence of execution of instruction diverts its execution to some other program called Interrupt service Routine. After executing the ISR, control is transferred back again to the main program which was being executed at the time of interruption.

An 8086 interrupt can come from any one of three sources:

(1) An external signal applied to the nonmaskable interrupt (NMI) I/P pin or to the interrupt (INTR) I/P pin. An interrupt caused by a signal applied to one of these I/Ps is referred to as a Hardware Interrupt.

(2) Execution of the interrupt instruction, INT. This is referred to as a software Interrupt.

(3) Some error produced in the 8086 by the execution of an instruction. e.g - divide by zero interrupt.

At the end of each instruction cycle, the 8086 checks to see if any interrupts have been requested. If an interrupt has been requested, the 8086 responds to the interrupt by stepping through the following series of major actions.

1. It decrement the stack pointer by 2 & pushes the flag reg. on Stack.
2. It disable the 8086 INTR interrupt I/P by clearing the interrupt flag (IF) in the flag register.
3. It reset the trap flag in the flag register.
4. It decrement the SP by 2 & pushes the current code segment register contents on the stack.
5. It decrement the SP by 2 & pushes the current IP contents on the stack.
6. It does an indirect far jump to the start of the procedure you wrote to respond to the interrupt.
7. An IRET instruction at the end of ISR returns execution to the main program.

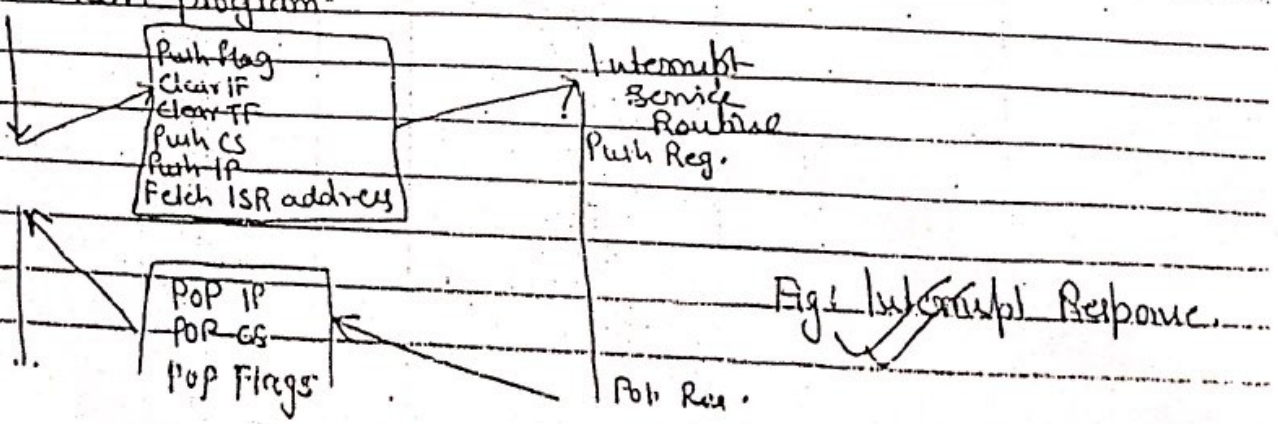


Fig: Interrupt Response.

When 8086 respond to an interrupt, it goes to four m/m location to get the CS & IP values for the start of the interrupt service procedure. In an 8086, the 1st 1KByte of m/m from 00000H to 003FFH, is set aside as a table for storing the starting addresses of ISR, the table can hold the starting addresses for upto 256 interrupt procedure.

The Starting address of an ISR is often called the Interrupt Vector or Interrupt Pointer, so the table is referred to as the Interrupt Vector Table for Interrupt Pointer Table. Each doubleword Interrupt vector is identified by a number from 0 to 255. Intel call this number the type of the interrupt. The lowest 5 type are dedicated to specific interrupts. Type 5 to 31 are reserved by Intel for use in more complex up, such as 80286, 80386 -- . The upper 224 interrupt type 32 to 255 are available for you to use for H/w or S/w interrupt.

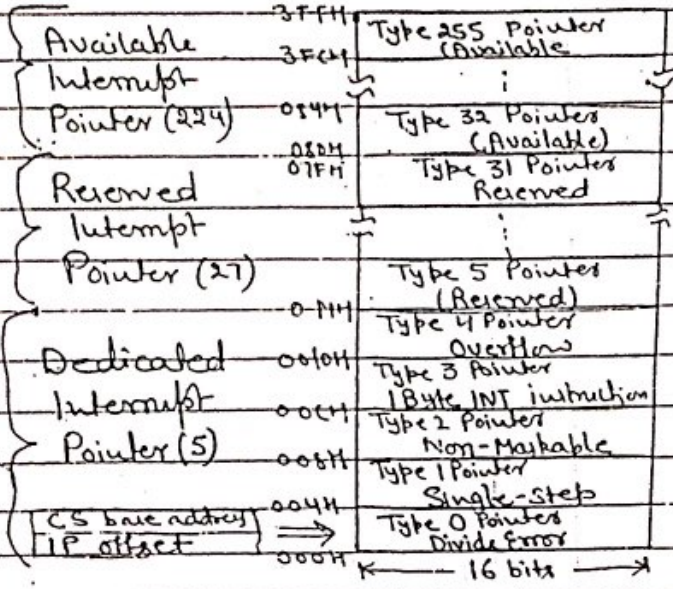


Fig: 8086 Interrupt Vector Table

The vector for each interrupt type require 4 m/m locations. Therefore when the 8086 responds to a particular type interrupt, it automatically multiplies the type by 4 to produce the desired address in the vector table. It then goes to that address in the table to get the starting address of the ISR.

