Microprocessor (μp) :- An integrated circuit (IC) that take a micro second to execute an instruction.

The first processor was 4004

The 2nd processor was 8008.

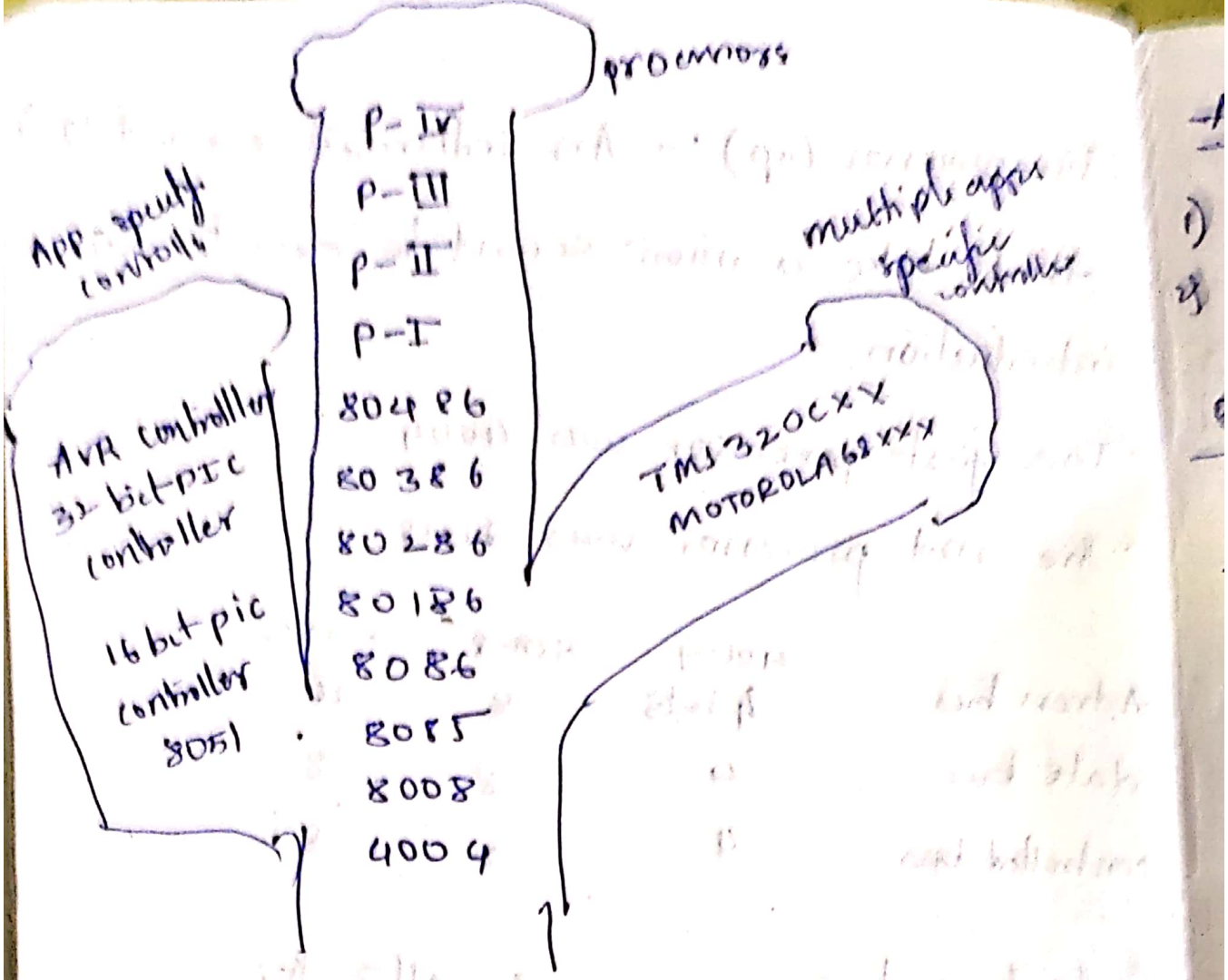
	4004	8008	8085
Adress bus	# bytes	8	16
date bus	4	8	8
controlled bus	4	8	8.

Evaluation of micro processor :- After the development of 8085, the development of micro-processors have been developed to streams as:

i) Application specific controllers

ii) processors

iii) Multiple application specific controllers.



8086 Architecture

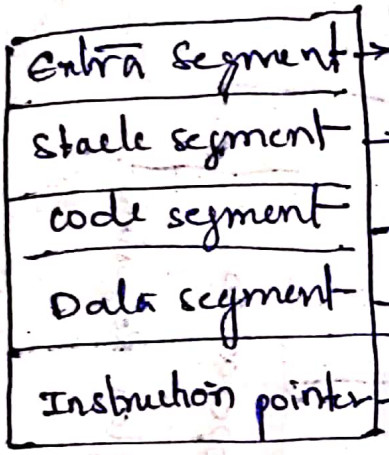
- 20 bit micro-processor
- Adress line 20
- Data line - 16
- It has 4 segment registers with 64 kB space
- 16 bit instruction pointer & data pointer.
- It has 6 byte instruction queue
- It support 1MB of memory storage
- has 16 bit flag register (among them 9 are defined, 2 are undefined).

Architecture of 8086

1) Bus interface unit
2) Execution unit

Bus interface unit

Segment Registers



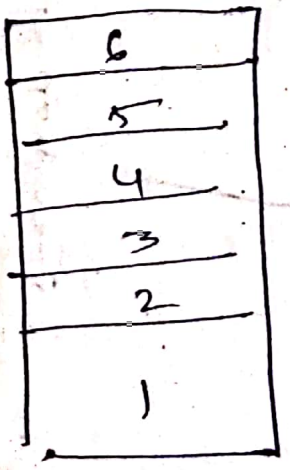
convert 16 bit address line to 20 bits Address line

→ supports base point & stack pointer

→ basic end & sub routines

→ string, alphabets, numeral, etc

→ next instruction which is to be executed. it points.



