Taught By:
Hemant
Pokhariya Sir

Prepared By:
Ayank Gupta

MICROPROCESSORS ⇒ [μP]

⇒ It is a CPU on a single chip.

(Central Processing Unit) ← CPU → (brain of computers)

Contains ⇒ (1) registers as internal memory.

(2) Control unit (CU)

↳ controls all peripherals

(3) Arithmatic & Logical Unit

* Microprocessor is a multipurpose programmable clock-driven semiconductor device.

⇒ It takes i/p from i/p device in binary form and processes it according to given instruction and again produces o/p in binary form.

* Machine language ⇒ Binary language (0/1)

* Assembly language ⇒ Mnemonic language.

* High level language ⇒ ⊙ Programmer friendly language.
⊙ Easy to understand.

* Low level language ⇒ ⊙ machine friendly
⊙ Tough to understand.

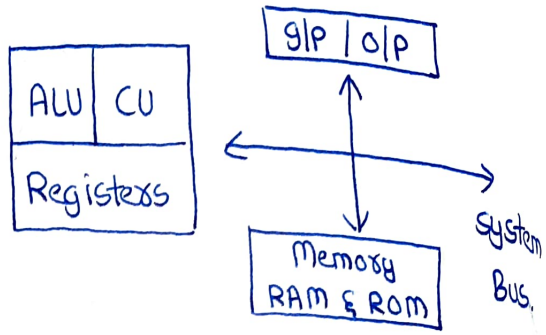
* VLSI chip is a massive integration.

=) It contains more than 10000 registers.

Microcomputer => It is a digital computer whose CPU is a microprocessor.

* It can be represented with 4 components =>

(1) microprocessor
(2) memory
(3) input
(4) output



=) these components will form a system with help of System Bus.

* Arithmatic Logic Unit => In this area, various arithmatic operations (+ - * /) and logical operations (AND, OR, EX-OR) are performed.

* Register Array => It is used to store data temporarily during execution of a program and are accessible to users through instructions.

⊙ This area is identified by letters such as]

B, C, D, E, H and L.

* Control Unit ⇒ It provides necessary timing and control signals to all operations in µp.

⊙ controls data flow b/w µp and memory & peripheral.

* Memory ⇒ ⊙ Primary memory ↴

(Code memory) ∈ (1) ROM ⇒ Read only memory.

(Data memory) ∈ (2) RAM ⇒ Random access memory

* Input / Output ⇒ used to communicate with outside world.

⊙ devices are known as peripherals.

* System Bus ⇒ It is a communication path b/w µp and peripherals.

⊙ It is a group of conducting wires used to carry information (bits).

⊙ 3 types of bus ⇒ (1) Address Bus ⇒ (Unidirectional)

⇒ It carries address only, from one point to other point.

⇒ In 8085 ⇒ length is 16 bit.

(2) Data Bus ↴

⇒ carries data (bidirectional)

⇒ In 8085 ⇒ 8 bits

(3) Control Bus ↴ ⇒ used to generate timing / control signals

eg. ⇒ memory read / write.

i/o read / write.

⊙ **Analogy** ⇒ Suppose a letter is to be delivered to H.No. 55, and the concerned person is called by ringing the bell of his house.

* House No. 55 ⇒ ADDRESS BUS
* letter ⇒ DATA BUS
* ringing the bell ⇒ CONTROL BUS

** All registers are designed to work in 8 bits as length of data bus is 8-bits.

Differences b/w Microprocessor & Microcontroller ⇒

(1) Microprocessor contains only CPU, whereas, controller contains a CPU, memory & i/o all integrated in one chip.

(2) ⊙ μprocessor ⇒ Heart of Computer system.
⊙ μcontroller ⇒ Heart of Embedded system.

(3) ⊙ μprocessor ⇒ used in personal computers.
⊙ μcontroller ⇒ used in embedded computers.

(4) ⊙ μprocessor ⇒ uses external buses to interface
⊙ μcontroller ⇒ uses an internal controlling bus.

(5) ⊙ μprocessor ⇒ based on Von Neumann model.
⊙ μcontroller ⇒ based on Harvard architecture.

* μprocessor (practically) cannot be used as single chip.

Programming Model of 8085 =>

=> Information needed to write an assembly language programme is called programming model.

Eg. => 2020 : MVI A 25H => A = 25

(address) (instruction to move 25 to A)

* It contains registers, Address and Data bus.
* 8085 has 8 bit bidirectional data bus and 16 bit unidirectional address bus.
* Another function of Address bus => Determine Memory Capacity.
* 8085 has 64 kb memory capacity.
*

$2^{16} \Rightarrow 2^6 \times 2^{10}$

64 kb

Formula of Memory capacity

=> 2 to power of length of address bus (AB).

Registers of 8085 =>

(1) General Purpose Registers => Stores data temporarily.

* 6 registers to store 8 bit (DB) data.
* B, C, D, E, H, L
* To perform 16 bit operation => can be combined as register pairs.

=> BC, DE, HL

(2) Memory Registers => Holds memory addresses [16 bit size] (length of AB)

* It is of 2 types ↴

(a) Program Counter => Holds the address of next instruction to be fetched.

* It is incremented by n if size of instruction is n bytes.

Eg. =>
2000 : MVI A 25H
2002 : MVI B 30H
} incremented by 2 as inst. size is 2 bytes.

(b) Stack Pointer => used as a memory pointer.

* points to read/write memory called stack.

* always incremented / dec. by 2 during push / pop.

(3) Specific Purpose Registers =>

(a) Accumulator (ACC) => 8 bit register => Part of ALU.

* After performing arithmatical / logical operators, result is stored in accumulator.

* For 8085, ALU operation destination operand must be ACC.

(b) Flag register (FR) => size: 8 bit

* It becomes set or reset depending upon condition of a result after any ALU operation.

⊙ 5 main flags =>

When, value = 1 = set
value = 0 = reset

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
S	Z	-	AC	-	P	-	CY

(A) Sign flag => After any operation, if MSB B(7) of the result is => 1 => no. is negative and sign flag is set.

(SF) => 0 => no. is positive and sign flag becomes reset.

- ⊙ from 00H to 7F, sign flag is 0.
- ⊙ from 80H to FF, sign flag is 1.

(B) Zero flag => After any operation, if result is 0 (00(H)), zero flag becomes set (1), otherwise reset (0). (ZF)

- ⊙ at 00H => zero flag is 1 (result is 0)
- ⊙ from 01H to FFH => zero flag is 0 (result is non zero).

* useful in loops when writing in Assembly code.

(C) Auxiliary Carry flag => used in BCD no. system (0-9).

* Only flag NOT ACCESSIBLE to programmer. (AC)

- ⊙ If after any operation => B(3) generates a carry to B(4) => flag becomes set (1), otherwise reset (0).

(D) Parity flag => If after any operation, result has even parity (even no. of 1s), flag becomes set (1), otherwise 0. (PF)

(E) Carry flag => Carry is generated when performing n bit operation and result is more than n bits => then flag is set (1) otherwise reset (0) (CY)

(imp)**

⊙ During subtraction ⇒ if A > B ⇒ reset
if A < B ⇒ set

* Also known as BORROW FLAG.

* Example ⇒ 2000 : MVI A 25H A = 25
2002 : MVI B 3DH B = 3D
2004 : ADD B A = A+B

Donot write ADD A,B (Invalid)

* Find value of all FR ⇒ 25 = 00100101
3D = 00111101
A+B = 01100010

(1) AC = 1 (as carry to B(4))

(2) ZF = 0 (not 0 result)

(3) CY = 0 (as 8bit operation = 8 bit result)

(4) PF = 0 (odd parity)

(5) SF = 0 (MSB is 0)

* Shortcut ⇒ 25 + 3D ⇒ 62 H

(0 MSB of 6 ⇒ SF=0)

(sum of 5+D > F(16) ⇒ AC=1)

(62 ≠ 0 ⇒ ZF=0)

(result not more than ff ⇒ CY=0)

(3 1's ⇒ PF=0)

Types of Instructions in 8085 =>

* An instruction is a binary pattern designed inside a microprocessor to perform a specific function.
* Entire group of instructions => Instruction Set.
* 8085 has 246 instructions.
* Each instruction is represented by 8-bit binary value.
* 8 bits of binary value => Op-code / Instruction Byte.

  - ⊙ Mnemonic => MOV A,B
  - ⊙ OP-code => MOV ⊙ Operand => A,B
  - ⊙ Hex-code => 78H (0111 1000)

* Notations => ⊙ R => 8085 8-bit register (A, B, C, D, E, H, L)
  - ⊙ M => memory register
  - ⊙ Rs => Source register
  - ⊙ Rd => register destination
  - ⊙ Rp => register pair (BC, DE, HL, SP)
  - ⊙ () => contents of

* M is the memory location whose address is specified by the contents of HL Rp. If H = 20H and L = 50H, then

  M = 2050H.

(A) DATA TRANSFER INSTRUCTIONS =>

* These move data b/w registers, or b/w registers and memory.
  - ⊙ both cannot be memory.
* These copy data from source to destination.
* Source data is not modified.

