UNIT-II

Levels of Prog. languages { low level, high level

→ It is a notation for writing prog which are specified specific" of a comput" / algorithm.

(1) Low level languages

- → ↑η
- → optimal use of memory & processing time
- → But takes ↑ time to write ∴ used for small codes
- →

(2) High level language

- → Fast development of large prog.
- → ↓η
- → each LOC relates to 1 m/c instruct" code.

• Prog. language can also be { compiler language, Interpreter language.

(1) M/C language

- → programming in m/c native code.
- → consists of strings of 0 & 1
- → ↑ mistakes can be done.

(2) Assembly language.

- → Symbolic represent" of m/c code, but computer cant understand it
- → " design" of mem. loc"
- → An instruct" to add can be :-

add B, A

(dest" source & save result in dest")

- → This code is ~~forced to~~ related to m/c lang. using assembler

↳ The code in assembly language = source code
" " " m/c = object code

(3) Compiler language → high level assembly lang.
↳ The program written is converted to m/c lang using compiler.
↳ After the source code is compiled, the object code can be executed anytime later.

(4) Interpreter language
↳ Also a high level language.
↳ Uses interpreter program to execute user's prog.
↳ Interpreter converts the code line by line.

8085 Microprocessor → (Used in washing m/c, ovens, mobile phones)

↳ 8 bit microprocessor by Intel
↳ 100% s/w compatible
↳ ↑ performance.
↳ 16 bit add. bus ∴ 2^{16} mem loc^n
↳ 16 bit Prog Counter (PC)
↳ 16 bit Stack ptr (SP)
↳ Architecture follows Von-Neuman Architecture

Functional Units

- Registers
  ↳ It has built in register array usually labelled as A (Accumulator), B, C, D, E, H & L.
  ↳ Special purpose registers are :- 16 bit { PC (Prog. counter), SP (stack pointer) }
  8 bit – flag reg

- **Accumulator (A)**
  - 8 bit reg attached to the ALU.
  - Reg stores result of 8-bit data & performs arithmetic & logic operations.

- **Register Pairs**

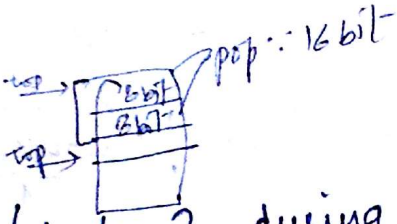
Called general purpose registers

16 bit	B (8)	C (8)
16 bit	D (8)	E (8)
16 bit	H (8)	L (8)

→ low order byte

Temp Reg. 8	
B 8	C 8
D 8	E 8
H 8	L 8
Stack Ptr (SP) 16	
P C. 16	

- **Stack Pointer**
  - 16 bit
  - Always ↑/↓ by 2 during PUSH & POP oper^n
  - holds addr. of topmost entity of stack
  - Increment/Decrement is used to ↑ or ↓ value of sp



- **Prog. Counter**
  - Holds add. of next inst. to be executed
  - Microprocessor ↑ the PC by 1 whenever an instruct^n is being executed. So that it can point to the next instruct^n to be executed.

- **Temporary Register**
  - Holds temporary data of arithmetic & logic oper^n.
  - holds intermediate results. Also holds data from general purpose registers

- **Flag Reg**
  - 8 bit reg
  - has 5 one bit FF which hold 0/1 based on result stored in accumulator.

5 flip flops are

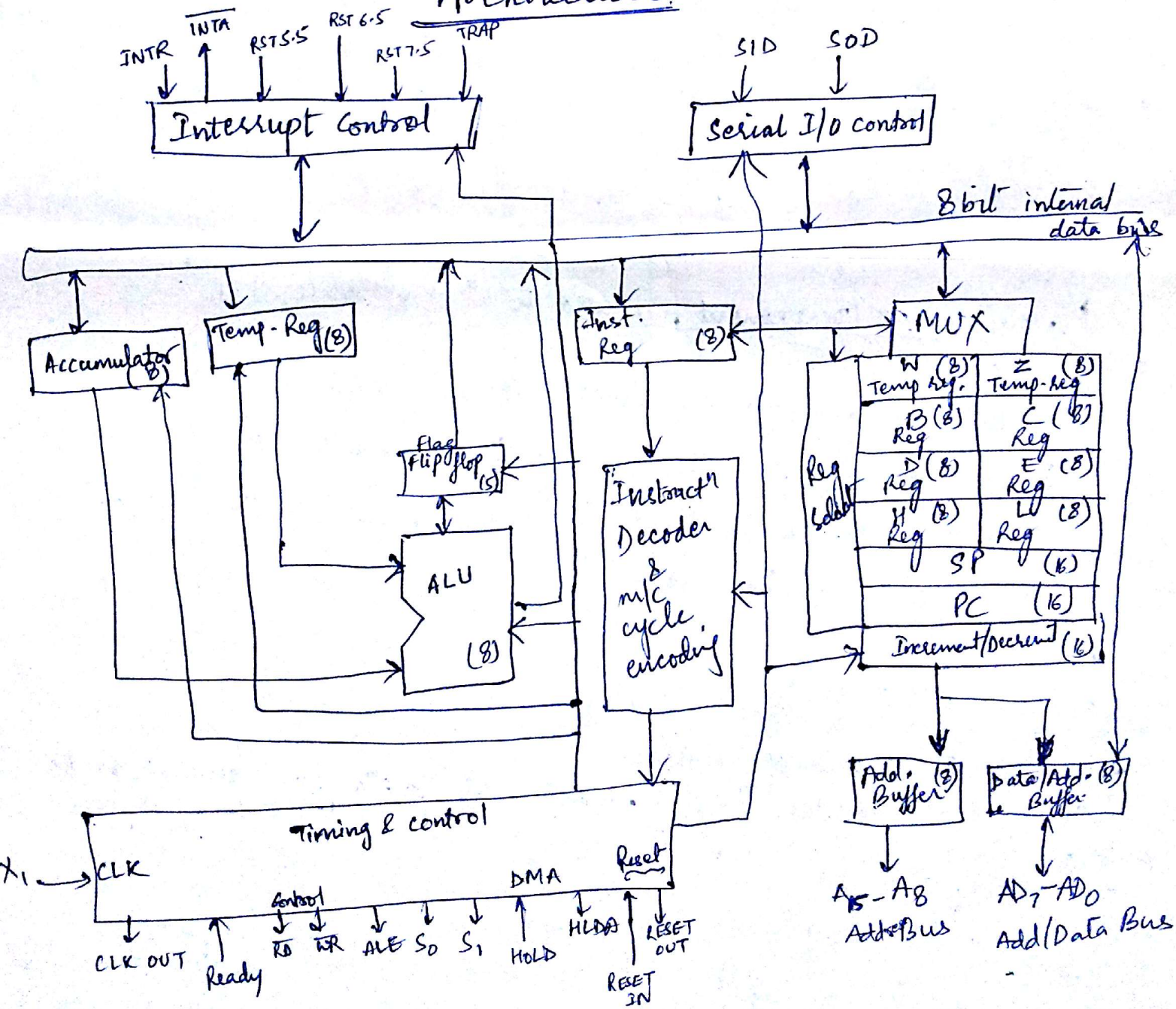
-ve = 1
+ve = 0

- Sign (S): It tells sign of the result stored in Accumulator
- Zero (Z): Tells whether the result in acc. is zero or not after oper^n.
  If result = 0; Z = 1
  " " = 1; Z = 0
- Aux. Carry (AC): Used in BCD oper^n; If carry is there in result ⇒ AC = 1
  If no carry ⇒ AC = 0
- Parity (P): Tells parity of data stored in acc; even parity = 1, odd parity = 0
- Carry (C)