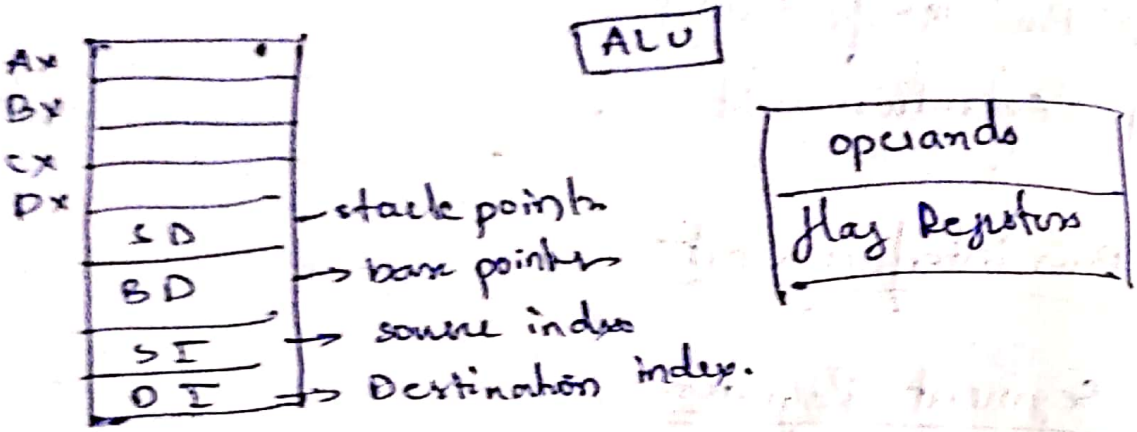
6 byte info. queue

line 1 fetch | decode | execute |

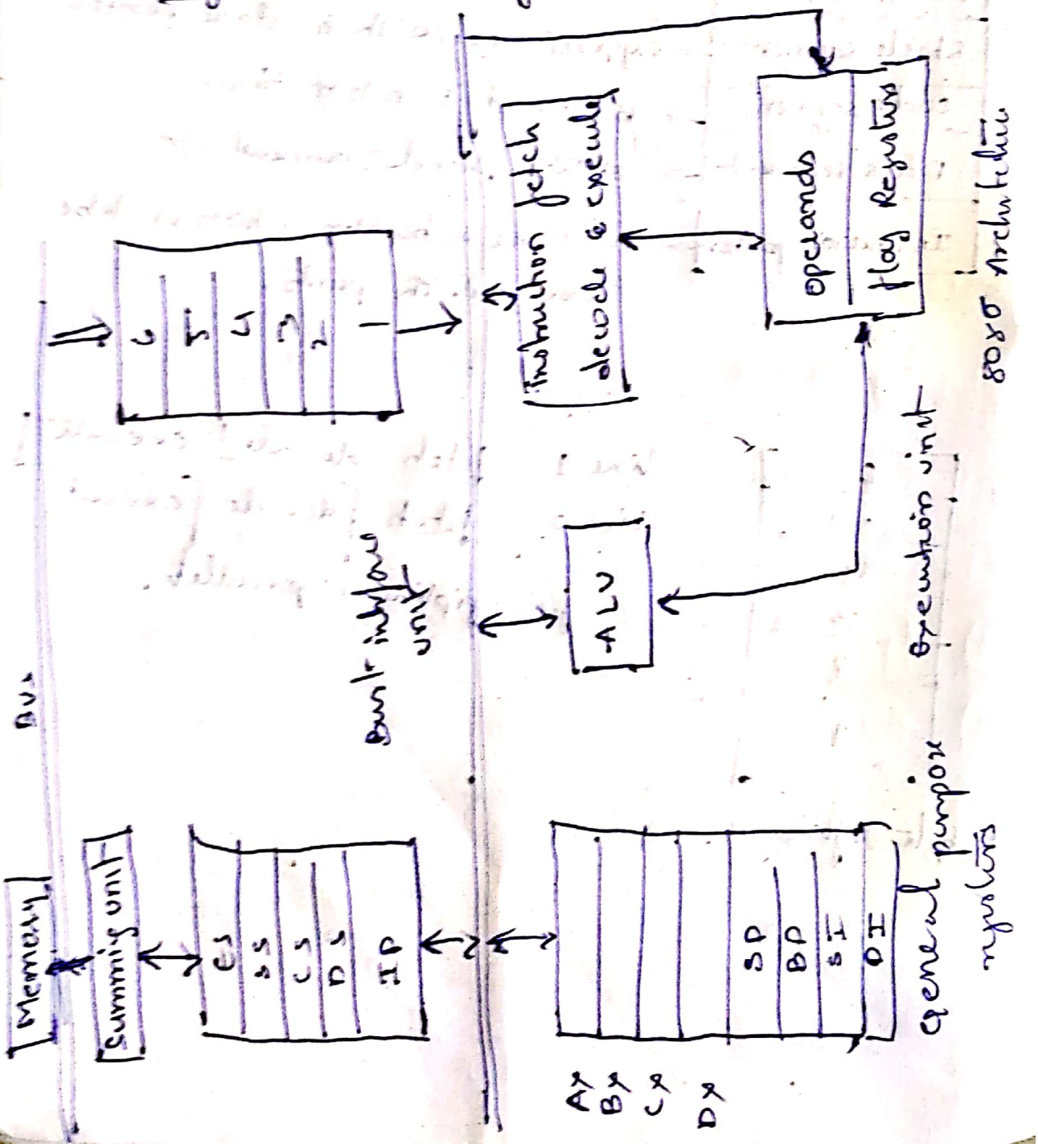
line 2 fetch | decode | execut

fig:- pipline or parallel.

Execution unit :- Group of 8 registers



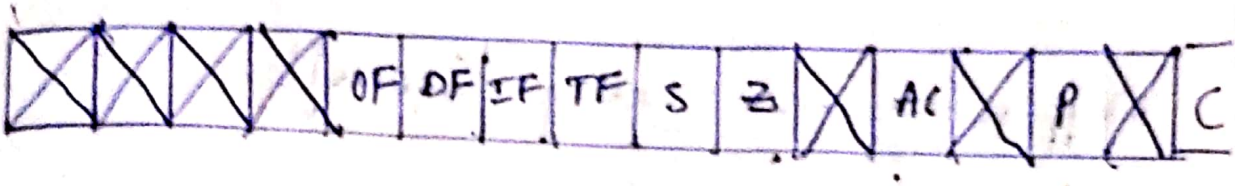
General purpose Registers: 16 bits of data.



Flag Register : status of result.

It is going to store in accumulator (Ax)

Flag Register is 16 bit register in which 9 are defined & 7 are undefined.



Overflow flag (OF)

enable | set | 1 → overflow generate

Disable | Reset | 0 → not generated.

sign flag :-

1 | set → result negative

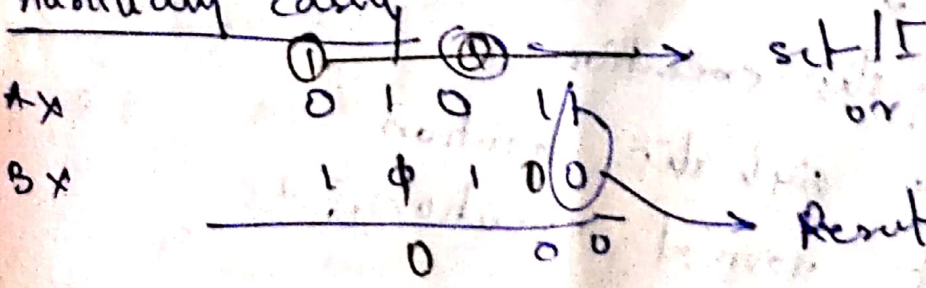
0 | Reset → Result +ve

Zero flag :-

Ax = 0 zero flag will be set | 1

Ax ≠ 0 zero flag will be Reset | 0

Auxiliary carry



Parity

Even parity → 1/set.
Odd parity → 0/Reset.

Carry

bit enable.

Difference b/w carry & overflow

Carry	Overflow
unsigned (Arithmatic operation)	signed (Arithmatic) operations

Direction flag

auto ++
auto --
} memory location

Eg:- ctrl + z

1/set → auto decrement
0/Reset → auto increment

Trap flag

Eg:- F2

step by step execution
1/set → step by step execution
0/Reset → normal execution.

Interupt flag

1) Maskable
   - If interupt is generated it doesn't stop execution. After the execution it execute the interupt
2) Non maskable. → two causes of interupts
3) software
4) hardware.

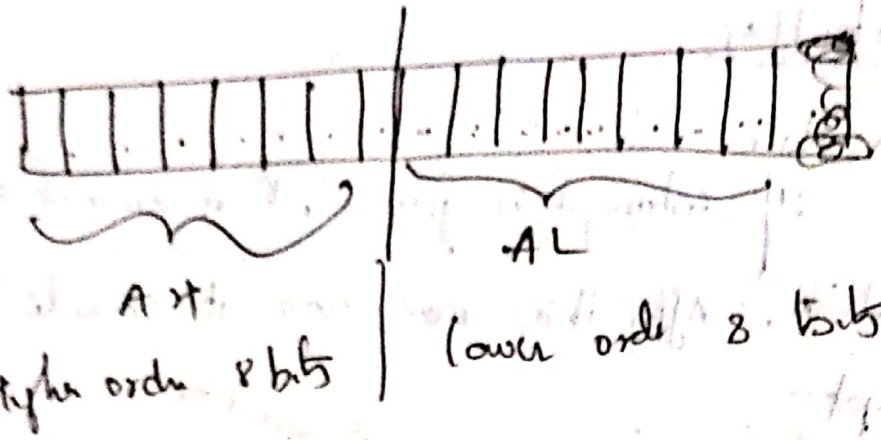
set | enabled | 1 — Generated.
Reset | disabled | 0 not Generated.

*Register Organization: → In 8086 — 14 Register

1) General purpose Registers (5) Register.
2) Segment Registers
3) pointer Registers
4) Index Registers.
5) flag Register.