(1) Opcode => MOV

Operands => Rd, Rs
M, Rs
Rd, M

} * Copy from source to destination.

* size of both operands must be same.
* If one of the operands is a memory location, its location is specified by the contents of HL registers.

⊙ Eg. => MOV B, C or MOV B, M

B = C B = ← M

(2) Opcode => Rd, data
Operands => M, data

} * Move immediate 8-bit

* 8-bit data is stored in destination register or memory.

⊙ Eg. => MVI B, 57H or MVI M, 57H

B = 57 M ← 57

(3) Opcode ⇒ LDA

Operands ⇒ 16-bit address

* Load Data Accumulator

* can only use default register as accumulator.

* contents of memory location, specified by 16 bit address in the operand, are copied to accumulator.

⊙ Eg. ⇒ LDA 2034H

A ← 2034

⊙ if content at 2034 is 50, then 54 is given to A.

(4) Opcode ⇒ LDAX

Operands ⇒ B/D Register pair

* Load accumulator indirect.

* contents of designated register pair point to a memory location.

* this instruction copies the contents of that memory location into accumulator.

* contents of either register pair or memory are not altered.

⊙ Eg. ⇒ LDAX B } always specify high order register.

if B = 20, C = 50 and contents of 2050 is 25, then

A = 25

(5) Opcode ⇒ LXI

Operands ⇒ Reg. pair, 16 bit Data

* Load register pair immediate

* loads 16 bit data in register pair.

⊙ Eg. ⇒ LXI H, 2034H } H = 20 & L = 34 ⇒ M = 2034

⊙ Eg. ⇒ LXI B 3050 ≈ MVI B 30, MVI C 50 } B = 30, C = 50

(6) Opcode ⇒ LHLD

Operands ⇒ 16 bit address } * Load H-L registers direct

* copies contents of memory location pointed out by 16-bit address into register L.

* copies contents of next memory location into register H.

⊙ Eg. ⇒ LHLD 2040H

L ← 2040

H ← 2041

(7) Opcode ⇒ STA

Operands ⇒ 16 bit address } * Store accumulator direct
* reverse of LDA.

* contents of accumulator are copied into memory location specified by operand.

⊙ Eg ⇒ STA 2500H

2500 ← A

(8) Opcode ⇒ STAX

Operands ⇒ Reg. Pair } * Store accum. indirect
* reverse of LDAX

* contents of acc. are copied into memory location specified by contents of reg. pair.

⊙ Eg. ⇒ STAX B { MVI B 30, MVI C 50 } ⇒ 3050 ← A

(9) Opcode ⇒ SHLD
Operands ⇒ 16 bit address

* store H-L reg. direct
* reverse of LHLD

* contents of L are stored into memory location specified by 16-bit address.
* contents of H are stored into next memory location

⊙ Eg. ⇒ SHLD 2550 H

2550 ← L

2551 ← H

(10) Opcode ⇒ XCHG
Operands ⇒ None

* Exchange H-L with D-E

* contents of register H exchanged with contents of D.
* contents of L ex. with E.

⊙ Eg. ⇒ XCHG

(11) SPHL
Operands ⇒ None

* Copy H-L register to Stack Pointer (SP)

* loads contents of HL pair into SP.

⊙ Eg. ⇒ SPHL

↓

LXI H 2050

SPHL ↓

SP = 2050

(12) XTHL

Operands => None

* Exchange HL with top of stack

* contents of L register are exchanged with location pointed out by contents of SP.

* contents of H reg. exchanged with next location (SP+1)

⊙ Eg. XTHL

(13) Opcode => PCHL

Operands => None

* load program counter with contents of H-L register.

* contents of H and L are copied into program counter.

* contents of H are placed as high order byte and contents of L are placed as low order byte.

⊙ Eg. => PCHL =>

```
LXI H 2050
PCHL
PC = 2050
```

(14) Opcode => PUSH

Operands => Reg. pair

* push register pair onto stack

* contents of reg. pair copied into stack.

* SP is decremented and contents of high order register (B, D, H, A) are copied into stack.

* SP is again decremented and contents of low-order registers (C, E, L, flags) copied into stack.

(15) Opcode => POP

Operands => Reg. pair

* pop stack to reg. pair

* exactly reverse of PUSH.

(16) Opcode => OUT

Operands => 8 bit port address

* copy data from acc. to port with 8 bit address.

* contents of acc. are copied into I/O port.

⊙ Eg. => OUT 78H (address, not data as not OUT I)

78H ← A => (content of A will go to address 78)

(17) Opcode => IN

Operands => 8 bit

* copy data from port to accumulator.

⊙ Eg. => IN 8C H

* reverse of OUT

A ← 8C H

Assembly Language Programme to exchange contents of two memory locations 2050(02) & 3050(03).

⊙ Concept => T = A, A = B, B = T

C ← 2050, 2050 ← 3050, 3050 ← C

```
LDA 2050
MOV C, A
LDA 3050
STA 2050
MOV A, C
STA 3050
```

* using direct memory

(Disadvantage => if we need to transfer 100 memory locations, we have to write 100 times)

* using indirect memory =>

```
LXI H, 2050
MOV C, M
LXI H, 3050
MOV A, M
LXI H, 2050
MOV M, A
LXI H, 3050
MOV M, C
HLT
```

* uses dynamic memory allocation.

(B) ARITHMATIC INSTRUCTIONS =>

* Addition, Subtraction
* unfortunately, no multiplication / division
  => still by Successive addition / subt. => it can be done.

⊙ Key point => Destination operand must be Accumulator.

* Source operand can be 8 bit no. / immediate data or it can be a register or memory.

Addition ↓

* Accumulators stores the result after arithmatic operation.
* cannot add two other 8 bit registers directly.

Eg. => if we want to add B and C => transfer contents of either B or C into A => then add A or B / A or C.

Subtraction => * Any 8 bit no. / contents of register / contents of memory location can be subtracted from contents of accumulator.

* Result is stored in Accumulator.
* Subt. is performed in 2's compliment form.
  => If result is -ve => it is stored in 2's comp. form.
* no two other 8 bit registers can be subtracted directly.

Increment / Decrement => These can be inc./dec. by 1].

(1) 8 bit contents of a register or memory location.

(2) 16 bit contents of register pair.

* This can be performed on any register or memory pair.

(1) Opcode => ADD
Operands => R, M

} * adds register or memory to accumulator.

* contents of register or memory or register are added to
* if operand is memory location, its address is specified by H-L pair.
* All flags are reflected as result of addition.

⊙ Eg. => ADD B or ADD M

↓ ↓

A = A+B A = A+Mc

(2) Opcode ⇒ ADC
Operands ⇒ R, M
} * Add registers or memory to acc. with carry.

* contents of registers | memory and Carry flag (CY) are added to contents of accumulator.

⊙ Eg. ⇒ ADC B or ADC M

A = A+B+(CY)pr A = A+Mc+(CY)pr

(3) Opcode ⇒ ADI
Operands ⇒ 8-bit data
} * Add immidiate to accumulator.

* 8-bit data added to contents of acc.

* Eg. ⇒ ADI 45H

A = A+45

(4) Opcode ⇒ ACI
Operands ⇒ 8 bit data
} * Add imm. to acc. with carry.

⊙ Eg. ⇒ ACI 45 H

A = A+45+(CY)pr

(5) Opcode ⇒ DAD
Operands ⇒ Reg. pair
} * Add reg. pair to H-L pair

* 16 bit contents of reg. pair are added to contents of HL pair.

* result stored in HL pair.

* if result is larger than 16 bits, then CY is set.

* no other flags are changed.

⊙ Eg. => DAD B

↳ H = H+B

L = L+C

(6) SUB

Operands => R, M

* Subtract register or memory from accumulator

* same as ADD

* All flags are modified to reflect result of subtraction.

⊙ Eg. => SUB B or SUB M

↓ ↓

A = A-B A = A-M

(7) Opcode => SBB

Operands => R, M

* subt. register / memory from acc. with borrow

* same as ADC

⊙ Eg. => SBB B or SBB M

↓ ↓

A = A-B-(cy)pr A = A-M-(cy)pr

(8) Opcode => SUI

Operands => 8 bit data

* subt. immediate from accumulator

* same as ADI

⊙ Eg. => SUI 45H => A = A-45

(9) Opcode ⇒ SBI
Operands ⇒ 8 bit data
} * Subt. immediate from acc. with borrow

* same as ACI
* Eg. ⇒ SBI 45 H ⇒ A = A - 45 - (cy) or

(10) Opcode ⇒ INR
Operands ⇒ R, M
} * increment register / memory by 1

* result is stored in same place.
⊙ Eg. ⇒ INR B or INR M

B = B + 1 , M = Mc + 1

(11) Opcode ⇒ INX
Operands ⇒ Reg. pair
} * inc. reg. pair by 1

* Eg. ⇒ INX H
HL = HL + 1
} used to point next memory location.