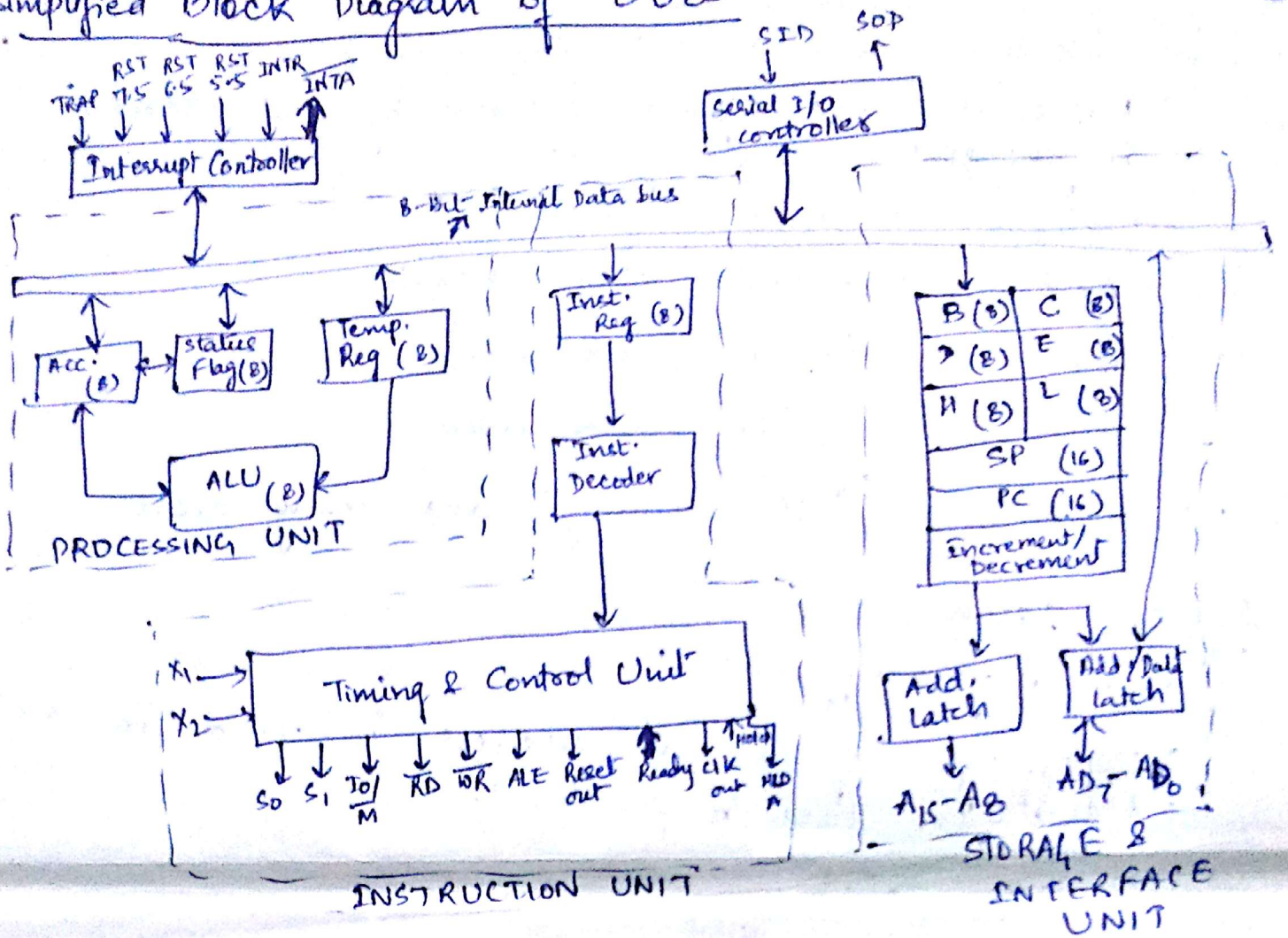
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	Z	X**	AC	X**	P	X**	CY

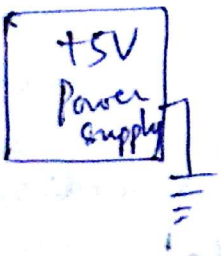
X** = Don't care cond^n

Architecture

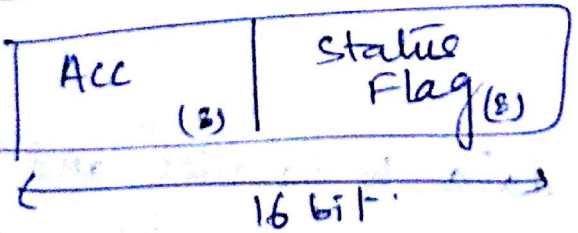


Simplified Block Diagram of 8085





- **Prog. Status Word (PSW)**



- **Inst. Reg (IR)**
  holds the current inst. to be executed.

- **Inst. Decoder**
  - Interprets the contents of IR
  - Generates m/c cycles based on the inst. in IR
  - M/c cycles are given to Timing & control unit

- **Timing & Control Unit**
  - Controls oper" of microprocessor & peripheral devices
  - Based on m/c cycles received by inst. decoder, it generates control signals.
    - S_0, S_1 = Status signals
    - ALE = Address latch Enable
    - $\overline{RD}$ = (Read, active low)
    - $\overline{WR}$ = (Write, active low)
    - $IO/\overline{M}$ = (I/O, Memory) and many more

- **Address latch**
  - Group of 8 buffers
  - Upper-byte of 16 bit address is stored in it
  - Add. is then given to peripherals

- **Address/Data latch**
  - lower byte of ~~8-bit~~ 16-bit add. is stored. or 8 bit data is stored.
  - It is decided by ALE signal.
    - If ALE=1 ⇒ It contains lower byte of address
    - If ALE=0 ⇒ It contains 8 bit data

- **Serial I/O Controller**
  - Used to convert serial data to parallel and vice versa
  - Microprocessor works with 8-bit parallel data.
  - Serial I/O devices work with serial X-fer of data
  - This unit is the interface b/w microprocessor & serial I/O devices.

- **Interrupt Controller**
  - Receives interrupts as per priority and applies them to microprocessor.
  - One outgoing signal called $\overline{INTA}$ = interrupt acknowledge.
  - TRAP → ↑ priority eg → in power failure
    RST 7.5 → 2nd ↑ priority
    RST 6.5
    RST 5.5
    } H/W interrupts

INTR → when this is high, the processor does its work & sends an active low to $\overline{INTA}$

Addressing Modes of 8085

(Various formats for specifying operands is called addressing)

(1) **Immediate Addressing**

Data is immediately loaded to the destin^n

eg → MVI R, data

(2) **Register Addressing**

Data is provided through registers

eg → MOV R1, R2

(3) **Direct Addressing**

Data is directly sent to/received from accumulator

eg → IN 00D (I/P data from ~~as~~ port 00D to accumulator)

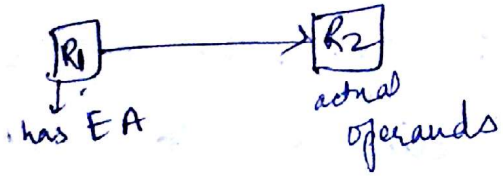
OUT 00A (O/P data from accumulator to port 00A and subsequently to O/P device)

④ Indirect Addressing

- Data is x-ferred from addr. pointed by the data in reg to another.