General purpose Registers

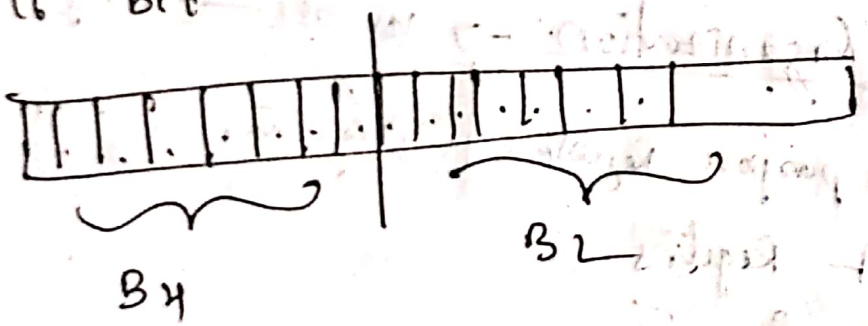
1) AX (Accumulator)
   - 16 bits / 2 bytes
   - 2 groups



* Results are stored in AC.
* status of AC is stored in flag Register.

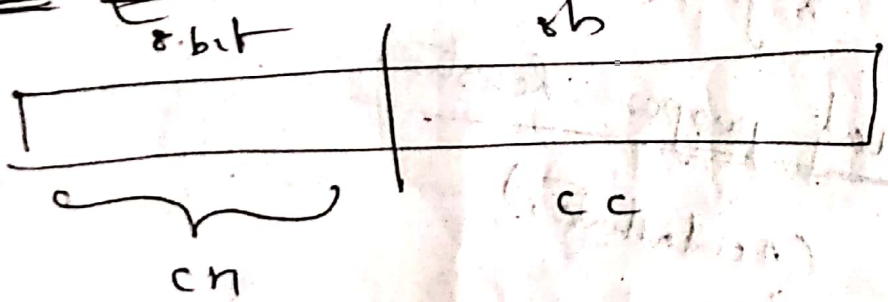
* Bx (Base register)

16 bit



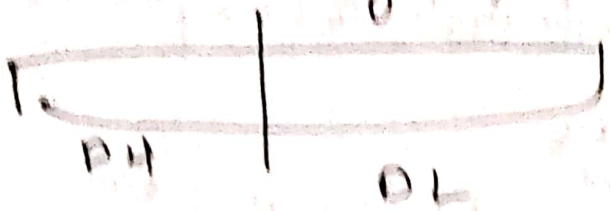
* Arithmatic operation.

* CX (Count register)



performs the counting operations
Eg:- counting takes place in loop.

* DX (Data Register)



* Data transfer or Address Transfer.

Segment Register

- ES - extra segment
- SS → stack segment
- DS → Data segment
- CS → code segment.

} 64 kilobytes

Extra segment

* physical Address of 2564H → 7640H
* offset Address 125H + 1254H
* effective Address → 2678H

Stack segment

pop LES
Load effective segment

To change the data.

* 64 kilobytes

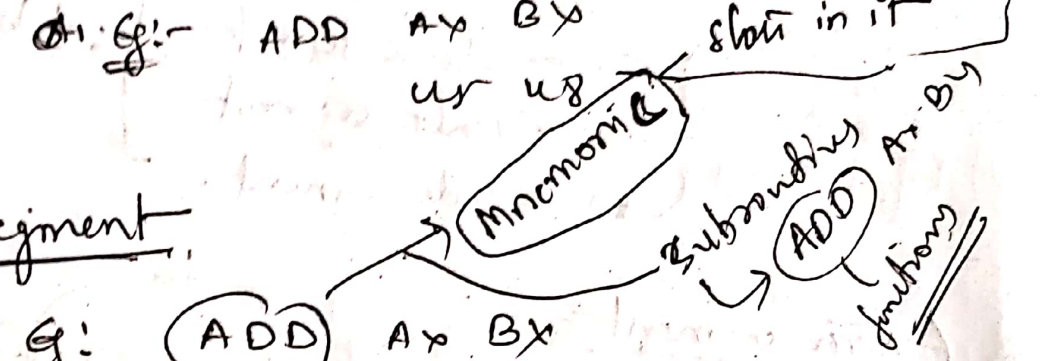
Stack Segment

* provide support stack & base pointer.
* 64kbyte space
* Data segment
* can be changed by pop instruction

Data segment

* 64kbyte of memory
* going to deal data which you want to execute.

Eg:- ADD AX BX



Code segment

Eg: ADD AX BX

This type of instruction is taken care by the code segment.

* 64kbyte space

Pointer Registers:

IP → instruction pointer (next instruction)
BP → Base pointer [starting of execution process]
SP → stack pointer [current execution process instruction]

This are controlled by stack segments.

* Index registers

SI → source index
DI → Destination index
} — by using this index we can store in general purpose Registers

* stores & retrieves the data

Memory Organization

∞ BHE, A0 ⟶ Active LSB
↳ Bus High Enable

odd memory locations.

④ Pin diagram of 8086

1) Common signal description → connect to other process (multiprocessing operation)
2) maximum mode signal Description
3) minimum mode signal Description
↳ not using connect other devices
↳ alone process going to work.
4)

* mathematical operations are not perform like ×, ÷

* 8089 — can perform ÷.

8086 ⟷ 8089 — combine can perform mathematical operations

Common signal description

1) AD_0 to AD_{15} →
4) A_{16}/S_3, A_{17}/S_4, A_{18}/S_5, A_{19}/S_6 — stack signal status
3) $\overline{BHE}/S_7$
4) NMI → Non mas. Interrup
5) INTR — Interrup
6) clk — clock
7) RESET Reset
8) READY
9) $\overline{TEST}$ complement
10) $\overline{RD}$
11) MN / $\overline{MX}$
12) Vcc (5V)
13) Vss (Ground / Gnd)

* AD_0 to AD_{15}
  - Address line
  - Data line

* Just 1 instruction → 4 clock cycles

1 — clock → Address line enable

2 — clock → Data line Activated.

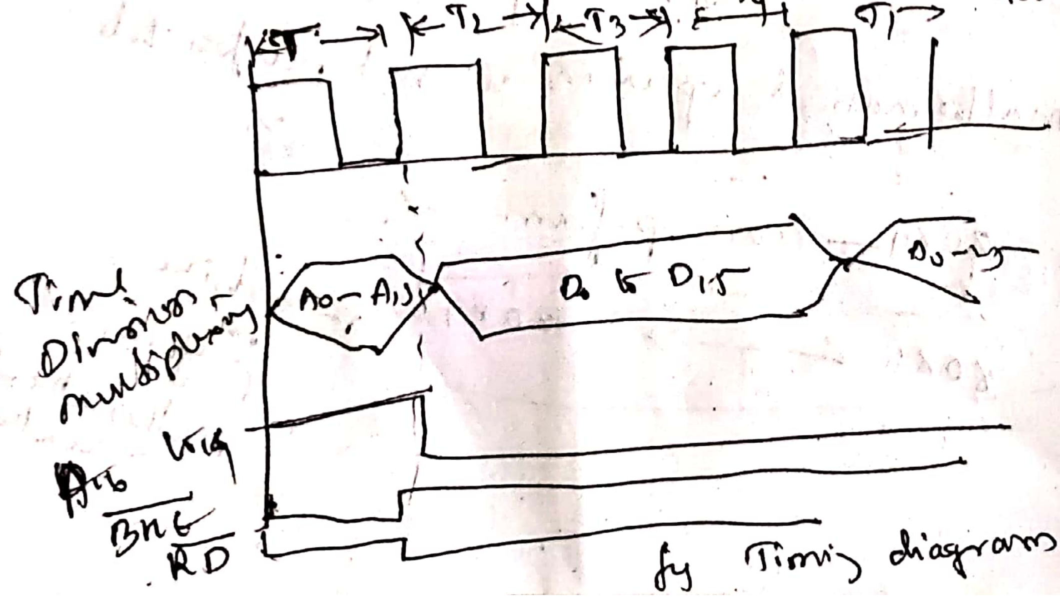


fig Timing diagram.