(12) Opcode ⇒ DCR
Operands ⇒ R, M
} * dec. register / memory by 1

* Eg. ⇒ DCR B or DCR M

B = B - 1 M = Mc - 1

(13) Opcode ⇒ DCX
Operands ⇒ Reg. pair
} * dec. reg. pair by 1

⊙ Eg. ⇒ DCX H ⇒ used to point previous memory location
HL = HL - 1

ALP to Add two 8-bit numbers =>

⊙ Input data ↴

4F	D6
2051H	2050H

⊙ Output data ↴

01	25
3051H	3050H

=> LDA 2050 - A = D6

MOV B, A - B = D6

LDA 2051 - A = 4F

ADD B - A = A+B (CY ≠ 01)

STA 3050 - 3050 = 25, CY = 01

MVI A 00 - A = 00

ADC A - A = A + A + (CY) PS => A = 01

STA 3051 - 3051 = 01

HLT

* 8085 Instruction set can be classified according to size as =>

(1) 1-byte instruction => opcode

* opcode & operands are represented in 1 byte.

* each instruction req. only one memory location.

⊙ Eg. => MOV A, B

Hex Code => 78 H (0111 1000)

MOV A B

0 1 1 1 1 0 0 0

Code	Registers
000	B
001	C
010	D
011	E
100	H
101	L
111	A

⊙ Eg. ⇒ ADD B

Hex Code ⇒ 80H

(10000000)
↓
ADD

* If operands are registers or register + implied addressing mode ⇒ 1 byte instruction.

(2) 2-byte Instruction ⇒ | opcode | 8bit data/add |

* first 8 bits ⇒ opcode

next 8 bits ⇒ operand → can be 8 bit data or 8 bit address

* requires 2 memory locations.

⊙ Eg. ⇒ MVI A 32

⇒ ADI 45

⇒ IN F2

(3) 3-byte instruction ⇒ | opcode | 8bit data/add | high bit data/add |

* first 8 bits ⇒ opcode
* next 16 bits ⇒ 16 bit data / address (operand)
* requires 3 memory locations.

⊙ Eg. ⇒ LDA 3050

JMP 3050

LXI B 3040

Addressing Modes in 8085 =>

The way of specifying data to be operated by an instruction is called addressing mode.

* Types of Addressing Modes =>

(1) Immediate Addressing Mode => (I in opcode)

* source operand is always data.

=> if data is of 8 bits => 2 bytes

16 bits => 3 bytes

⊙ Eg => MVI B (65)
ANI (56) — (Data)
ADI (45) —

(2) Register Addressing Mode =>

* data to be operated is available in registers and register(s) are operands.

=> operation is performed within registers.

⊙ Eg. => MOV A, B (both A & B are registers)

ADD C (add contents of reg. A & reg. C and store result in reg. A)

INR B (inc. contents of reg. B)

SUB D

* will require only 1 memory location => 1 byte instruction

(3) Direct Addressing Mode =>

* data to be operated is available inside a memory location and that memory location is directly specified as an operand

⊙ Eg. => LDA 2050
LHLD 3050
IN 35
STA 3050
(memory locations)

(4) Register Indirect Addressing Mode =>

* data to be operated is available inside memory location, but mem. location is indirectly specified by register pair.

⊙ Eg. => MOV A, M (move contents of mem. location specified by HL pair to accumulator)

LDAX B (move contents of mem. location whose address is specified by BC reg. pair to acc.)

STAX D (move contents of A to mem. location whose add. is specified by DE pair)

* All instructions here will be of 1 byte.

(5) Implied / Implicit Addressing Mode =>

* In this operand is hidden and data to be operated is available in instruction itself.

* if address of source of data as well as address of result is fixed => no need to give any operand along with instruction.

⊙ Eg. => CMA (finds and stores 1's comp. of contains of accumulators A in A)

RRC (rotate acc. A right by 1 bit)

DAA

* DAA (Decimal Adjust after Addition) => (used for BCD no. system)

=> after applying this result will be in BCD no. system.

=> used immediately after normal addition instruction operating on BCD codes.

⊙ Step 1 => It checks lower nibble first, if it is greater than 9 and 'AC' flag is set => it adds 6 to it.

⊙ Step 2 => Then it checks upper nibble, if it is greater than 9 and carry flag is set => it adds 60 to it.

* Eg. => 25 + 45 = 6A + 06 = 70

(C) LOGICAL INSTRUCTIONS =>

* These perform operations on data stored in registers, memory and status flags.

*⊙ Destination operator will be always Accumulator.

⊙ Source can be memory / register.

⊙ AND, OR, XOR => Any 8-bit data or contents of register or memory location can have these operations with contents of acc.

⊙ Rotate => Each bit in accumulator can be shifted either left or right to next position.

⊙ Compare => Equality
Greater than
Less than
} Result is reflected in status flags.

⊙ Compliment => Replace 0 by 1 and 1 by 0.

(1) Opcode => ANA
Operands => R, M
} * Logical AND register or memory with accumulator.

* Contents of accumulator are logically ANDed with contents of register or memory. => result in Accumulator.

* If operand is a memory location, its address is specified by contents of H-L pair.

* S, Z, P are modified to reflect the result of operation.

* CY is always reset and &AC is set.

⊙ Eg. => ANA B or ANA M

A = A AND B A = A AND M

(2) Opcode => ANI

Operands => 8 Bit data

* Logical AND immediate with accumulator.

* Same as ANA => Here immediate data is ANDed.

⊙ Eg. => ANI 86 H

A = A AND 86

⊙ Eg. => MVI A 45 => 01000101

ANI 0F => 00001111

A = 05 → 00000101

(3) Opcode => ORA

Operands => R, M

* logical OR register or memory with acc.

* contents of accumulator are logically ORed with cont. of register or memory.

⊙ Eg. => ORA B or ORA M

A = A OR B/M

(4) Opcode => ORI

Operands => 8 bit data

* Logical OR imm. with accumulator.

=> imm. data ORed with contents of acc.

⊙ Eg. => ORI 86H

A = A OR 86

(5) Opcode ⇒ XRA
Operands ⇒ R, M
} * Logical XOR registers or memory with accumulator.

⇒ contents of acc. XORed with contents of registers / memory.

⊙ Eg. ⇒ XRA B or XRA M

A = A XOR B/M

(6) Opcode ⇒ XRI
Operands ⇒ 8bit data
} * XOR imm. with acc.

⇒ ⊙ Eg. ⇒ XRI 86H

A = A XOR 86

* ANA ~ ORA ~ XRA and ANI ~ ORI ~ XRI

(7) Opcode ⇒ CMA
Operands ⇒ None
} 1's compliment accumulator

* contents of acc. are complemented.

* no flags affected

⊙ Eg. ⇒ CMA ⇒ A = A'

Ques. ⇒ Write any 4 instructions used to reset accumulator.

Ans. ⇒ (1) MVI A 00 ⇒ A = 00

(2) SUB A ⇒ A = A - A ⇒ 00

(3) ANI 00 ⇒ A = A AND 00

(4) XRA A ⇒ A = A XOR A ⇒ 00

Ques. ⇒ Write an ALP in 8085 for 10110011

(a) which always switch ON 6th bit

⇒ LDA 2050
ORI 40

```
  10110011
  01000000   OR
  ----------
  11110011   (6th Bit ON)
```

(b) always switch OFF 6th bit

⇒ LDA 2050
ANI BF

(c) toggles 6th bit ⇒ XRI 40 } 1 XOR 1 ⇒ 0

(8) Opcode ⇒ CMP
Operands ⇒ R, M } * Compare registers or memory with accumulator.

* if (A) < reg./mem. ⇒ carry flag is set
* if (A) = reg./mem. ⇒ zero flag is set
* if (A) > reg./mem. ⇒ carry and zero flags are reset.

⊙ Eg. ⇒ CMP B | CMP M } * it performs subtraction but data is not modified.
A-B

**

(9) Opcode ⇒ RLC
Operands ⇒ None } * Rotate accumulator left.

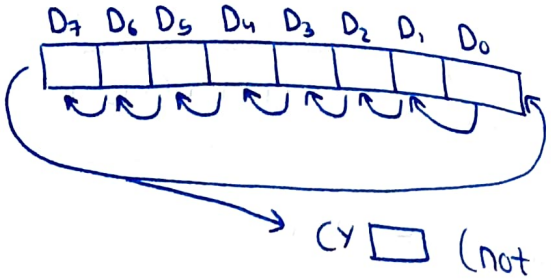
* Each binary bit of acc. is rotated left by one position.
* Bit D_7 is placed in position of D_0 as well as carry flag.

* Cy modified according to D_7.

* S, Z, P, AC flags are not affected } Only Cy is affected.

⊙ RLC

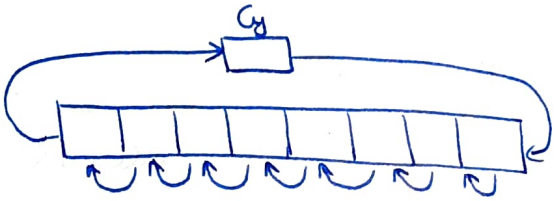
* 4 times RLC will reverse the content.