MOV A, M

- 2 accesses are used :- 1st to extract address
  2nd to " data
- Here, the effective address (EA) of operand is stored in register



INSTRUCTION FORMAT

↳ Instruction is a command given to microprocessor to perform a specific task.

↳ Instruction has 2 parts

- Operation code (opcode) — tells the oper^n to be performed
- Operand

- The operand may be directly or indirectly specified here.

Instructions can be of 3 types :-

↳ (1) One byte Instruction (3) 3 byte inst.
(2) 2 byte inst.

① One byte instruction (for register to register xfer).

- Here opcode & operand are in the same byte.

Eg → (i) MOV C, A
(opcode) (operand)

Equivalent binary code → 01 001 111 = 4F in hex.
MOV C A

MOV D, E = 01 010 011

(ii) ADD B
(opcode) (operand)

Here A is assumed. The contents of B are added to contents of A. 10000,000

Binary code = 10000000 = 81 H
ADD B

ADD H = 10000100

- All reg, reg pair & oper^n have specific binary codes

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

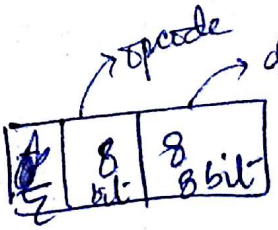
Reg. Pair
BC → 00
DE → 01
HL → 10

ADD → 10000 sss
MOV → 01 ddd sss

② Two byte instructions

1st byte = Opcode
2nd byte = operand

Eg MVI A, data
↳ move immediately



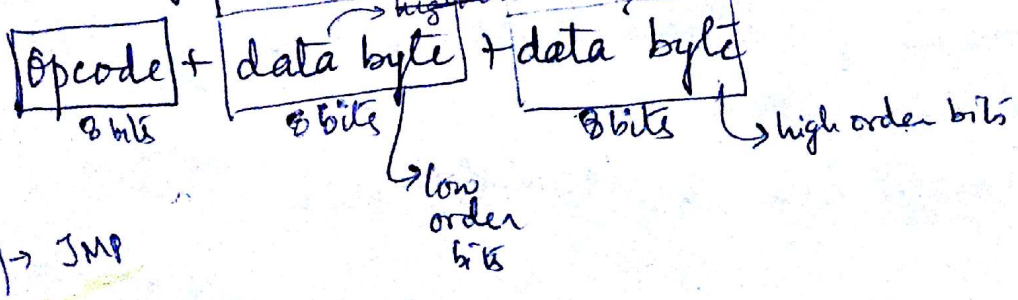
let data = 32H.

⇒ MVI A, 32H
1 byte 1 byte

This is represented in contiguous mem. loc^n.

③ 3 byte instructions

1st byte = opcode
2nd & 3rd byte = 16 bit ~~address~~ data → Total 16 bit data



Eg → JMP

INSTRUCTION SET OF 8085

(1) Data X-fer instructions

(i) MOV B, C

(ii) MVI B, 57H → data

(iii) LDA 20H → 16 bit addr. of mem-loc^n

(iv) LHLD 16 bit addr. (load H&L reg)

(v) STA 16 bit addr (Store accumulator direct in memory)

Here contents of accumulator are copied into memory loc^n.

(vi) SHLD 16 bit addr.

Store H&L register in memory loc^n.
The contents of 'L' are stored in mem loc^n given and contents of 'H' are stored in "mem-loc^n + 1".

SHLD + 8 bit (low order) + 8 bit (high order)
Total 16 bit add.

(vii) XCHG

Here contents of reg. H are exchanged with contents of reg D and contents of L are exchanged with contents of E.

(viii) X THL

Exchange top of stack contents with reg. L and
" (top+1) of stack with reg. H.

(2) Arithmetic Instructions

(i) ADD B (Contents of B added to A)

(ii) ADI 8bit data

Add immediate data to accumulator
ADI 20H

(iii) ADC B

Add reg. to accumulator with carry.

(iv) INR R
Increment reg. or mem by one
Here contents of R are ↑ by 1 & result is stored there only.

(3) Logical Instructions

(i) ANA R
contents of R and contents of A are logically ANDed and result stored in A.

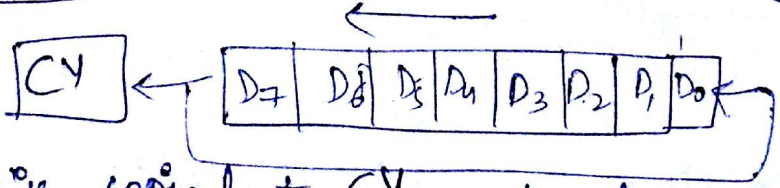
(ii) ANI 8 bit data
Result of logical AND of 'A' & 8 bit data is stored in A.

(iii) ORA R (logical OR of A & R)

(iv) CMP R
Compare reg. with contents of A.

if A < reg ⇒ carry flag = 1
if A = reg ⇒ Zero flag = 1
if A > reg ⇒ Carry & zero flag = 0

(v) RLC (rotate accumulator left)



D_7 is copied to CY and also rotated to D_0.
the CY bit is updated as per value of D_7 & rest (S,Z,P,AC) remain same.

(vi) RAL (rotate acc. left through carry)

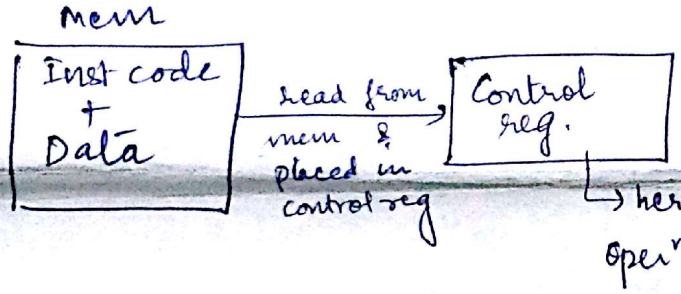
Bit D_7 is placed in carry flag
& Carry flag is placed in D_0.

Ques Add 2 hex no.s 23H & 84H & write assembly code for it.

Soln.
```
MVI , A, 23H
MVI   B, 84H
ADD   B
MOV   C, A
HLT
```

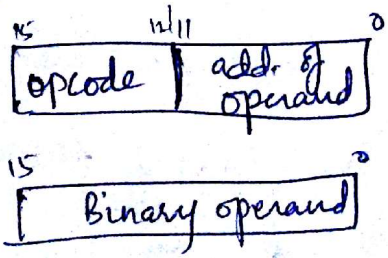
INSTRUCTION CODE

- It is a group of bits that tell the computer what oper^n to be performed.