7) A
S_4	S_3	function
0	0	Extra seg in use
0	1	stack " "
1	0	code segm in use
1	1	Data segment in use.

) $\overline{BHE}$ Bus high Enable.

~~Adrs bus by~~ → indicates whenever high → high or else 0

Adrs bus — high
Data bus — low } But complement opposite.

NMI → non maskable interrupt → 0 / high
→ softw
INTR → hardware → 0 / high

clk → crystal → external connecting clock

RESET → Reset or back to first position.

READY → duration of read or write → high - ready / low

$\overline{TEST}$ → inspect wait → wait instruction waits

$\overline{RD}$ → Read → Adrs high — 0
Data by = high.
key cycle reading of reading.

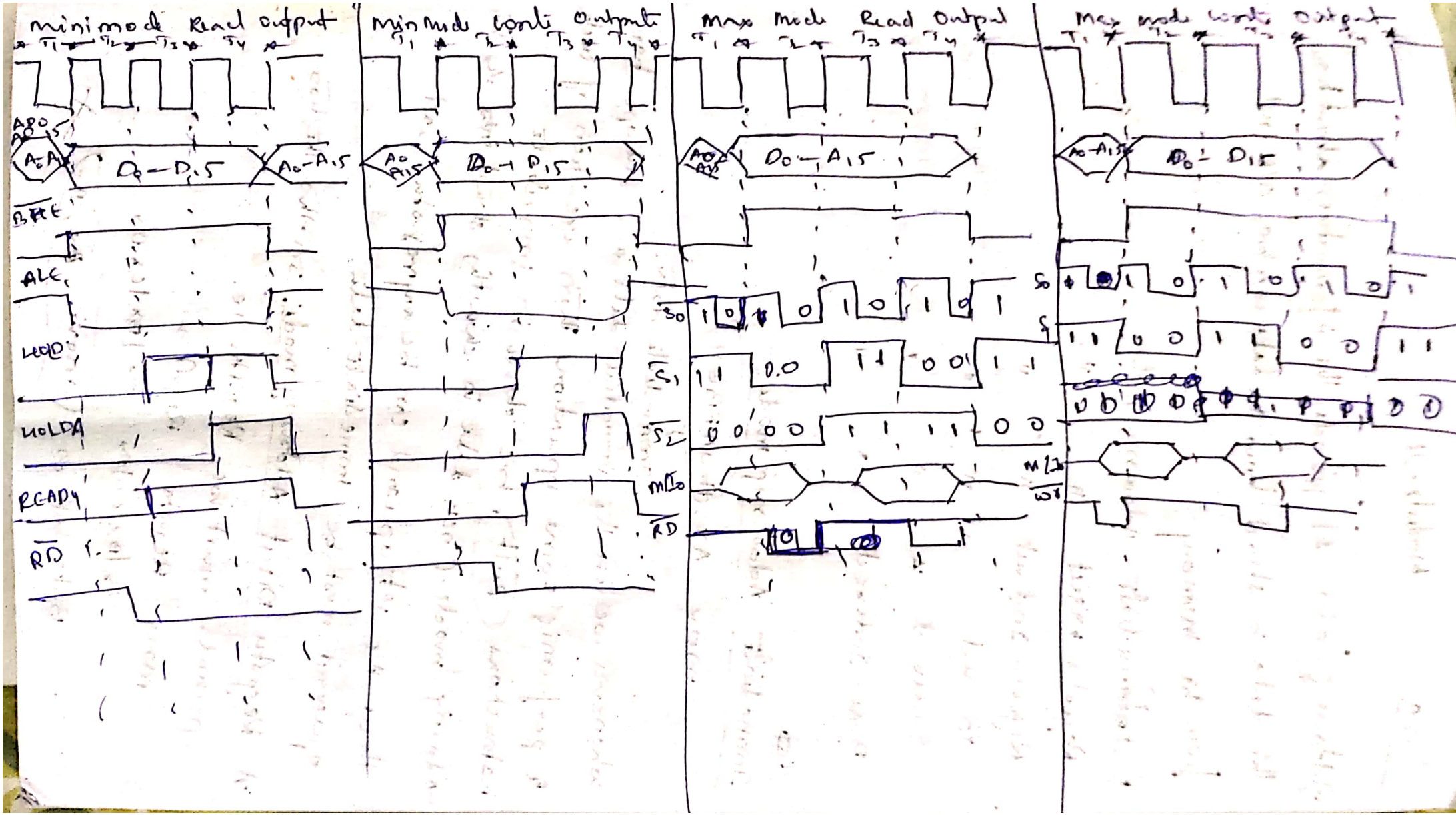
MN / $\overline{MX}$ minimum mode

V_{CC} provide supply

- 5 MHz
- 8
- 10 } clock signal

S_2	S_1	S_0	
0	0	0	
0	1	0	} mem read, mem write
0	1	1	
1	0	0	
1	0	1	} I/O read, I/O write
1	1	0	
1	1	1	

Timing diagram:-



6/1/20 Module - II

Addressing modes :- It provides the information about how the data is stored how the data can be accessed from the memory

1) Immediate Addressing modes.
   Register Direct " "
   Register Indirect " "
   Based " "
   Indexed " "
   Based - Indexed " "

Immediate Addressing mode :-

mov AX, #0AFBEH
(#0AFBEH → Immediate data.)

whenever we use the immediate data as a 2nd operand & performing some operation then execution of such instructions (fast) be done in immediate Addressing mode (here data is directly stored into Register Here the 1st operand can be any register and 2nd operand must be immediate data.

2) Register Direct Addressing mode. In this the two operands must be Register i.e.

mov Register, Register

AX/BX/CX/DX AX/BX/SI/DI

But not as (mov AX, 2)

such type of instructions will be executed in Register direct Addressing mode (Register to Register)

Ex: Add Ax, Bx ; Sub Ax, B; AND Ax, Bx

In the above example the result will be stored again in 1st operand (ie Ax in above example)

(3) Register Indirect Addressing mode: Here 1st operand is a register & 2nd opnd can be memory location or register memory location.

4) Base Addressing mode :- If the instructions (insetting) Bx. (or) BP then such instructions would execute in Base Addressing mode

Ex:- mov Bx, Ax ; mov BP, SI

F) Indexed Addressing mode: If we are using SI, DI in the instructions then the execution of such instructions would execute in Indexed Addressing mode.

Ex mov SI, ABCDH.

1) Base - Indexed Addressing mode: combination of both Base & Indexed Addressing mode. Instruction like

mov AX, (BX+SI) ; mov AX, [BP+SI] will execute in Base - Indexed Addressing mode.

Here the final result is addition of address location in BX & SI stored in AX.

2) Instruction set: It involves How many instructions are present how they are indexed in execution how they are going to execute

It is divided into multiple groups They are

(1) Data Transfer Instruction.
(2) Arithmetic Instructions.
(3) logical instructions.
4) Program execution control Information
5) String manipulation instructions.
6) Bit manupulation Instructions.
7) Jump Instruction.
8) Data Transfer Instructions: It involves.

1) mov: It is used with 2 operands.

Eg:- mov operands, operands.
↓ ↓
Registers Register /
memory location memory location / Data.

→ we can have several combinations like

1) mov AX, BX (mov register, register)
2) mov AX, [SI] (mov register, memory location)
3) mov SI, 4567H (mov register, data)
4) mov (SI), AX (mov memory loc, Register)
↳ content of SI & content of AX
5) mov (SI), [DI] (mov mem loc, mem loc)
mem. loc.
6) mov [DI], 4567H (mov mem. loc, data)

(2) Swap :- It is an instruction which is going to shift the contents of one register to another & vice versa
→ Here 2 operands are registers only

Eg:- SWAP AX, BX

(3) XCHG.