(10) Opcode ⇒ RAL
Operands ⇒ None
} * Rotate acc. left through carry.

* D_7 placed in carry flag and carry flag placed in D_0

⊙ Eg. ⇒ RAL



(11) Opcode ⇒ RRC
Operands ⇒ None
} * Right accumulator right.

* reverse of RLC.

* used to check whether no. is odd / even.

(12) RAR
Operands ⇒ None
} * rotate acc. right through carry.

* reverse of RAL

* AA ⇒ becomes 81H.

— x — rotating instructions — x —

(13) Opcode ⇒ CMC
Operands ⇒ None
} * compliment carry.

* carry flag complimented.
* no other flags affected.
* Eg. ⇒ CMC ⇒ CY = CY'

(14) Opcode ⇒ STC
Operands ⇒ None
} * Set carry

* carry flag is set to 1.
* Eg. ⇒ STC ⇒ CY = 1.

Ques. ⇒ Write an ALP in 8085 to convert an 8 bit binary no. (75) to Gray no. (4F).

⊙ STA ⇒ CY = 1
CMC ⇒ CY = 0
LDA 2050 ⇒ A ← 2050
MOV B, A ⇒ B = A
RAR
XRA B
STA 3050
HLT ⊙

(75) ⇒ 01110101
00111010
XOR ⇒ 01001111 ⇒ (4F)

Ques. ⇒ ALP to determine sum of digits of 8 bit no. (75)F)

⊙ LDA 2050 ⇒ A ← 2050
MOV B, A ⇒ B = A = 75
ANI 0F ⇒ A = 05
MOV C, A ⇒ C = A = 05
MOV A, B ⇒ A = B = 75
ANI F0 ⇒ A = 70
RLC
RLC
RLC
RLC → ADD C ⇒ A = A + C
STA 3050 HLT ⊙

A = 07

Branching Instructions ⇒ alter the normal sequential flow.

* these instructions alter either unconditionally or conditionally.
  (control is transferred to a specified location only when condition is true)

(1) Opcode ⇒ JMP
Operands ⇒ 16bit address
} * jump unconditionally

* program seq. is transferred to memory location specified by 16-bit address given in operand.
  - ⊙ Eg. ⇒ JMP 2034H ⇒ PC = 2034
* Immediate addressing mode + 3byte inst.

(2) Opcode ⇒ Jx
Operands ⇒ 16bit address
} * jump conditionally

* - - - based on specified flag of PSW.
  - ⊙ Eg. ⇒ JZ 2034 H
    ↓
    Jump to 2034 if ZF (zero flag) = 1

* Conditions ⇒

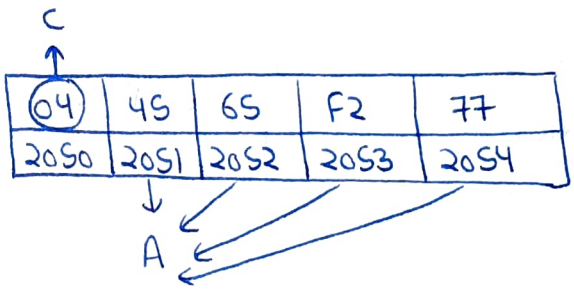
Opcode	Description	Status flags
JC	Jump if carry	CY = 1
JNC	— no carry	CY = 0
JP	— positive	S = 0
JM	— minus	S = 1
JZ	— zero	Z = 1
JNZ	— non zero	Z = 0
JPE	— parity even	P = 1
JPO	— parity odd	P = 0

* AF not possible as not in control of the user.

* ALP to which determines the larger of 2 nos. available in 2050 (F2) and 2051 (23).

```
LHLD 2050
MOV A,L
CMP H        => A-H (F2-23)  CY=0
JNC LOOP1    => jump only if CY=0
MOV A,H
LOOP1: STA 3050
HLT
```

* ALP to determine largest no. in an array.




```
LXI H,2050
MOV C,M
DCR C
INX H
MOV A,M
LOOP2: INX H
CMP M
JNC LOOP1    => for smallest no. => JC LOOP1
MOV A,M
LOOP1: DCR C
JNZ LOOP2
STA 3050
HLT
```

(all same)

⊙ for comparison one data must be in accumulator

⊙ ALP to determine sq. root of a no. =>

* will give a perfect root as 8085 deals with 8 bit no.

⊙ Eg. => No. = 9

=> 9-1 = 8
8-3 = 5
5-5 = 0

} * keep subtracting odd nos. until we get 0. => no. of subtractions = sq. root

=> MVI C,01 => C => no. of steps

MVI D,01 => D => 1, 3, 5 ...

LDA 2050 => A ← content of 2050 [9]

LOOP2: SUB D => A = A-D

JZ LOOP1 => if zero flag = 1

INR D
INR D } => D = D+2
INR C

JMP LOOP2

LOOP1: MOV A, C

STA 3050 => content of acc. copied to 3050 mem. location.

HLT

Introduction to 8085 =>

=> introduced in 1977.

=> 8-bit microprocessor

=> NMOS device => (e^- metal oxide semiconductor)

=> has 40 pins Dual Inline Package (DIP).

* Clock frequencies of 8085 are =>

8085A - 3MHz

8085AH2 - 5MHz

8085AH1 - 6MHz

* Various types of signals of 8085 =>

(1) Address Bus and Data Bus =>

* Address bus is group of 16 lines (A0 to A15).

* unidirectional => bits flow from MPU to peripheral devices.

⊙ There is multiplexing of lower order address bus and data bus. (AD0 to AD7).

* At one time, up can perform only one task i.e. it can either deal with address or data bus.

∴ we have to separate add. & data in AD0 to AD7 ↓

=> done using Control signal.

(2) Control and Status Signals =>

(a) ALE => Address Latch Enable Signal. It goes high during first T state of a machine cycle.

=> It enables lower 8 bits of address if value is 1.

=> It activates data bus from AD0 to AD7 if value is 0.

(b) IO/M' => It is status signal that distinguishes whether the address is for memory or I/O.

=> when high (1) => it is for I/O

=> when low (0) => it is for memory.

(c) S0, S1 => These are status signals sent by microprocessor to distinguish b/w various operations.

* RD' => when it is 0, read operation.

* WR' => when it is 0, write operation.

(both cannot be 0 at one time)

* READY => used by up to sense whether peripheral is ready to transfer data or not.

=> if it is high (1) => peripheral is ready.

=> if it is low (0) => peripheral is not ready.

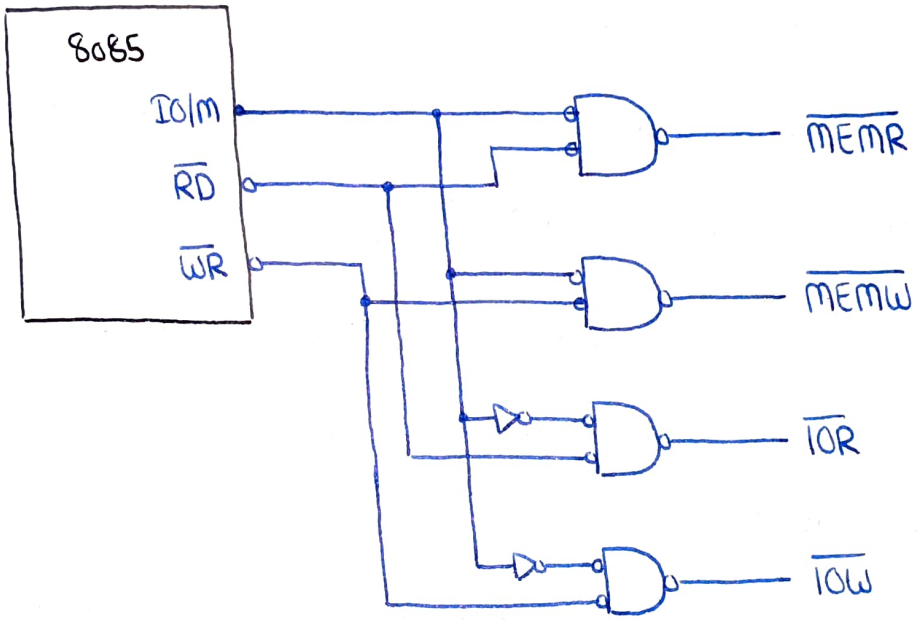
S1	S0	Operation
0	0	Halt
0	1	Write
1	0	Read
1	1	Opcode Fetch

⊙ memory read => data is coming from memory to up.

⊙ memory write => data going from up to memory.

Operations	IO/M'	S1	S0
Opcode Fetch	0	1	1
Memory read	0	1	0
Memory write	0	0	1
I/O read	1	1	0
I/O write	1	0	1
Interrupt Ack.	1	1	1
Halt	High Impedence	0	0

* Schematic to generate Control Signals =>



(3) Power Supply and Freq. signals =>

* V_{CC} => +5V power supply * V_{SS} => Ground reference

* X_1, X_2 => A crystal is connected at these 2 pins. Freq. is internally divided by two, ∴, to operate a system at 3MHz crystal should have freq. of 6MHz.