- The length of instruct^n code must be as per the no. of distinct oper^n is available in computer.
- If 64 oper^n are available ∴ 6 bit inst. code must be used.
- Let ADD is represented as 110010. After decoding this, control signals are generated to read operands.

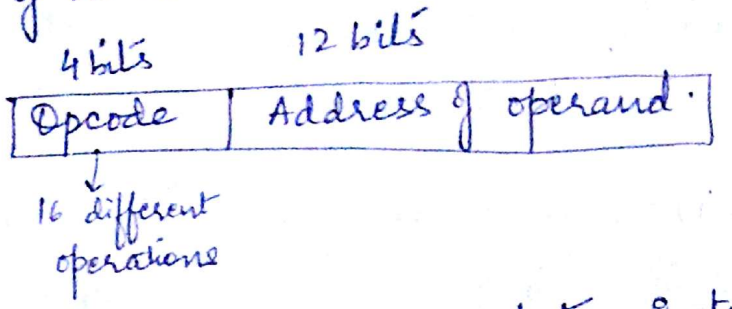
How instructions are stored in mem



How a program is stored ?

(instruction)

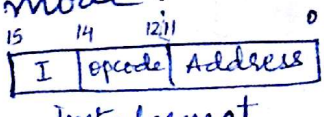
- If we have a 16 bit memory word and we store inst. in it, then out of 16 bits :



- The control reads this complete instruct^{n} from memory, uses the 12 bits to locate operand & executes the 4 bit operation specified.
- If the operand is not taken from memory (like in some cases), the remaining bits are used for other purposes.

Direct & Indirect Addressing

- When instruction code doesnot store the address of operand, then it contains the operand itself; It is called immediate operand
- And when the instruction stores (in its 2nd part), address of the operand, then it is called as Direct address.
- When the bits in 2nd part of instruction store an address of a memory word that contains the operand, then it is called indirect address mode.

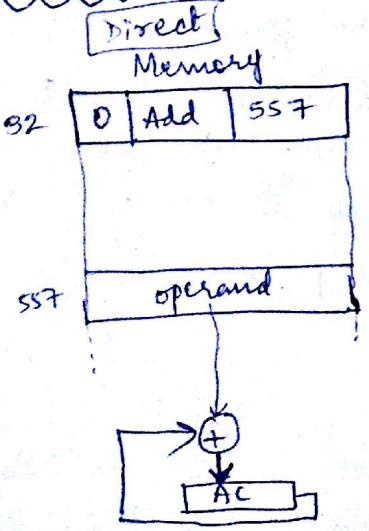


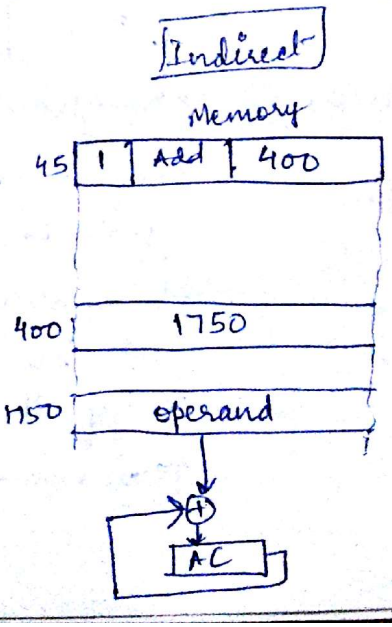
Inst. format

when I = 0 ⇒ Direct addressing

when I = 1 ⇒ indirect addressing

Here opcode specifies an ADD operation





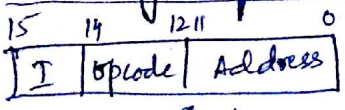
Computer Registers

- After the inst. code is read from memory, it is stored in some register.
- The memory capacity is of 4096 words & each word can have 16 bits.

Register	Bits	Register Name	Function
DR	16	Data Reg.	Holds operand
AR	12	Address Reg	holds address of memory
AC	16	Accumulator	Processor Reg.
IR	16	Inst. Reg.	holds inst. code
PC	12	Program counter	holds add. of next inst.
TR	16	Temp. Reg.	holds temporary data
INPR	8	I/P Register	holds I/P character
OUTR	8	O/P Register	holds O/P character

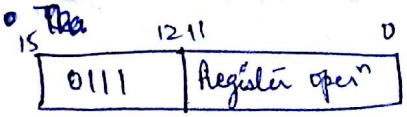
Computer Instructions

Memory Reference



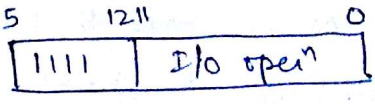
- Here 12 bits specify the address.
- If I = 0 ⇒ direct address
  If I = 1 ⇒ indirect address.
- opcode = (000 to 110)

Register Reference



- Recognised by 0111.
- Reg. reference specifies oper^n on the AC only.
- So, no need of operand and bits 0-11 (12 bits) specify the oper^n to be performed

I/O instruction



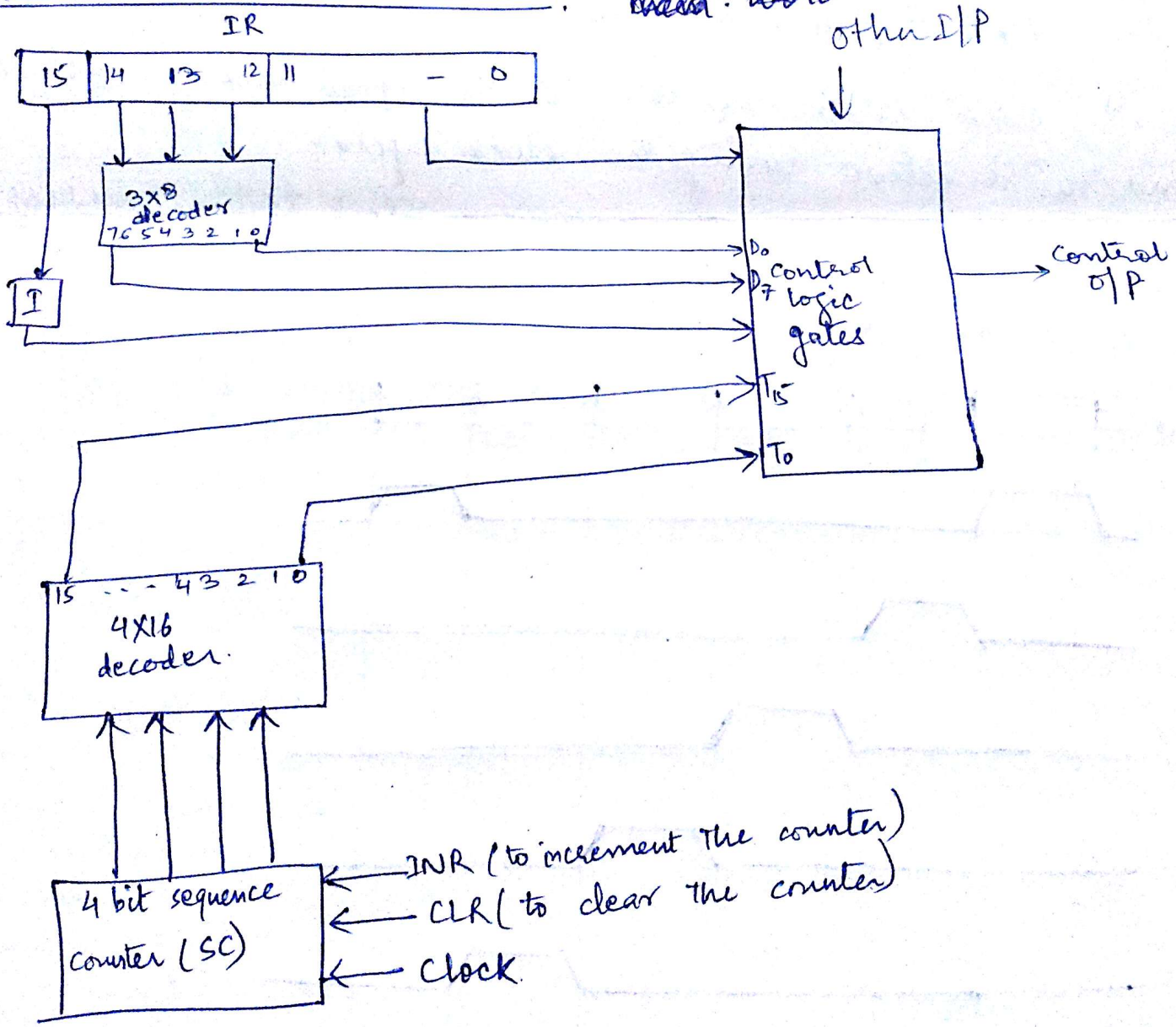
- Recognised by 1111
- Remaining 12 bits specify the type of I/O oper^n to be performed.