(2) Arithmetic instruction :-

1) Addition : 8086 can perform 5 diff addition operations.
They are 1) ADD :- 2) ADC :- Add with carry
3) AAA → ASCII adjust after addition :- whenever may be result it will be activated with ASCII values

Eg:- AAA AX, BX

4) DAA → Decimal Adjust after Addition:- The result of the no in AL16 will be hexadecimal. This DAA after addition will give the result in decimal format (converts hexadecimal to decimal).

5) INC:- INC AX
It will the value in AX will be added with 1 byte

② Subtraction:- 8086 perform 5 diff subtraction operation. They are
1) SUB
2) SBB
3) AAS
4) DAS
5) DEC

→ To perform unsigned multiplication

③ multiplication

⇒ MUL, IMUL, AAM, DAM.

IMUL → To perform signed multiplication
AAM → ASCII adjust After multiplication.
DAM :- Decimal adjust after multiplication.

4 Division:- DIV, IDIV, AAD, DAD
similar to above multiplications mnemonics

3) Logical Instructions:-

1) AND OP1, OP2
2) OR OP1, OP2
3) NOT, OP1
4) XOR, OP1, OP2

OP1 → Register/mem loc
OP2 → Data/Register/mem loc

4) program execution and control instructions.

CALL, LOOP, are the ones which controls the program execution.

1) Loop:- multiple the execution of statement

Ex: BX ← 0
LOOPS ADD AX, BX
RET loop stmt should end with RET

if we use CALL LOOPS the loop will be executed.

2) CALL:- It is a instruction which is used to call the set of instructions to the main program & get executed.

Syntax CALL LOOPS

3) JMP instruction

JNBE/JA → Jump if above

JNB/JAE jump if above or equal

JNAE/JB → jump if below

JNA/JBE Jump if below or equal

JNLE/JG → Jump if greater than.

JNL/JGE → jump if greater than or equal

JNGE/JL → jump if less than.

JNG/JLE → jump if less than or equal

→ Based of flag bits we will select the set of jump instructions.

→ second group based on the flag bits like carry, parity, sign, zero etc.

a) JC → jump if carry

b) JNC → jump if not carry

c) JP → jump if parity.

d) JS → jump if signed.

e) JNS → jump if not signed

f) JZ → jump if zero.

g) JNZ → jump if not zero.

8/1/20 Logical and Rotation instructions :-

1) AND
2) OR
3) XOR
4) NOT

5) ROR rotate right
6) ROL rotate left
7) SHL shift left
8) SHR shift Right

1Q perform AND or and XOR on

Ax = FEBAH   Bx = 7A9BH.

Ax FEBA	1111	1110	1011	1010
Bx = 7A9B	0111	1010	1001	1011
AND Ax	0111	1010	1001	1010
	7	A	9	A

Result 7A9A

OR	1111	1110	1011	1011
	15	14	11	11
	F	E	B	B
XOR	1000	0100	0010	0001
	8	4	2	1

(4) NOT Ax if Ax = 3333H

Ax = 0011 0011 0011 0011

NOT Ax = 1100 | 1100 | 1100 | 1100

12   12   12   12

Result CCCC

(5) ROR operand | operand | Reg (or) memory loc / data.

eg: ROR 45H ⇒ 01000101

remove

result → 22H.

(7) SHL } both have two arguments with it
SHR }

1st argument is operand (like AX, BX, etc)

2nd argument represents a no. i.e. by how many no. of places the bits must be shifted

Eg:- SHL, AX, 2 ; SHR BX, 5 ; SHL AX, 1

* flag instructions —

1) CLC → 'clear' carry flag.
2) CLD → clear Directional flag.
3) CLI → clear Interrupt flag.
4) CMC → complement carry flag
5) STC → set carry flag.
6) STD → set Directional flag.
7) STI → set-interrupt flag.

* miscellaneous instructions —

1) LES → load Extra segment } → used to transfer
2) LDS → load Data segment } the address
3) LEA → load Effective Address.

Assembler Directives: Directives are used to define the data types.

Data Byte DB - 8 bits
Data word DW - 16 bits
Data Quad DQ - 32 bits
Data DT - 80 bits

ENDS :- To terminate any one of the code or, data segments ENDS is used.

START: It starts / originates a particular code. To terminate this functionality, we use END START

MACRO :- In a can collect a group of lines to be executed repeatedly we usually use 'loop' directive (we mention loop before) and these instructions in the loop can be used only in the segment in which it is declared so, if we want to use it in another segments also then we need to declare it globally & this can be done by using MACRO directive

* MACRO → has faster execution
* MACRO - use more memory.

PROC :- It is just an alternative for MACRO The only diff is that, with MACRO, each time it is called, the code that is called is pasted in that each time hence MACRO uses more

~~More~~ memory when as ~~RE~~ PROC does not copy the code each time
It uses only original copy of the code.
* ~~RE~~ PROC - has slower execution.
RROC utilizes less memory.

MACRO

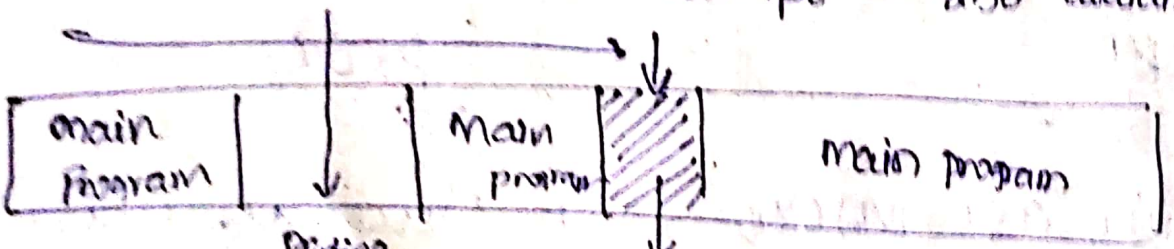
1) Directives are MACRO and ENDM.
2) More memory is required
3) No need of transfer control
4) Speed of execution is high (or) fast computation
5) Easy to pass the parameters
6) No instruction is defined to call the MACRO
7) No requirement of stack.

~~AAA~~ PROC

1) Directives are PROC and ENDP
2) Less memory is required
3) control is required to transfer to place where procedure is present
4) Slow computation
5) Difficult to pass the parameters.
6) CALL instruction is used to include procedure
7) The instruction are stored in stack

Interrupts:

* Two types
  - H/w interrupt
    - Maskable
    - Non-maskable (NMI, INTR)
  - S/w interrupt — 256 available



S/w interrupt:

256 :- 5 System defined interrupt
251 : User defined interrupt

INT 0
:
INT 255

* S/w interrupts ~~Categorised~~ Categorized into Program Interrupt, System Interrupt, User-defined Interrupts.