* CLK (Out) => Signal can be used as the system clock for other devices.

(4) Interrupt Signals => 8085 has 5 int. signals that can interrupt the execution of program.

=> INTR, RST 6.5, RST 5.5, RST 7.5, TRAP

=> µp acknowledges interrupt signal by INTA signal.

(5) Reset Signals =>

* RESETIN => when signal on this pin goes low => program counter is set to zero, thus buses are tristated and MPU reset

* RESETOUT => this signal indicates that MPU is being reset.

=> can be used to reset other devices.

(6) DMA signals => used for fast data transfer techniques.

* HOLD => It indicates that another device is requesting use of address and data bus.

=> the processor after removal of HOLD signal regains the bus.

* HLDA => It indicates that the hold req. has been received and control of address and data bus is transferred to DMAC (8257).

(7) Serial I/O signals => 2 signal to transmit serials =>

(1) SID => dataline for serial i/p.

(2) SOD => dataline for serial o/p.

* Serial transfer of data is transmitted bit by bit on a single line.

* SID is used in RIM instruction and SOD in SIM instruction.

Stack and Subroutine in 8085 =>

(group of memory locations present in read/write memory)

(group of instructions which is written separate from main program)

* **Stack** => An area of memory identified by programmer for temporary storage of information.

=> LIFO structure.

=> Stack normally grows backwards into memory => Programmer defines bottom of stack and stack grows into reducing address range.

=> Stack ptr. is initialised highest memory available in RAM.

=> Stack is defined by setting SP (stack ptr) register.

LXI SP, FFFFH → (end of memory of 8085)

* Push Instruction ↴

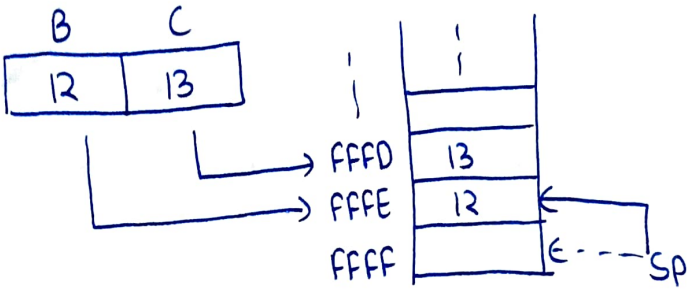
=> PUSH B (B refers to BC reg. pair => can use any pair)

⊙ Decrement SP

⊙ copy contents of reg. B to memory location pointed by SP.

⊙ Dec. SP

⊙ copy contents of reg. C ——— .



* Pop Instruction ↴

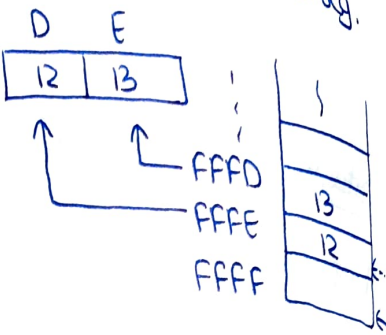
⇒ POP D

⊙ copy contents of memory location pointed by SP to E reg.

⊙ Increment SP

⊙ copy ——— to D reg.

⊙ Increment SP



* The PSW Register Pair ⇒ Program status word.

⇒ This pair is made up of Accumulator and flag register.

(high order) (low order)

⇒ It is possible to PUSH and POP on this pair.

(similar to normal reg. pairs)

⊙ ALP to find 2's comp. of contents of flag register.

LXI SP 3999 ⇒ SP ← 3999

PUSH PSW ⇒ push value of acc. & flag to stack

POP B ⇒ pop value from stack to BC reg. pair

MOV A,C ⇒ A ← C

CMA ⇒ A ← 1's comp. of A

INR A ⇒ A ← A+1

MOV C,A ⇒ C ← A

PUSH B ⇒ push value of BC to stack

POP PSW ⇒ pop value from top of stack to Acc. and Flag.

HLT ⇒ Stop.

* **Subroutine** ⇒ sequence of program instructions that perform specific task repeatedly.

⇒ It is implemented using Call and Return instructions.

* Unconditional Call Address (Call Address) ⇒

⇒ After execution of this instruction program control is transfered to a sub-routine whose starting address is specified in instruction. Value of PC is transfered to stack and value of SP decremented by 2.

⇒ **CALL = PUSH + JMP**

* same as function in C language.

* Conditional Call Address ⇒ Program control is transfered to subroutine only if condition is satisfied.

* same table as JMP ⇒ just replace J with C.

* Unconditional Return ⇒ program control is transfered back to main program. Value of SP inc. by 2.

⇒ **RET = POP + JMP**

* Conditional Return ⇒ replace J by R. (all same)

* Advantages of Subroutine ⇒

(1) Decomposing a complex prog. to simpler steps.

(2) Reducing duplicate code in a program.

(3) Enabling reuse of program across multiple programs.

(4) makes debugging easy.

* 8085 code to convert binary no. to ASCII code ⇒

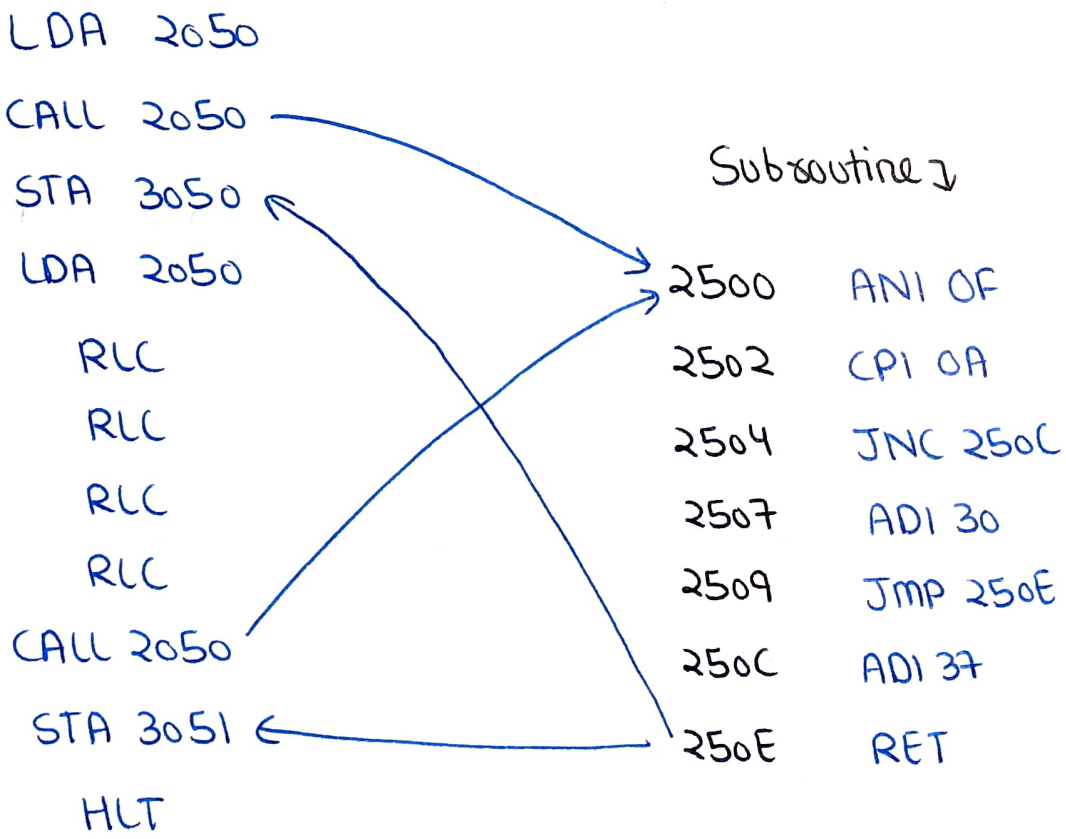
⊙ Eg. ⇒

i/p data →	4A
address →	2050

41	34	⇒ o/p
3050	3051	⇒ addres

Digit → 4 ⇒ 52 (ASCII D) ⇒ 34 (ASCII H)

→ A ⇒ 65 (ASCII D) ⇒ 41 (ASCII H)



* Interrupts in 8085 => (disturbances)

=> When up receives any interrupt request from peripheral(s) which are req. its services, it stops current execution and program control is transfered to subroutine by generating CALL signal and after executing subroutine by generating RET signal again program control is transferred to main program from where it had stopped.

* when up receives interrupt signals, it sends an acknowledgement (INTA') to peripheral which is req. for service.

if value → 0 => req. accepted
→ 1 => req. ignored

⊙ Types of Interrupts =>

(1) Hardware Interrupts => when up receives signals through pins (hardware) => hardware interrupts.

* 5 hardware interrupts =>

INTR, RST 7.5, RST 6.5, RST 5.5, TRAP

(2) Software Interrupts => inserted in b/w the program that means these are mnemonics of up.