• The type of instruction is specified by the last 4 bits (12 - 15)

→ If (12-14) bits ≠ 111 (opcode can never be 111) ⇒ memory reference type & bit 15 ≙ value of 'I'

→ If (12-14) bits = 111 ⇒ inspect the bit no. 15.
if 15th bit = 0 ⇒ Register ref.
if 15th bit = 1 ⇒ I/O inst.

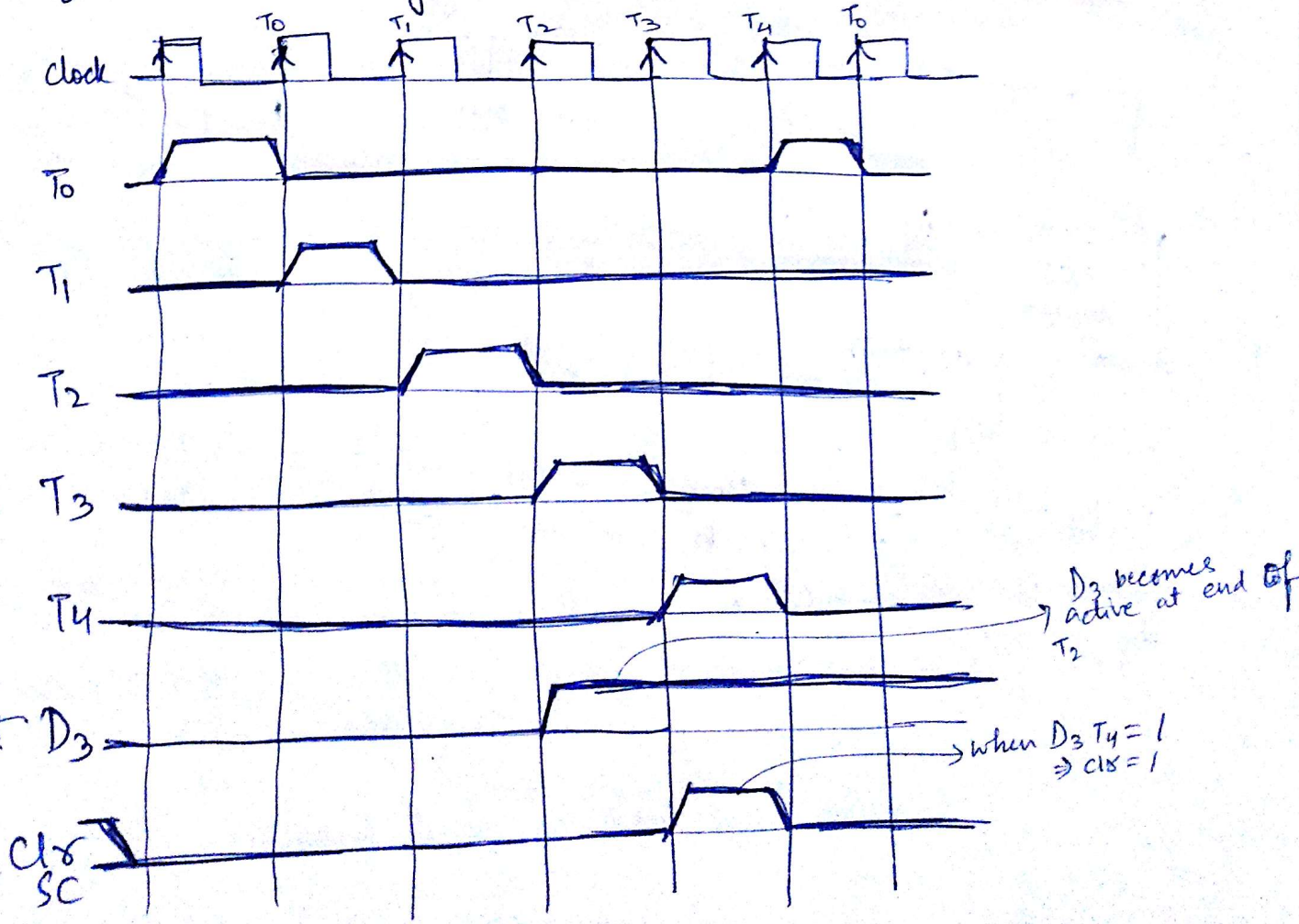
TIMING & CONTROL (considering mem has $4096 = 2^{12}$ words)



- SC responds to +ve X-sition of clock.
- The O/P of counter SC are decoded into 16 timing signals T_0 to T_{15}
- The SC is ↑ or cleared synchronously.
- Most of the time SC is ↑, but once in a while it is cleared to 0 causing next active timing signal to be T_0.
- let SC is ↑ to give signal T_0, T_1, T_2, T_3, T_4 in seq. At time T_4, SC is cleared to 0 if D_3 is active

  ∴ $D_3 T_4 : SC \leftarrow 0$

- At 1st, clock clears SC to 0 ⇒ T_0 signal becomes active and T_0 is active during the clock cycle.
- SC is <u>incremented</u> with every +ve clock X-sition, unless CLR is active. If SC is not cleared, it will continue to ↑ T_0 to T_{15} Then again T_0.



Instruction Cycle

- A program under execution goes through a cycle set of instructions called the instruct^n cycle. And these instructions are then divided into sub instructions.