Type 0 Interrupt – Divide by 0 → An attempt is made to divide a 32 bit number or a 16-bit no. by zero. The result of dividing by 0 is infinity, which is considered too large to fit in AX or AL. Whenever the quotient from a DIV or IDIV operation is too large to fit in the result register, the 8086 will automatically do a type 0 interrupt.

004H
000H

When In response to this type 0 interrupt, 8086 proceed as follows
(1) decrement the SP by 2 & push flag reg. to stack.
(2) Clear IF & TF
(3) Decrement the SP by 2 & pushes CS value of return address on the stack & again decrement SP by 2 & pushes IP value of return address on the stack.
(4) 8086 get the starting address of ISR from the type 0 location in Interrupt Vector Table. It get the new value for CS from 00002H & 00003H & new value IP from 00000H & 00001H.
(5) After the starting address of the procedure is loaded into CS & IP, the 8086 then fetches & executes the 1st instruction of the procedure.
(6) At the end of ISR, an IRET instruction is used to return execution to the interrupted program & restore the IF & TF to the state they were in before the interrupt. IRET Pops the value of IP & CS for returning to main program.
* The 8086 type 0 response is automatic & cannot be disabled in any way

Type 1 Interrupt – Single Step Interrupt – In Single step 8086 execute one instruction & stop. We can then examine the contents of register & m/m locations. If they are correct, then we tell the system to go on & execute the next instruction. (trap flag)
If the 8086 TF is set, 8086 automatically do a type 1 interrupt after each instruction executes. When the 8086 does a type 1 interrupt, it pushes the Flag reg. / CS, & IP on the stack, reset the TF & IF. It then gets the CS value for the start of type 1 interrupt service procedure from address 00006H & it get IP value from 00004H.
8086 has no instruction to directly set or reset the trap flag.
To set the TF

```
PUSHF          ; push flag on Stack
MOV BP, SP     ; Copy SP to BP for use as index
OR Word PTR [BP+0], 0100H ; set the TF bit
POPF           ; Restore flag reg.
```

Type 2 Interrupt - Nonmaikable Interrupt

8086 will automatically do a type 2 interrupt response when it receives a low to high transition on its NMI I/P pin. The name nonmaikable given to this I/P pin on the 8086 means that the type 2 interrupt response cannot be disabled by any program instruction. Because this I/P cannot be intentionally or accidentally disabled, we use it to signal the 8086 that some condition in an external system must be taken care of.

Use of Type 2: — To save the program data in case of a system power failure. Some external ckts detects when the ac power to the system fails & sends an interrupt signal to the NMI I/P. Because of the large filter capacitor in most power supplies, the dc system power will remain for perhaps 50 ms after the ac power is gone. This is more than enough time for a type 2 ISR to copy program data to some RAM which has a battery backup power supply. When the ac power returns, program data can be retrieved from the battery-backed RAM & the program can resume execution where it left off.

Type 3 interrupt - Breakpoint Interrupt - is produced by execution of the INT 3 instruction.

Use of Type 3 → to implement a breakpoint function in a system. The system executes the instruction upto the breakpoint & then goes to the breakpoint procedure. 8086 actually do it by temporarily replacing the instruction byte at that address with an OPCODE, the 8086 code for the INT 3 instruction.

A breakpoint interrupt procedure usually saves all the reg. contents on the stack. Depending on the system, it may then send the reg. contents to the CRT display & wait for the next command from the user or simply it may just return control to the user.

Type 4 - Overflow Interrupt - There are two ways to detect & respond to an overflow error
(1) Put the Jump if overflow (JO) immediately after the arithmetic instruction. At this address, we can put an error routine which responds to the overflow in the way we want.
(2) Put the Interrupt on overflow instruction (INTO) immediately after the arithmetic instruction in the program. If OF is set, indicating an overflow error, the 8086 will do a type 4 interrupt after it executes the INTO instruction.
8086 get the CS value for the start of the interrupt ISR from address 00012H & IP value from 00010H.

Software Interrupt - Type 0 through 255
The 8086 INT instruction can be used to cause the 8086 to do any one of the 256 possible interrupt types. The desired interrupt type is specified as part of the instruction.
For eg. INT 32 will cause the 8086 do to type 32 interrupt response.
eg- INT 0, instruction to send execution to a divide by zero ISR without having to run the actual division program.

* INTR Interrupt - Type 0 through 255 (Hardware Interrupt)
8086 INTR I/P allows some external signal to interrupt execution of a program. Unlike the NMI I/P, INTR can be masked (disabled) so that it cannot cause an interrupt.
If the IF is cleared, then INTR I/P is disabled.
When 8086 is reset, IF is automatically cleared.
To Set IF → STI instruction
To reset IF → CLI " .
* IF flag is automatically cleared as part of the response of an 8086 to an interrupt. It prevents a signal on the INTR I/P from interrupting a higher priority ISR in progress. But if we need, we can reenable INTR I/P with an STI instruction at any time.
2nd reason for automatically disabling the INTR I/P at start of an INTR ISR is to make sure that a signal on the INTR I/P does not cause the 8086 to interrupt itself continuously.