* 8 software interrupts =>

RST 0, RST 1 ..... RST 7

* Vector Addresses are calculated => 8 * TYPE (in Hex.)

* Vectored Interrupts ⇒ Interrupts having fixed vector address (starting address of subroutine) and after executing these, program control is transfered to that address.

⊙ Out of 5 ⇒ 4 are vectored ⇒ TRAP (RST 4.5) 24H
RST 5.5 2CH
RST 6.5 34H
RST 7.5 3CH

* Non-vectored Interrupts ↴

⇒ vector address is not predefined.

Interrupting device gives the address of subroutine for these interrupts.

⊙ INTR is only non-vectored int. in 8085.

* Maskable Interrupts ⇒ can be disabled or ignored by up.

⇒ they are either level triggered or edge triggered.

⇒ Eg. ⇒ INTR, RST 7.5, RST 6.5, RST 5.5. (Notifications in PC)

* Non-maskable ⇒ cannot be disabled or ignored by up.

⇒ it consists both edge and level triggering.

⇒ Eg. ⇒ TRAP (RST 4.5).

⇒ used in Critical power failure conditions. (Battery in PC)

* Priority of Interrupts ⇒

⇒ When up receives multiple int. signals simultaneously, it'll execute Interrupt Service Req. (ISR) according to priority of interrupts.

Interrupt	Priority
TRAP	Highest
RST 7.5	2nd Highest
RST 6.5	3rd Highest
RST 5.5	2nd Lowest
INTR	Lowest.

Machine Control Signals =>

⊙ no addressing mode

⊙ size of every inst. is 1 byte

(1) HLT => It halts any further execution and CPU enters into wait state. Contents of registers are unaffected during HLT.

(2) NOP => Only instruction that is fetched and decoded but not executed.

=> used to increase the time delay

=> used to delete and insert and delete instructions while troubleshooting.

=> to delete any n byte instruction (undesirable) => insert NOP instruction n times.

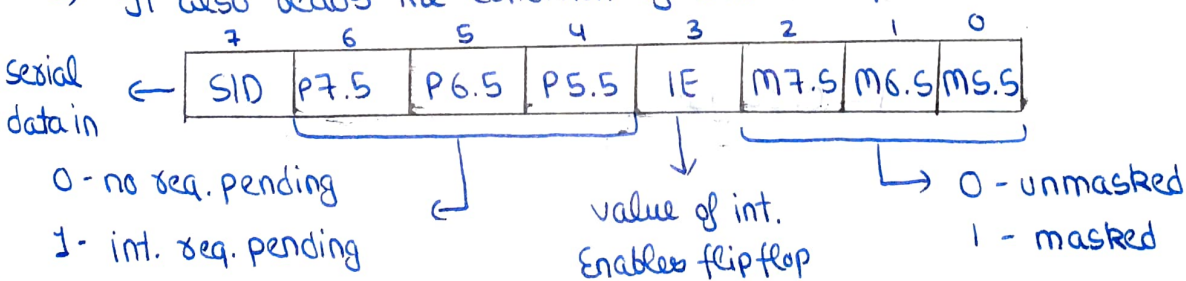
(3) DI (disable interrupts) => Interrupt enable flip flop is reset and all interrupts are disabled except TRAP.

(4) EI (Enable interrupts) => Interrupt enable flipflop is set and all interrupts are enabled.

* RIM (Read Interrupt Mask) => This instruction is used to read the status of hardware interrupts (RST 7.5, RST 6.5, 5.5) by loading into A register. [SID => data coming in μP]

=> defines the condition of mask bits for interrupts.

=> It also reads the condition of SID on μP.



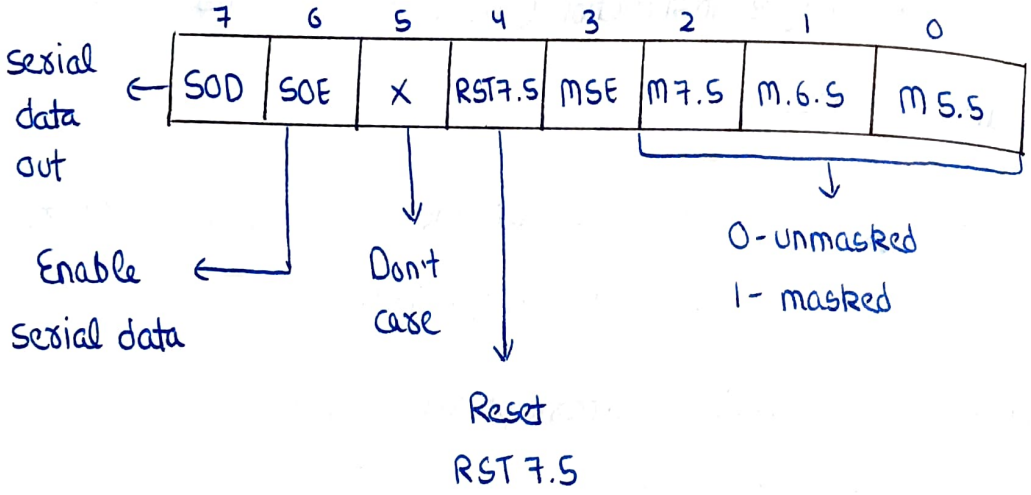
* SIM (Set Interrupt Mask) => It is used to implement Hardware interrupts by setting various bits to form masks or generate o/p data via SOD line.

=> first, req. value is loaded in accumulator, then SIM will take bit pattern from it.

⊙ In table ↓

=> to make SOD effective => SOE must be 1

=> MSE must be 1 to make M7.5, M6.5, M5.5.



* Timing Diagram => It represents the execution time taken by each instruction in a graphical format.

=> Execution time is represented in T-states.

* T-states => System clock period.

$T = 1/f \Rightarrow f = 3MHz$

$T = 0.33 \mu sec.$

* Machine cycle ⇒ Total time req. to execute any operation.

OR Time req. to access memory or ip/op devices.

* Instruction cycle ⇒ Time req. to execute any instruction.

OR, Tstates to complete an instruction.

OR, Machine cycles req. to execute an instruction.

⊙ Machine cycle ⇒

* opcode fetch = 4T
* memory read = 3T
* memory write = 3T
* I/O read = 3T
* I/O write = 3T

* first operation is always opcode fetch (4T) ⇒ min. Tstate of any instruction is 4T.
* Eg. ⇒ MOV A, B (A ← B)

(i) fill the bytes first.

2005 : opcode - 4T

(both A & B are part of µp ∴ no other operation req.)

⊙ memory read ↓

⇒ data coming from memory to µp.

⊙ memory write ↓

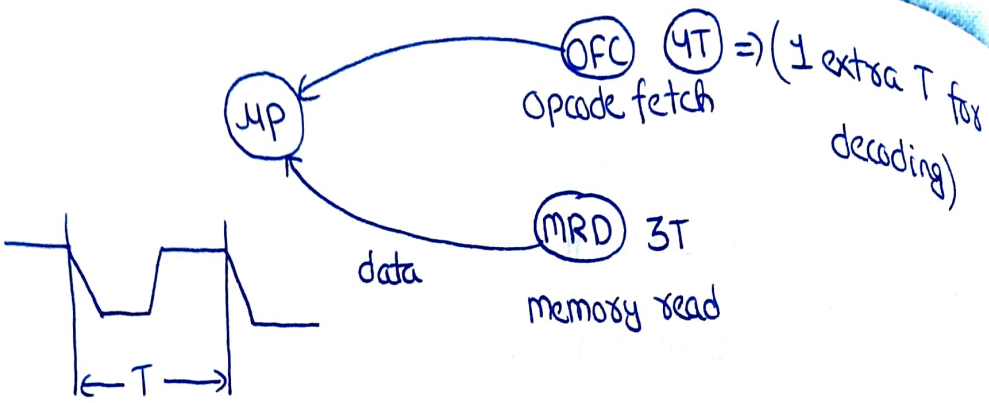
⇒ data going from µp to memory).

⊙ I/O read ⇒ data coming to acc. from any i/p port.

⊙ I/O write ⇒ data going from acc. (µp) to o/p port.

* In Implied and Register Add. modes, only 4T req.

⇒ as only opcode fetching happens.



⊙ Eg. ⇒ **MOV A,M**

⇒ 2000 : OFC - 4T
A ← M - MRD - 3T
} (7T states)

⊙ Eg. ⇒ **MVI B 45**

⇒ 2000 : OFC - 4T
2001 : 45 - MRD - 3T

* (7 states)

* μP doesnot accept data directly from user ⇒ it has to be stored in memory first.

* In case of Immediate Add. mode ⇒ if data is of 8bit
⇒ always 7T states.

⊙ if 16 bit data ⇒ 10T states.

Eg. ⇒ **LXI H 3050**

⇒ 2000 : OFC - 4T
2001 : 50 - MRD - 3T
2002 : 30 - MRD - 3T
} (10)

⊙ Eg. ⇒ **LDA 3050**

⇒ 2000 : OFC - 4T
2001 : 50 - 3T
2002 : 30 - 3T
2003 : A ← 3050 - MRD - 3T
} (13T)

* Why in MOV A,M , only 7T states are used?