Program Interrupt :-

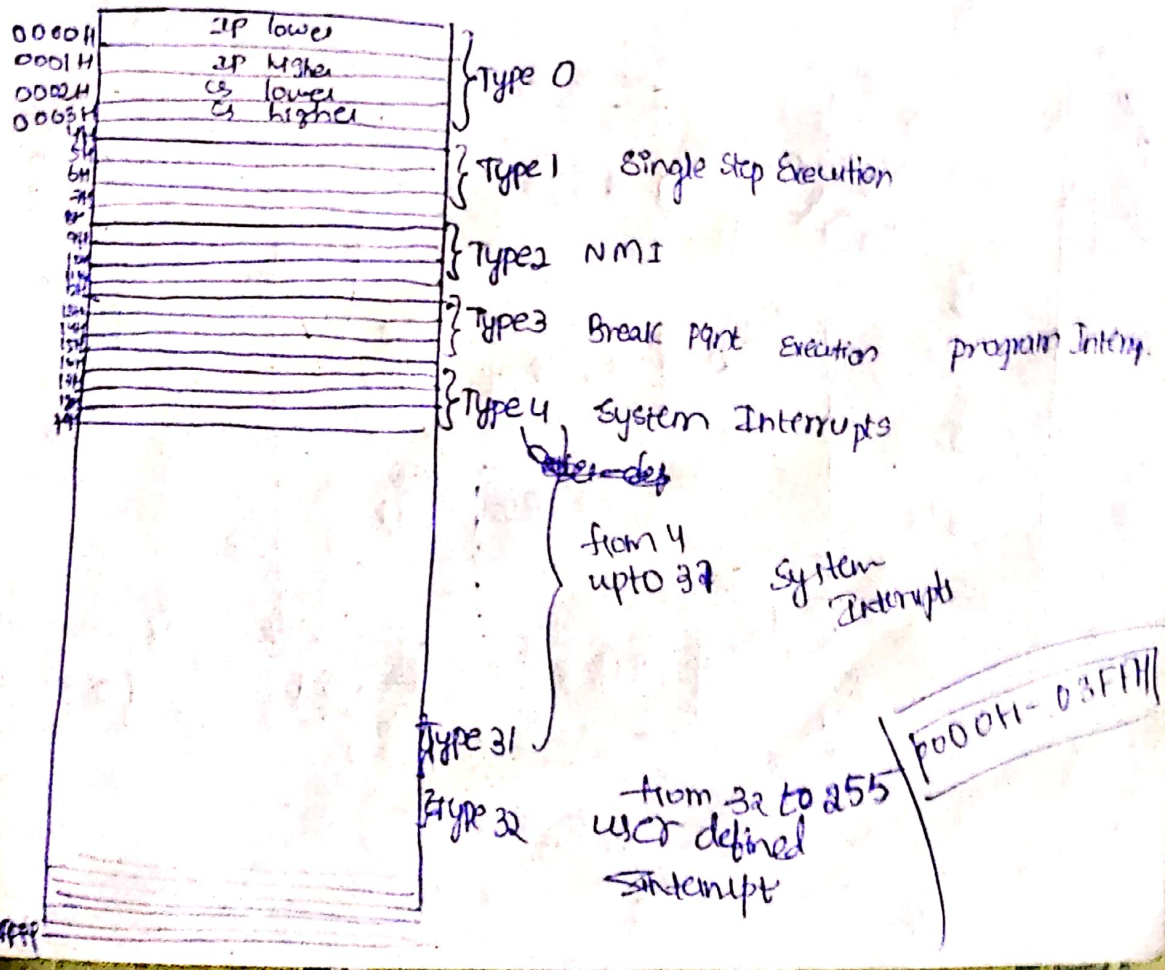
→ while executing a program for eg division operation any thing divides by zero.

ISR - Interrupt sub Routine [when ~~zero~~ divided by zero]

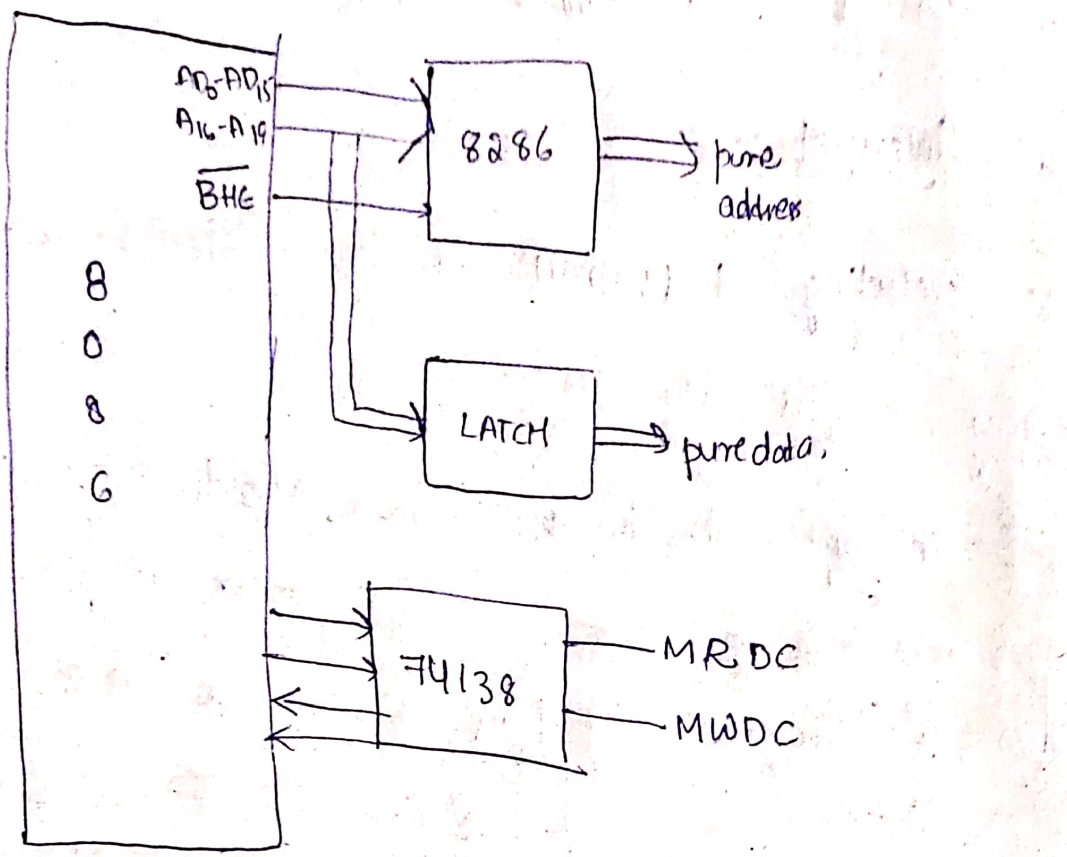
- It comes out and prompt error message.
- It requires 4 bytes of storage space
  - 2 for IP. (Instruction pointer)
  - 2 for Code Segment (CS)

255 × 4 = 1020 bytes assigned for s/w interrupt.