Phases of instruction Cycle

Fetch → Decode → Read → Execute

- Fetch: Instr^n is fetched from memory
- Decode: Inst^n is decoded
- Read: EA is read from memory
- Execute: Instr^n is executed

① Fetch Phase

- Initially At time T_0 SC ← 0, The PC contains next instr^n to be executed. The PC loads the address of next inst. into AR (12 bits)
- The instr^n stored at M[AR] is loaded in the IR & PC is ↑ by 1 at time T_1

$\therefore$ T_0 : AR ← PC

T_1 : IR ← M[AR], PC ← PC+1

② Decode Phase

- All bits of inst. in IR are analysed & decoded at time T_2.
- At time T_2, opcode (12-14 bits) in IR is decoded and indirect bit (15th bit) is xferred to flip flop I. & Address part (0-11 bits) is xferred to AR.
- Here SC gets incremented after each clock pulse T_0, T_1, T_2

(3) Decision / Read Phase

At first (12-14 bits) are decoded

(i) • If decoder o/p D_7 is 1 ⇒ (111)
∴ instrⁿ must be reg. reference or I/O reference.
• Then last bit of FF 'I' (15th bit) is decoded.
If IR(15) = 0 ⇒ registin reference
IR(15) = 1 ⇒ I/O reference.

∴ T_2 : $D_7 \leftarrow$ Decode IR(12-14), $0 \leftarrow IR(15)$ (Reg. ref.)

T_2 : $D_7 \leftarrow$ Decode IR(12-14), $1 \leftarrow IR(15)$ (I/O ref.)

(ii) If decoder o/p $D_7 \neq 1 \Rightarrow$ (not 111)

∴ IR(12-14) is decoded b/w D_0 to D_6. ⇒ Instrⁿ = mem. Reference

• Now last bit i.e. IR(15) is checked.
If IR(15) = 0 ⇒ Direct addressing
IR(15) = 1 ⇒ indirect addressing
• Then 1st 12 bits (0-11) are used to trace the operand.

∴ T_2 : $D_0, D_1, D_2 \ldots D_6 \leftarrow$ Decode IR(12-14), $0 \leftarrow IR(15)$

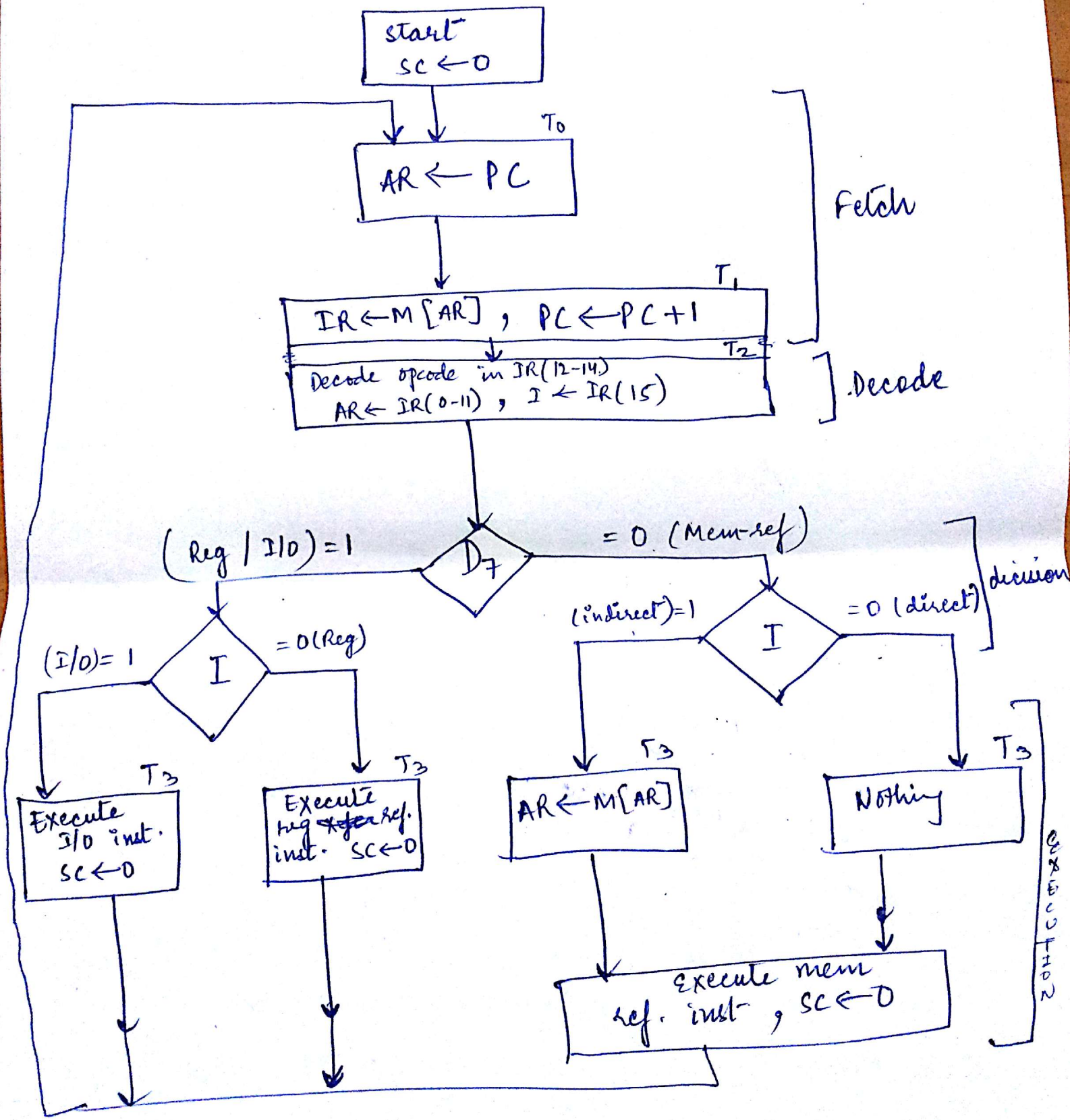
T_2 : $D_0, D_1, D_2 \ldots D_6 \leftarrow$ Decode IR(12-14), $1 \leftarrow IR(15)$

(4) Execution Phase.

Now the instrucⁿ is executed as mem ref / registin ref / I/O reference. at next clock pulse i.e. (T_3).

$D_7' I' T_3$: Direct addressing
$D_7' I T_3$: $AR \leftarrow M[AR]$ (indirect addressing)
$D_7 I' T_3$: Registin reference and $D_7 I T_3$: I/O reference

Instruct^{n} cycle



Memory Reference Instructions

- Memory reference instructions refer to those instructions that are invoked from decoded o/p D_0 to D_6 (in-lining & control unit). These oper^n are carried out at time T_4
- The instructions are as follows:

Symbol	Oper^n decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow Cout$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$, $PC \leftarrow AR+1$
ISZ	D_6	$M[AR] \leftarrow M[AR]+1$, If $M[AR]+1=0$ then $PC \leftarrow PC+1$

(1) AND to AC

- AND oper^n is performed on the bits of AC and mem word specified by the effective address. The result is transferred to AC.

D_0T_4 : $DR \leftarrow M[AR]$

D_0T_5 : $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$

- Here 2 clock transitions are required. At T_4 operands are x-ferred from mem. to DR. At T_5, the result of AND operation of AC and DR is stored in AC.

(2) ADD to AC

- The contents of memory word specified by AR are added to AC.
- The sum is X-ferred to AC at O/P carry C_{out} is Xferred to E (extended accumulator) flip flop.