⇒ M is contents of HL reg. pair ⇒ HL already part of µp.

∴ no extra GT is req.

and, µp already knows value of H-L.

⊙ Eg. ⇒ **STA 3050** ⇒ same as LDA

⊙ Eg. ⇒ **SHLD 3050**

⇒ 2000 : OFC - 4T
2001 : 50 - MRD - 3T
2002 : 30 - MRD - 3T
3050 ← L - MWR - 3T
3051 ← H - MWR - 3T

} (16T states)

⊙ Things we need to know to find T states ?

(1) function of instruction

(2) how much bytes it consumes in memory.

(3) Addressing modes.

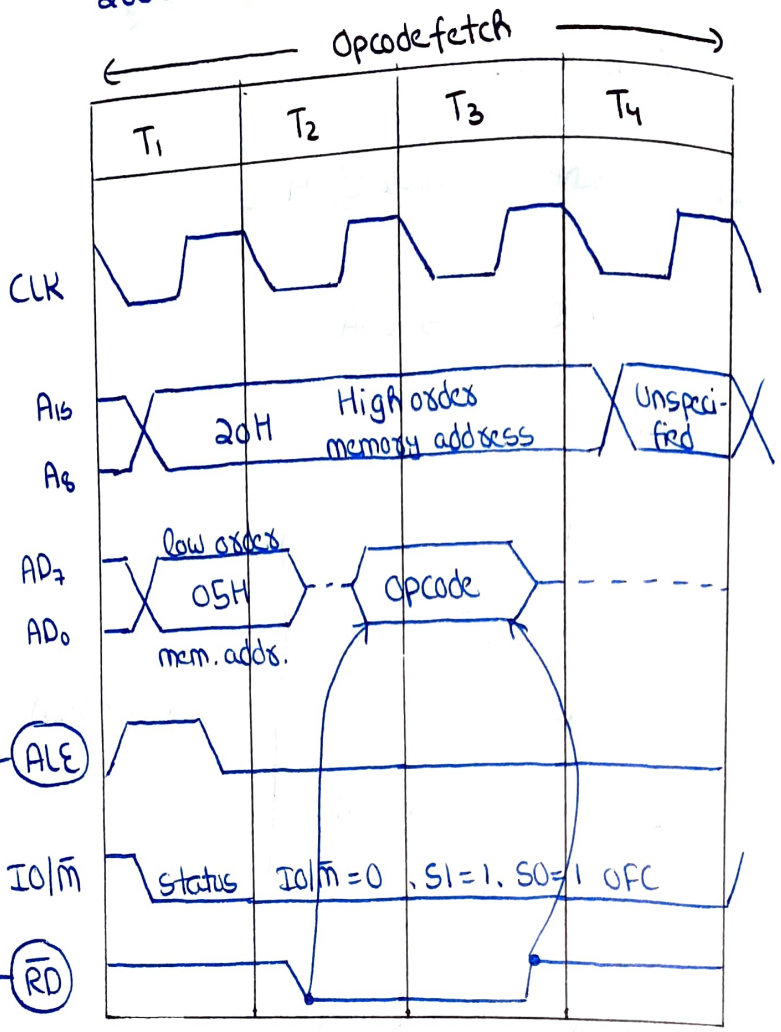
⊙ Eg. ⇒ **IN 50** ⇒ (same for OUT)

⇒ 2000 : OFC - 4T
2001 : 50 - MRD - 3T
A ← 50 - IO/RD - 3T

* 10T states.

* **Timing Diagram** ⇒ Eg. ⇒ MOV C, A

2005 : opcode



(demultiplexes address and data bus ⇒ during T_1 always high)

(when it is 0 (low) ⇒ read operation)

⊙ this above 4T is common in all diagrams.

⊙ Eg. ⇒ STA 8000

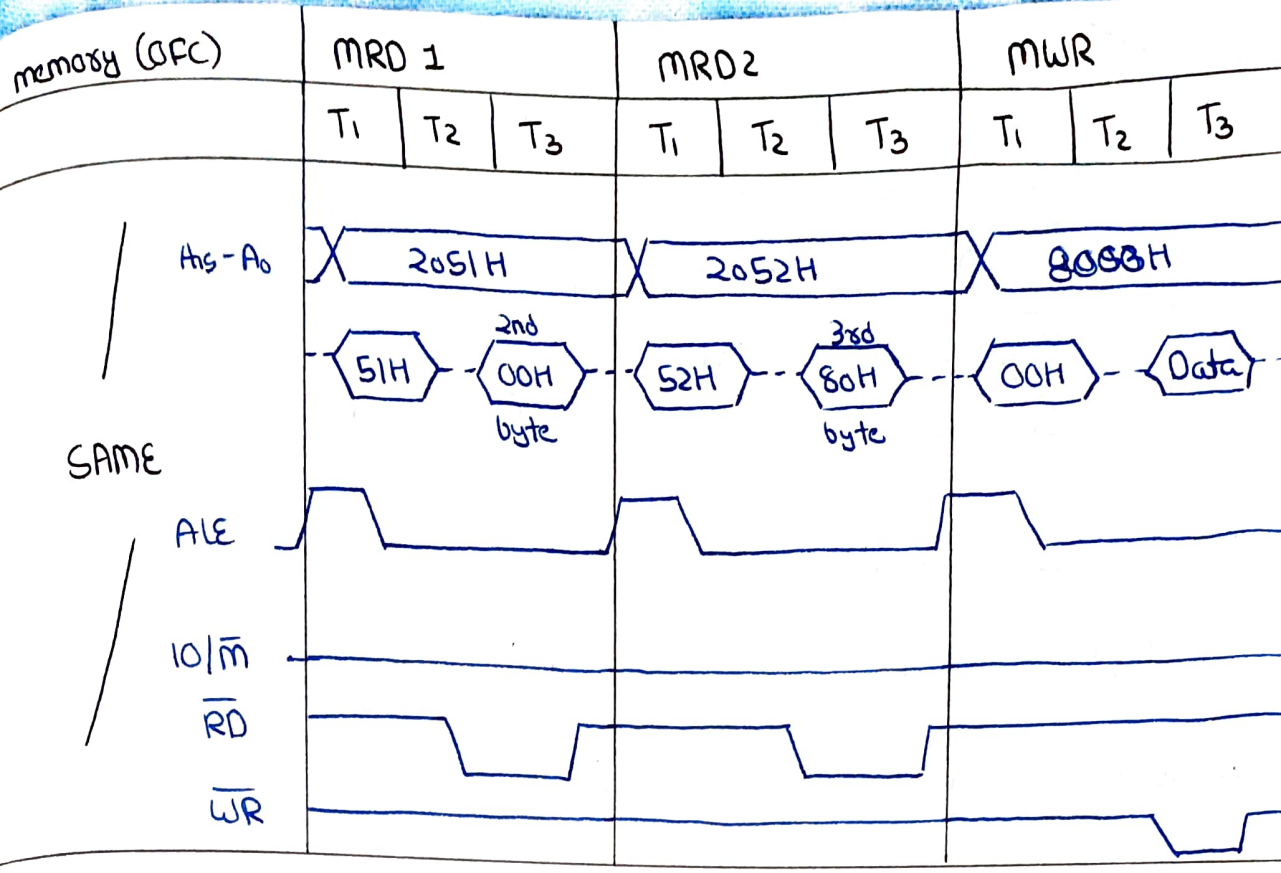
2050 : OFC - 4T

2051 : 00 - MRD - 3T

2052 : 80 - MRD - 3T

A → 8000 - MWR - 3T

* at one time either $\overline{RD}$ or $\overline{WR}$ is low.



⊙ IN C0 ⇒ 4125 : 0FC

4126 : C0 - MRD

A ← C0 - IO/RD

Time Delay Calculation in 8085 ⇒

* TDelay = T0 + TL

T0 ⇒ delay outside the loop

TL ⇒ delay of the loop

TL = ((TIL x N) - 3)T } → (TIL ⇒ Time inside loop)

⊙ Eg. ⇒ MVI C, FFH - 7T

LOOP DCR C - 4T

JNZ LOOP - 10/7T

⇒ FF = 255 ⇓ (single reg)

T0 = 7T states

TL = ((14 * 255) - 3)T = 3567T

Tdelay = 3567 + 7 = 3574T

⊙ let F = 2MHz ⇒ T = 1/F = 0.5μs

⇒ Tdelay = 3574 * 0.5 = 1787 μs

⊙ Eg. ⇒ LXI B, 1000H - 10T

LOOP DCX B - 6T

MOV A, C - 4T

ORA B - 4T

JNZ LOOP - 10T

$T_0 = 10T$

$T_L = ((24 * 4096) - 3)T$ (Decimal of 1000)

$= 98301T$

$T_{delay} = 98311T$

* DCX doesnot affect ZF

(to set zero flag)

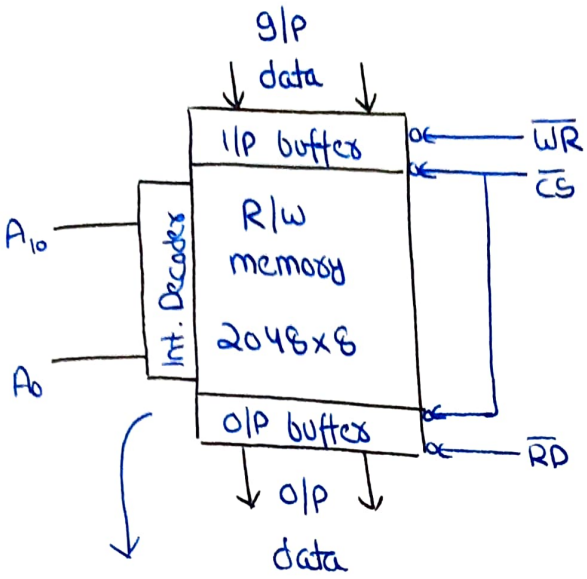
* To increase Time delay ⇒ use nesting of loops.