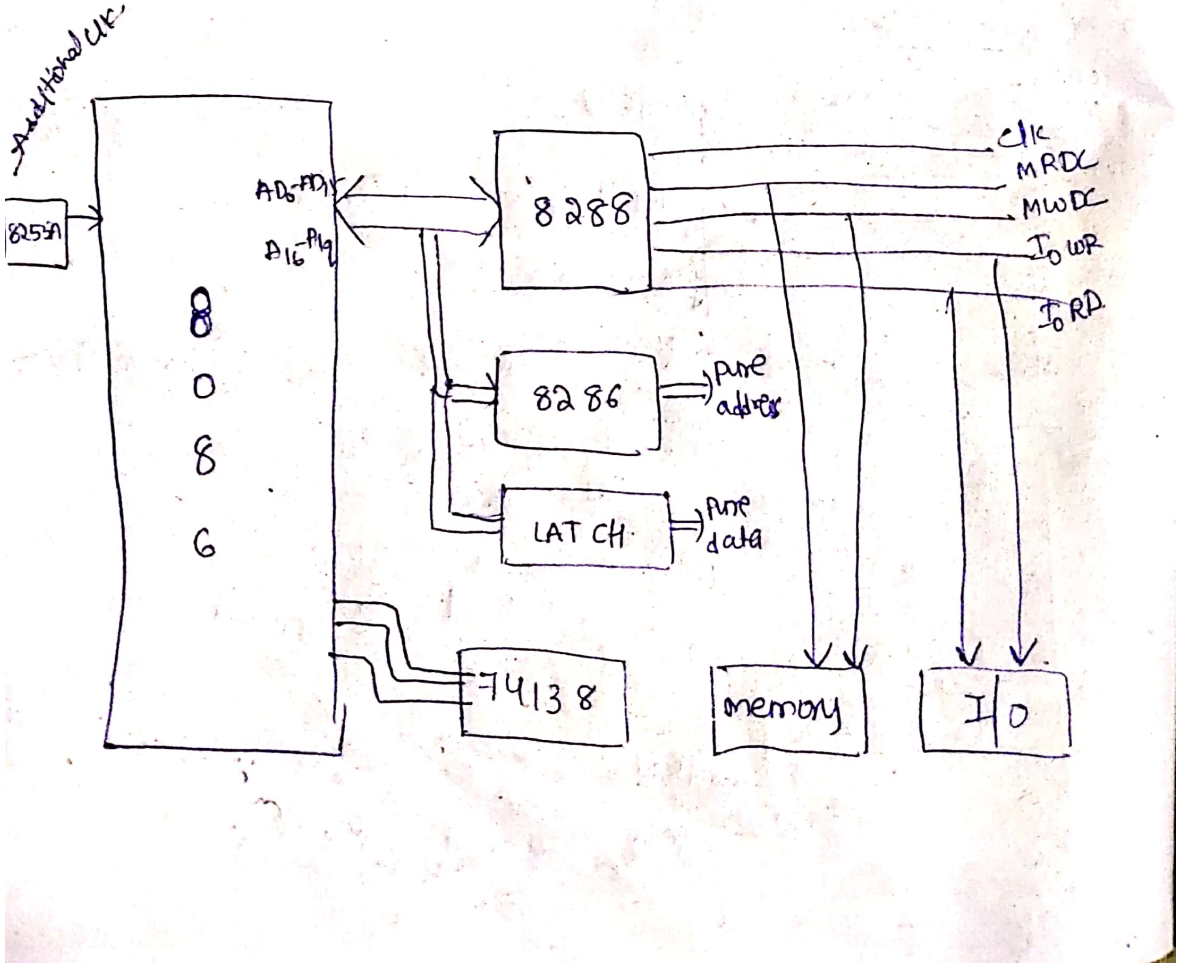
*



Minimum mode of 8086:-

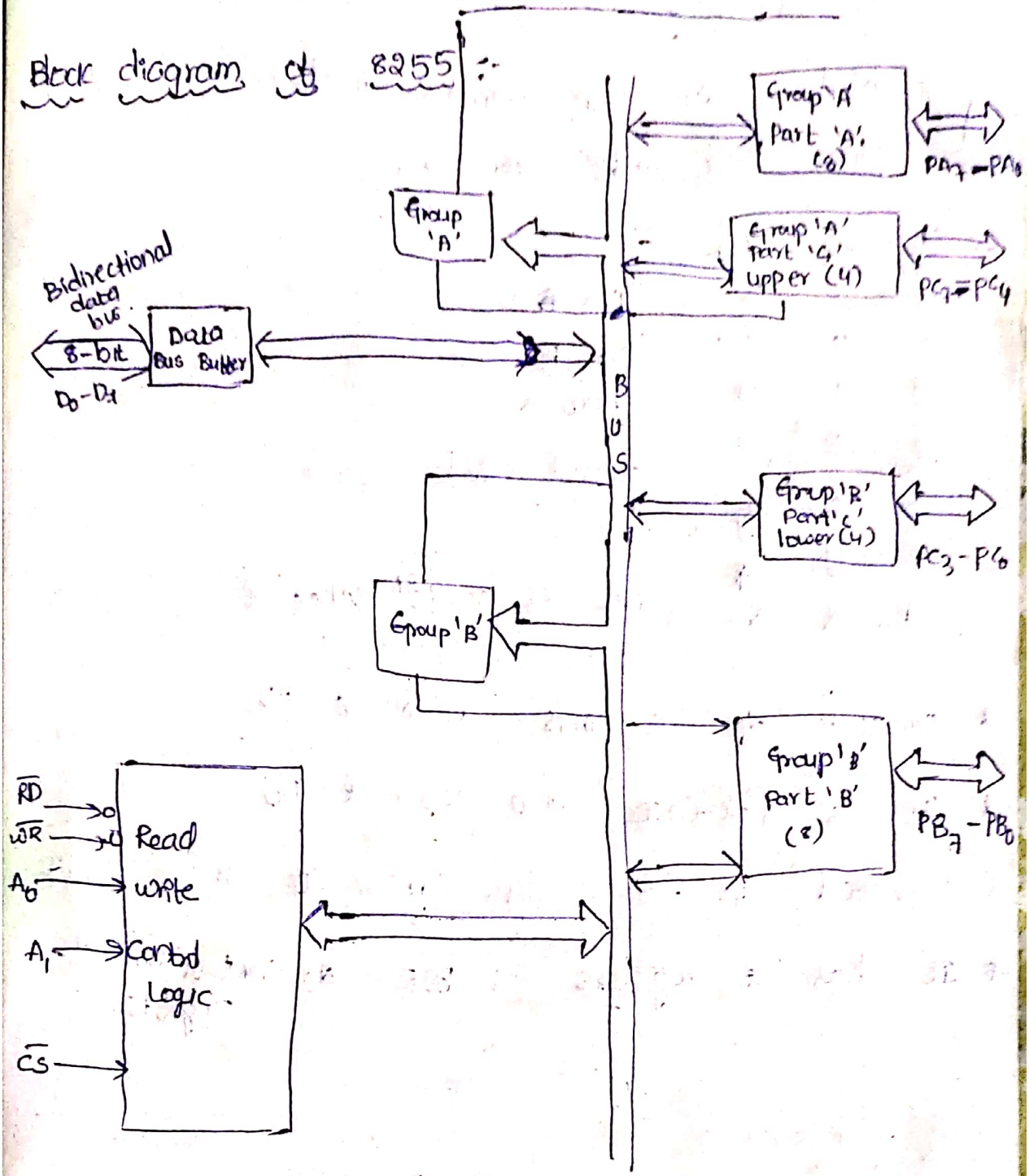


Maximum mode of 8086:-



module-III

Block diagram of 8255 :-



consisting 8 blocks.

1) Data Bus Buffer – It deals with 8 bit; pins $D_0 - D_7$

2) Read/write control logic :-

$\overline{RD}$ → low → high → read operation.

$\overline{WR}$ → low → high → write operation.

A_0, A_1 — Address lines from 8086

$\overline{CS}$ — high (works in minimum mode)
— low (max mode working)

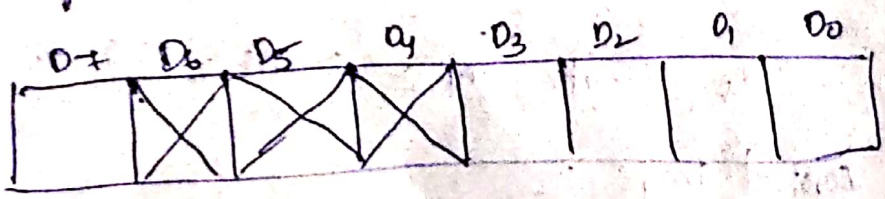
$\overline{CS}$	A_1	A_0	
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	control word register
1	X	X	No 8255 is selected

* Consists of 3 ports with 8 pin.
* It is interface btw 8086 & I/O.
* operated in 3 modes. mode 0, mode 1, mode 2
* It has 2 registers 1. BSR 2. control word register.

BSR register:-

Bit set/Reset Mode Register.

* Length of BSR register is 8 bits.

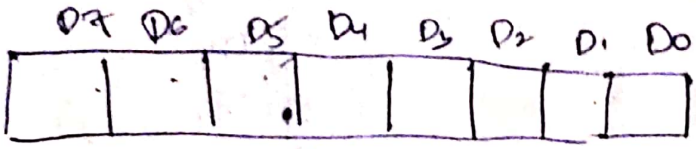


* $D_7 \Rightarrow 0$ → BSR register.
  $D_7 \Rightarrow 1$ → Control word register

* D_5, D_4, D_6 are not defined.