$\therefore$ $D_1 T_4$: $DR \leftarrow M[AR]$

$D_1 T_5$: $AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$

(3) LDA : Load to AC

- This X-fers the mem. word specified by effective address (EA) to AC.

$\therefore$ $D_2 T_4$: $DR \leftarrow M[AR]$

$D_2 T_5$: $AC \leftarrow DR$, $SC \leftarrow 0$

- Everytime the memory word gets Xferred to DR 1st, because there is no direct connection from AC to bus.

(4) STA : store AC

- It stores the contents of AC into the mem. word specified by EA.

$D_3 T_4$: $M[AR] \leftarrow AC$, $SC \leftarrow 0$

(5) BUN : Branch Unconditionally

- This instruct^n X-fers the program to the instruction specified by the EA.
- The BUN instruct^n allows the programmer to specify an instruct^n out of sequence & thus the program

. jumps unconditionally.

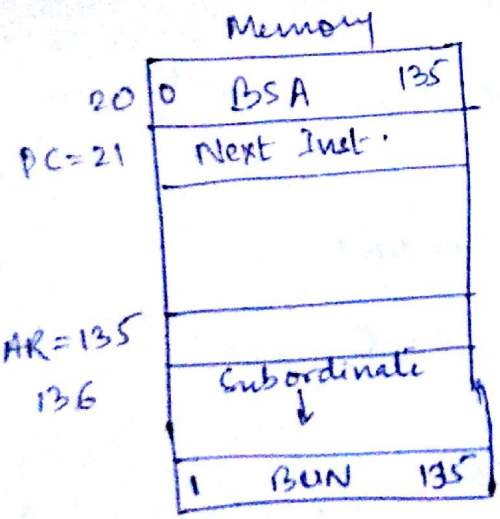
$D_4 T_4 : PC \leftarrow AR \;,\; SC \leftarrow 0$

- The EA from AR is given to PC, The next inst. is then fetched (∵ SC becomes 0) and executed from memory add. given by new value in PC.

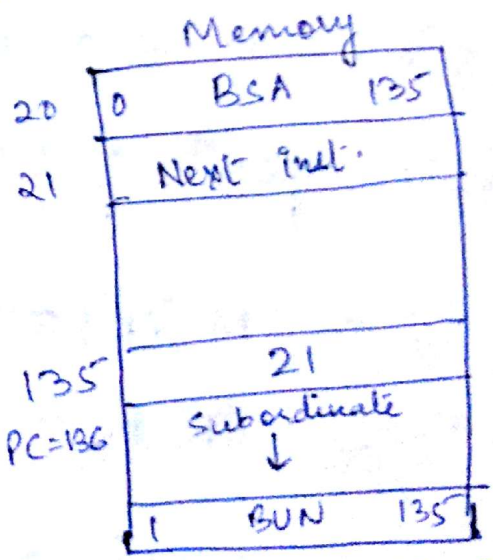
(6) BSA : Branch & Save Return Address

- This is useful for branching to a portion of the program called a subroutine or procedure.
- BSA stores the address of next instruct^n to be executed (which is stored in PC) into the mem loc^n given by EA.
- This EA+1 is stored in PC (This acts as mem loc^n of subordinate)

∴ $M[AR] \leftarrow PC \;,\; PC \leftarrow AR+1$



a) Mem PC & AR at time T_4



b) Mem & PC after execution

- Here BSA inst. is in mem address 20 (The inst. has I=0 and address part has binary equivalent of 135)

ADD to AC

- After the fetch & decode phase, PC contains 21. (∵ next inst. to be executed must be 21) and AR gets 135 ∵ it contains the inst. that has to be executed.
- The result of BSA oper^n (in fig (b)) shows:-

$$M[135] \leftarrow 21 \ , \ PC \leftarrow 135+1 = 136.$$

- After the inst. is executed, PC now containing the subordinate will get executed.
- After the subordinate prog is executed, the control is X-ferred to the original program (at address 21). This is done by BUN statement
- When BUN is executed, control goes to mem add. 135 and finds the previously saved address 21.
- The f^n of BSA is called as subroutine call and f^n of BUN is called as subroutine return.
- The whole oper^n of BSA is done in 2 clock pulses

$$D_5T_4 : \ M[AR] \leftarrow PC \ , \ AR \leftarrow AR+1$$

$$D_5T_5 : \ PC \leftarrow AR \ , \ SC \leftarrow 0$$

(7) ISZ : Increment & Skip if Zero

- This inst. increments the word in effective add. and if incremented word = 0 ⇒ PC is incremented by 1.
- This can happen when a -ve no. is stored as 2's complement. After repeated increments, it might reach 0.

- In this case, PC is incremented to skip this inst.
- The increment of memory word is done by transferring the word into DR and then ↑ it.

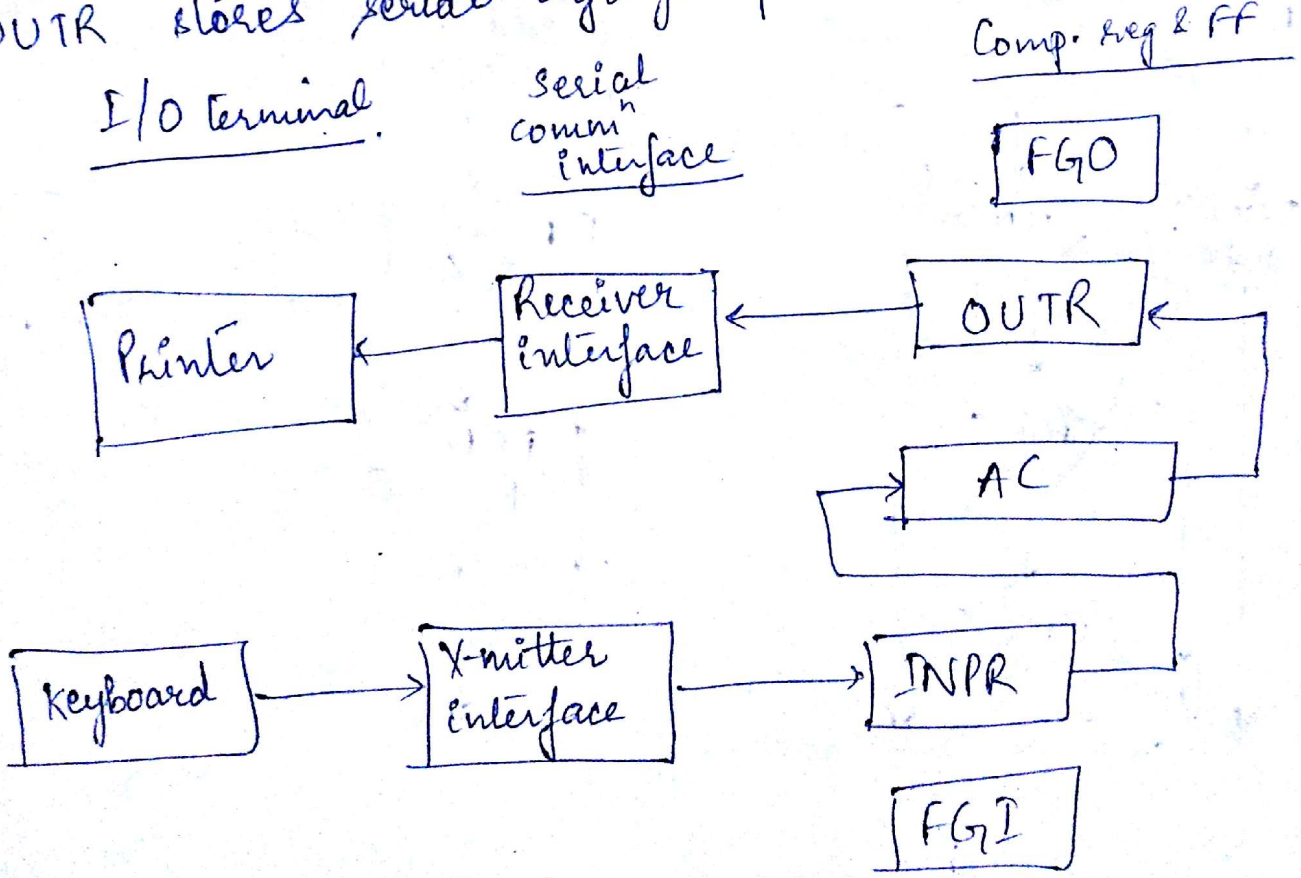
$\therefore D_6T_4 : DR \leftarrow M[AR]$

$D_6T_5 : DR \leftarrow DR+1$

$D_6T_6 : M[AR] \leftarrow DR,$
if (DR=0)
$PC \leftarrow PC+1$
$SC \leftarrow 0$

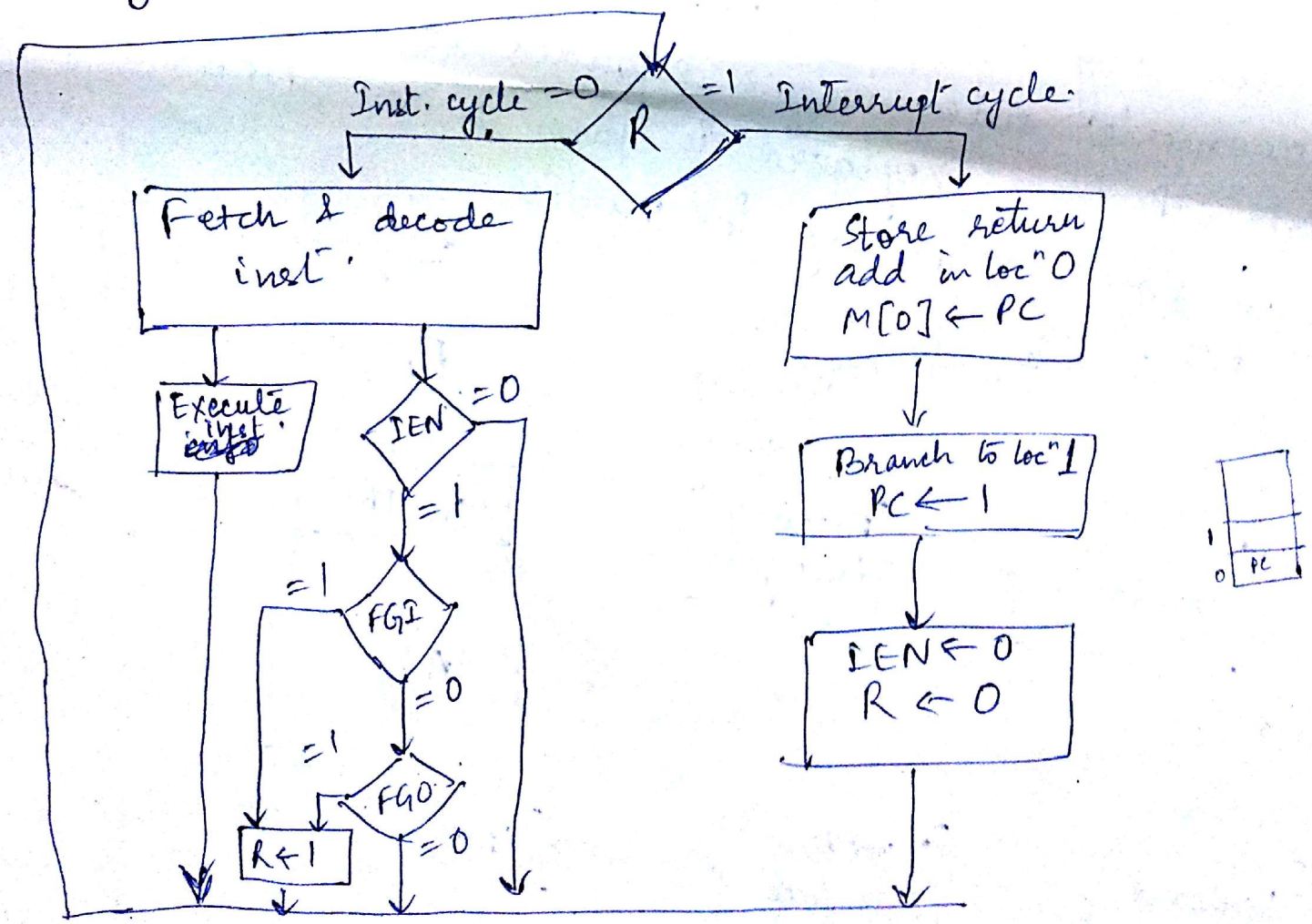
INPUT, OUTPUT

- whenever any info is input to the computer, an alphanumeric code is corresponded to it.
- Serial I/P from keyboard is given to the INPR
- OUTR stores serial info for printer



- Here FGI and FGO = 1 bit ~~reg~~ for I/P flag.
- Initially FGI = 0
  when keyboard is hit, an 8 bit code is shifted to INPR and FGI becomes 1
- Computer (control) sees that FGI = 1 and X-fers the contents of INPR to AC. and FGI becomes 0.
  After this new info can be loaded to INPR.
- Similarly, for OUTR, FGO = 1 (initially).
  when FGO = 1 ⇒ AC X-fers data to OUTR and hence to the receiver interface and FGO clears to 0.
  then again FGI becomes 1.

Program Interrupt



- For I/O oper^n to take place, each time flag is checked (cd)
  Thus this may result in wastage of time when I/O oper^n are very large. So it is not usually efficient to use flags.
- ∴ we may use interrupts.
- Whenever a computer is interrupted, flag bit becomes = 1
  This tells the computer that an interrupt has arrived and computer decides to perform the said oper^n.
- Flip flops used are IEN (Interrupt enable) & R (interrupt flip flop). when IEN = 1 ⇒ computer can be interrupted
- Initially 'R' flip flop is checked,
  → if R = 0 ⇒ computer goes into instruct^n cycle
  → In the execute phase of instruc^n cycle, IEN is checked.
  → if IEN = 0 ⇒ we donot want to use interrupt & it goes to next instr^n in ~~cy~~ instr^n cycle.
  → if IEN = 1 ⇒ check the control flags
  when FGI and FGO = 0 ⇒ registers are not ready for X-fer
  → when FGI or FGO = 1 ⇒ R is set to 1.
- Store ~~loc^n of~~ PC in a specific loc^n to be recovered later.