Memory Interfacing ⇒

* µp needs to access data in code which are stored in memory.

⇒ this access is done by memory interfacing.

Memory mapped I/o	I/o mapped I/o
(1) I/O treated as memory.	(1) I/O treated as I/O.
(2) 16 bit addressing	(2) 8 bit addressing.
(3) Can address 64K locations	(3) can address 256 locations.
(4) more decoders Hardware req.	(4) less dec. req.
(5) memory instructions used.	(5) special inst. like IN, OUT used.
(6) memory control signals ($\overline{MRD}$, $\overline{MWR}$) used.	(6) special signals (IORD, IOWR) used.
(7) Arithmatic & Logical op. ✓	(7) Ar. & Logical op. ✗
(8) Data transfer b/w register & memory.	(8) Data transfer b/w Acc. & I/O.

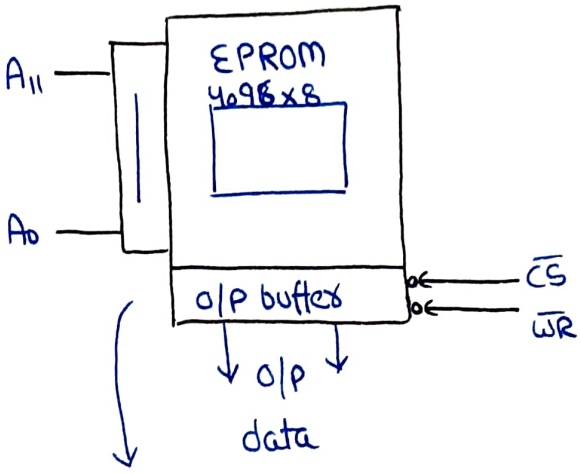


2048 - no. of registers (2×1024) (2^{10})

8 - size of each reg.

$(A_0 - A_{10}) \in (2^{11})$

(* Erasable Programmable Read only memory)



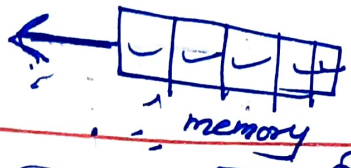
$4096 \Rightarrow 4 \times 1024 \Rightarrow_2 10$

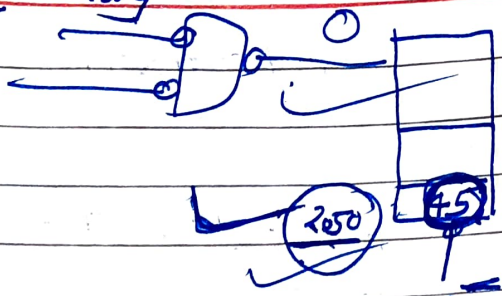
$\Rightarrow 2^{12}$ $(A_0 - A_{11})$

(1) Determine 8085 address lines which are connected to memory chip by using formula ⇒ Mem. capacity = 2^ address lines

(2) Remaining add. lines of 8085 are connected to decoder ⇒ O/P of decoder connected to $\overline{CE}$ which enables memory chip.

(3) Generate control signal. It will enable O/P buffer.





⇒ Mem cap = ?

4KB

$4 \times 2^{10} = 2^{A.L.}$

$2^{12} = 2^{A.L}$

A.L = 12

$A_0 - A_{11}$

2^{12}

000000000000

111111111111

$16\,KB = 2^4 \times 2^{10} = 2^{A.L}$

A.L = 14 lines

(i) A.L ⟶ memory chip $A_0 - A_{15}$

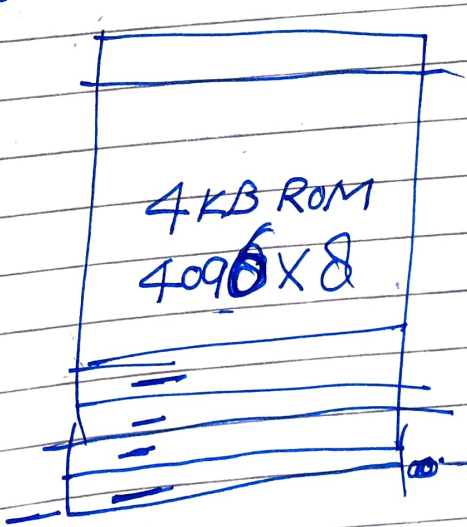
(ii) remaining lines of 8085 up A15 are connected with decoder or NAND lines it will enable $\overline{CS}$ of memory

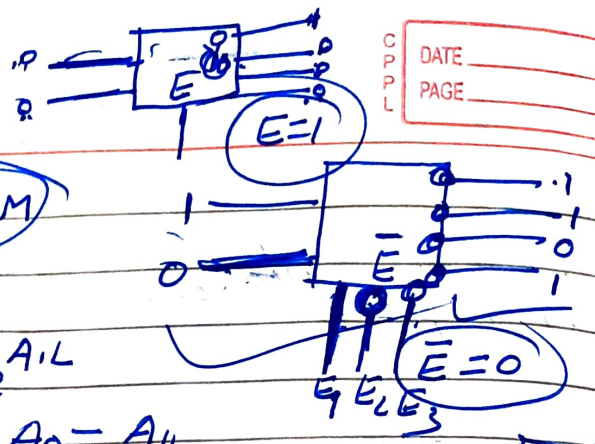
(iii) Generate control signals MEMR

Memory

4096 regs

Size of each reg is 8 bit

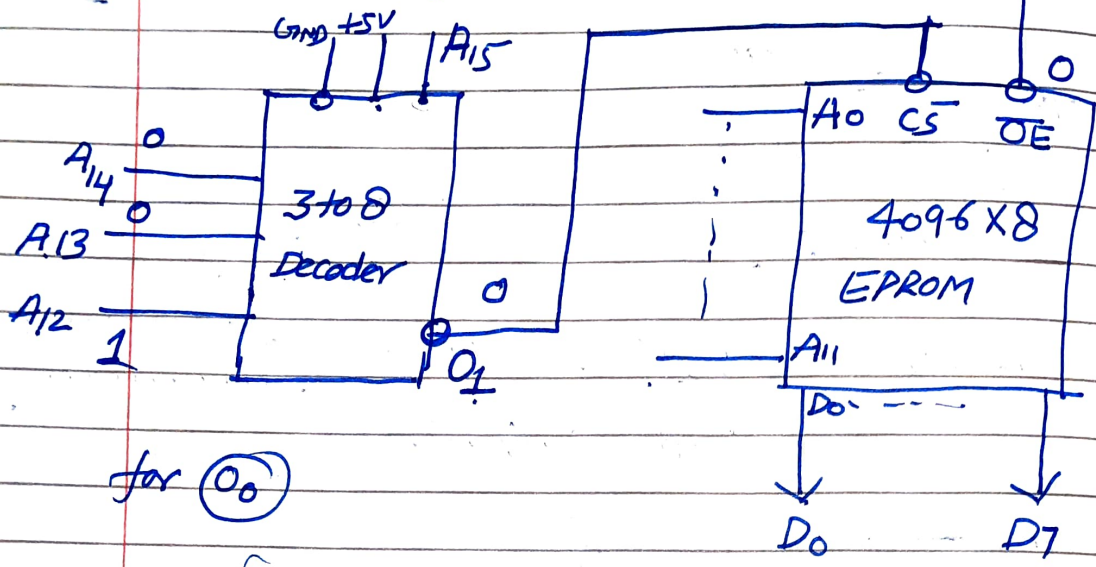




4KB - EPROM

$4KB = 2^{12} = 2^{A_1L}$

$A_1L = 12 \Rightarrow A_0 - A_{11}$



for O_0

Addressing range

A15	A14	A13	A12	A11 ——— A0
1	0	0	1	0000 0000 0000
1	0	0	1	1111 1111 1111

9000 to 9FFF H

for O_0 ⇒ 8000 to 8FFF

1101

1101 O_5 ⇒ D000 to DFFF

Taught By:
Hemant
Pokhariya Sir

Prepared By:
Ayank Gupta