D_3	D_2	D_1	
0	0	0	PC_0 pin will be set
0	0	1	PC_1 pin will be set
0	1	0	PC_2
0	1	1	PC_3
1	0	0	PC_4
1	0	1	PC_5
1	1	0	PC_6
1	1	1	PC_7

control word Register :-



D_3 → set '1' port 'C' lower '4' pins i/p mode
→ Reset '0' port 'C' lower 4 pin o/p mode

D_0 → set '1' port 'C' upper '4' pin i/p mode
→ Reset '0' o/p mode

D_4 → Reset port 'A' o/p mode
→ set port 'A' i/p mode

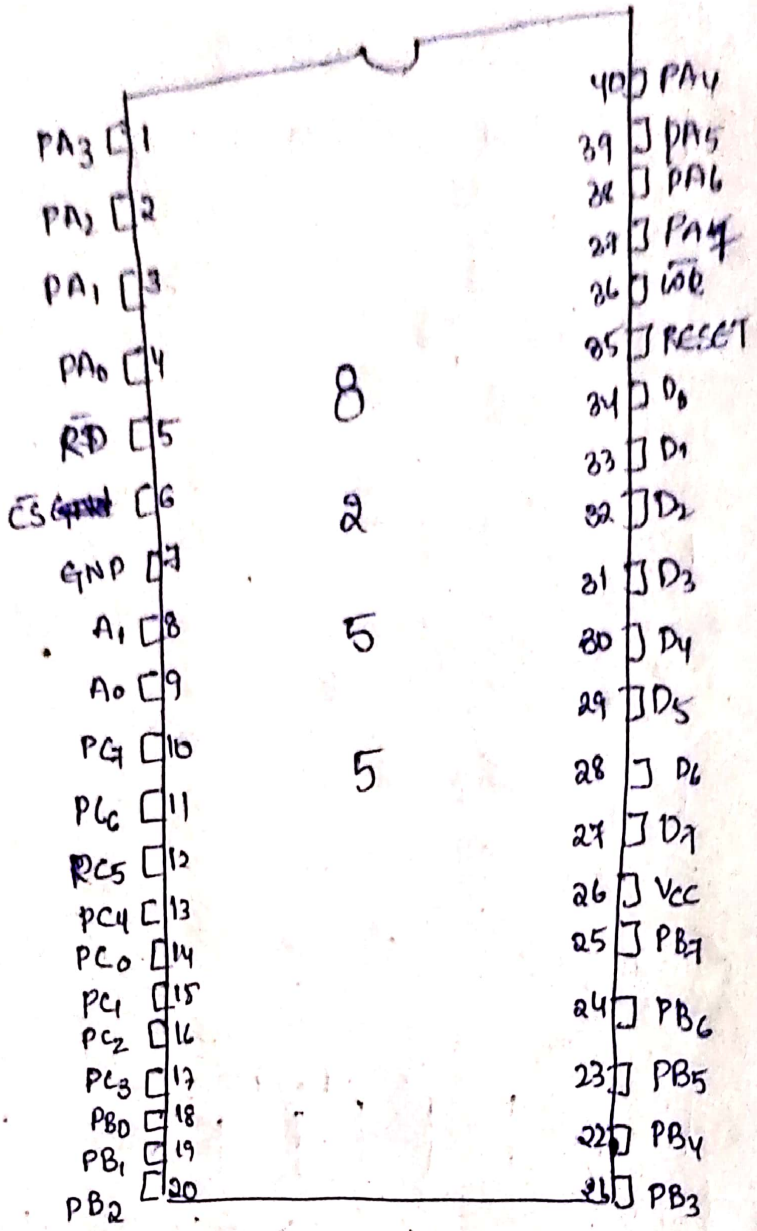
D_1 → Set '1' port 'B' i/p mode
→ Reset '0' port 'B' o/p mode

D_2 → Reset '0' port 'B' o/p mode
→ set '1' mode 1 port 'B'

D_7 → set port 'C' i/p mode
→ Reset port 'C' o/p mode

D_6	D_5	Port A functionality
0	0	mode 0
0	1	mode 1
1	X	mode 2

Pin diagram of 8255:



* It consists of 40 pins

for port A, B, C – 24 pins

data pins – 8

Remaing 8 :- 1 – Vcc Ground.

2 – $\overline{CS}$:- port selection.

* 8255 operated in 3 modes.

mode 0 :- Normal operation (ports work like individual block)

mode 1 :- ~~strobe~~ i/p (whenever data bus buffer is full then it strobe generate signal strobe

mode 2 * PORTC high order connect with PORT A. remaining 4 pins for hand shaking signals. 8 → data.

* If we want to connect with PORT B ~~big~~ lower order used.

* 24 ┌ 12 pins (PORT A + 4 pins ~~from~~ of port 'C' (upper))
└ 12 pins (PORT B + 4 pins of port 'C' (lower))

* connect only 2 devices.

→ $\overline{STB}$, $\overline{IBF}$, $\overline{OBF}$, $\overline{INTR}$ — handshaking signals connecting with 4 pins

Mode 2 :- ~~Step~~ bidirectional mode

* 5 hanking signals

* 13 pins ~~PORT A~~ ┌ 8 pins for data transfer (PORT A)
└ 5 pins for handshaking.

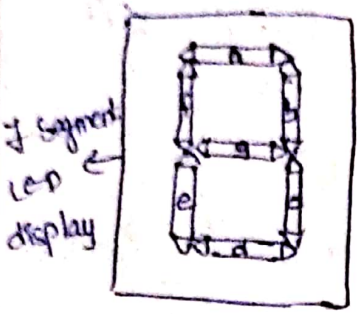
$\overline{STB}$, $\overline{IBF}$, $\overline{OBF}$, $\overline{INTR}$, $\overline{INTA}$ → These are send along with 8 pins

* can connect only devices.

LED:

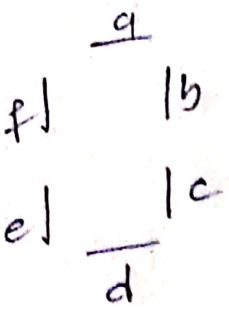
Common Cathode LED

- It is called 7 segment LED display
- It has 7 segments

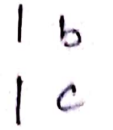


a	b	c	d	e	f	g		
1	1	1	1	1	1	0	— 0	— 0000
0	1	1	0	0	0	0	— 1	— 0001
1	1	0	1	1	0	1	— 2	— 0010
1	1	1	1	0	0	1	— 3	— 0011
0	1	1	0	0	1	1	— 4	— 0100
1	0	1	1	0	1	1	— 5	— 0101
1	0	1	1	1	1	1	— 6	— 0110
1	1	1	0	0	0	0	7	— 0111
1	1	1	1	1	1	1	8	— 1000
1	1	1	1	0	1	1	9	— 1001

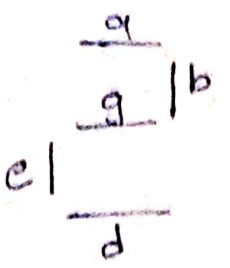
0 segment



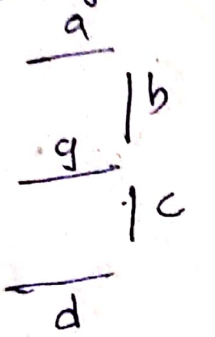
1 segment



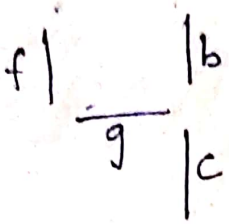
2 segment



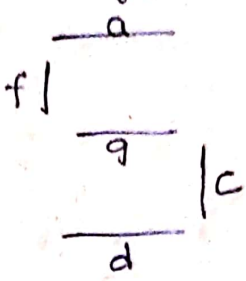
3 segment



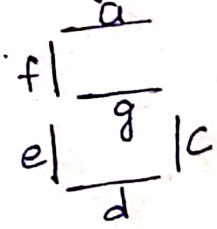
4 segment



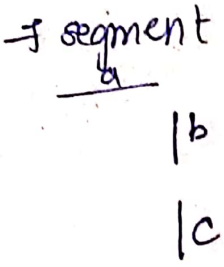
5 segment



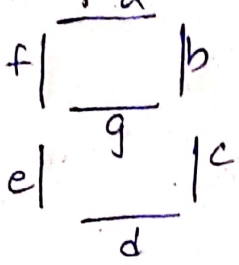
6 segment



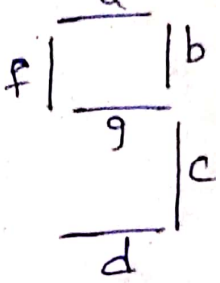
7 segment



8 segment



9 segment



LED interfacing with 8086: