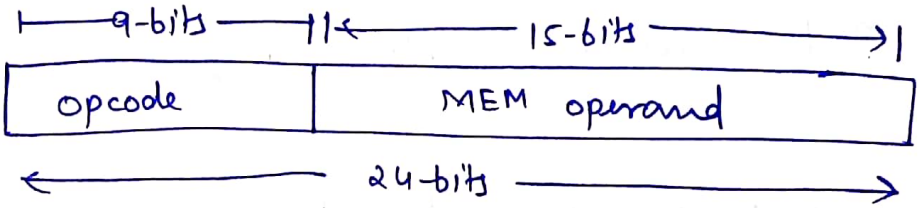
Q. Consider a hypothetical processor which supports both 1-addr Inst^n & 0-addr Inst^n. A 24-bit instruction is placed in a 32KBW memory. What is the range of a 1-Addr & 0-Addr Inst^n are possible in the system.

1-Addr Inst^n

primitive

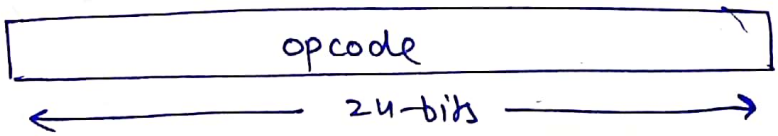


size of address of operand = ~~32 bit~~ $\log_2 32K$

= 15-bits

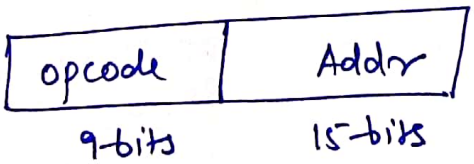
0-Addr Inst^n

Derived



Apply Expand opcode techniques :-

Step-1 :- Primitive inst^n



Step-2 :- Total number of opcodes = 2^9 = 512

Step-3 :- Possible free opcodes can be

{1 to 511}

So,

Range of 1-Addr Instⁿ : {1 to 511}

" 0-Addr Instⁿ : {1×2^15 to 511×2^15}

General Register CPU:-

- Based on the no. of register possible in the CPU, architecture is of two kinds :

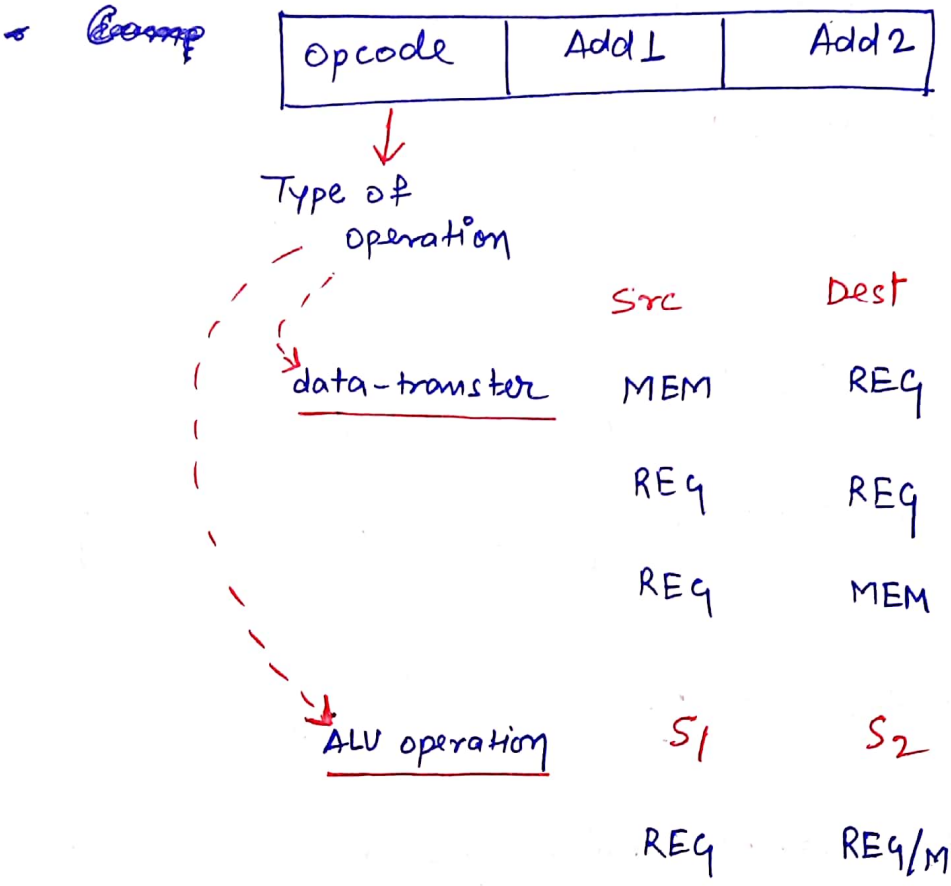
(i) Reg - memory refrence CPU (CPU contain less register)

(ii) Reg - Reg refrence CPU (CPU contain more register)

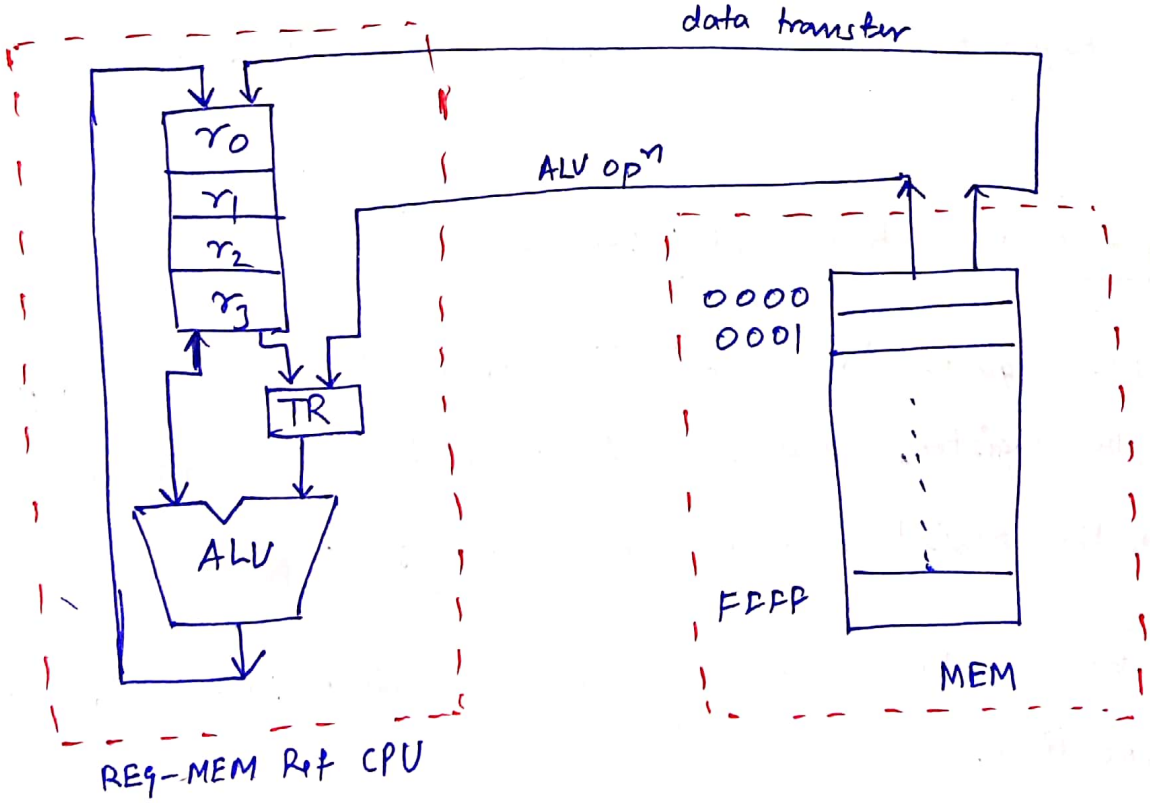
Reg - mem refrence CPU :-

- In this organisation, ALU first operand is always required in the register, and second operand is present either in the register or in the memory.

- After the data processing result will be place in SRC L location.



* Both data-transfer and ALU-operations are 2-Addr Instⁿ.



Inst^n Add r_0, [2000] ; MEM request

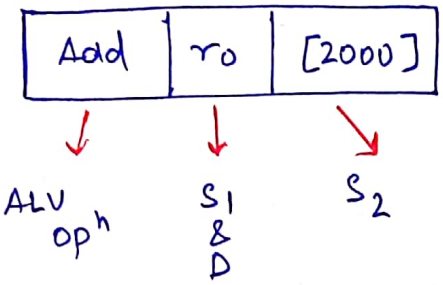
↓

Add r_0, [2000] ; gen reg CPU inst^n

PC : seq Address

} Fc

↓



($r_0 \leftarrow r_0 + M[2000]$)

} Ec

Eg:- $(A * B) + C$ variable are in main memory

Code:-

I_1 : MOV r_0, A ; $r_0 \leftarrow M[A]$

I_2 : MUL r_0, B ; $r_0 \leftarrow r_0 * M[B]$

I_3 : ADD r_0, C ; $r_0 \leftarrow r_0 + M[C]$

Eg:- $(A \overset{+}{*} B) * (C+D)$ variables are in main mem.

Code:-

I_1 : MOV r_0, A ; $r_0 \leftarrow M[A]$

I_2 : ADD r_0, B ; $r_0 \leftarrow r_0 + M[B]$

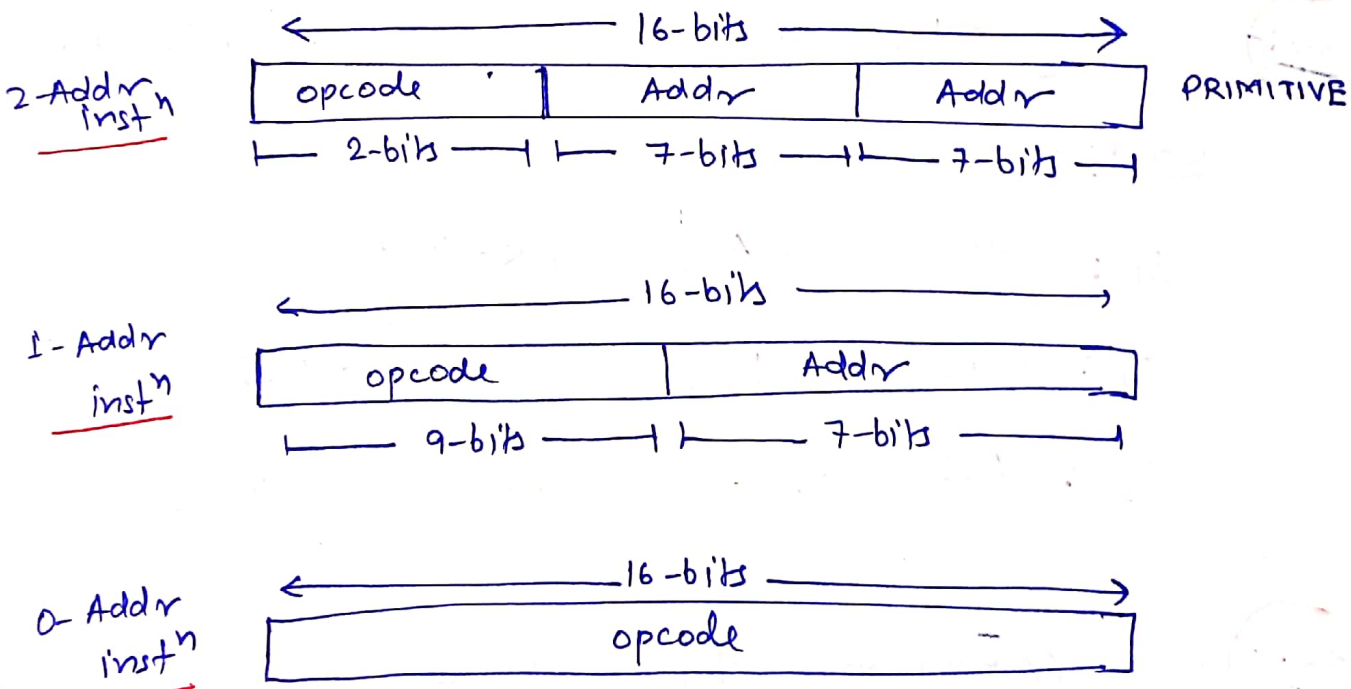
I_3 : MOV r_1, C; $\quad r_1 \leftarrow M[C]$

I_4 : ADD r_1, D; $\quad r_1 \leftarrow r_1 + M[D]$

I_5 : MUL r_0, r_1; $\quad r_0 \leftarrow r_0 * r_1$

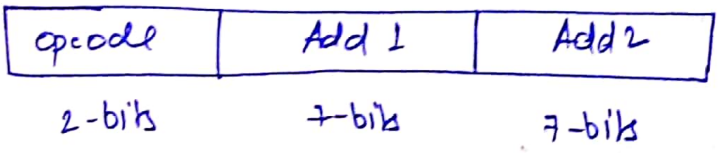
I_6 : MOV X, r_0; $\quad M[X] \leftarrow r_0$

Q. consider a hypothetical CPU which supports 16-bit inst^n placed in 128 W memory, If there exist three 2-Addr Inst^n and 120 1-Addr inst^n, then how many 0-Addr Inst^n are formulated.



Apply Expand opcode techniques :-

1.

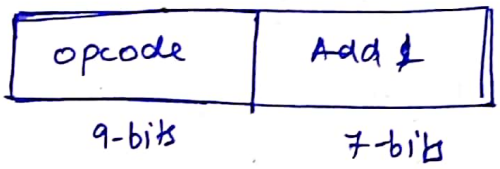


2. operation possible $= 2^n = 2^2 = 4$

3. Free opcodes $= 4 - 3 = 1$

after 2-Addr Instⁿ

4.



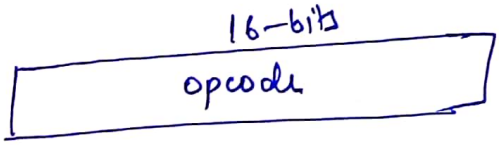
Total number of opcodes of 1-Addr Instⁿ $= 1 \times 2^7$

$= 128$

5. Free opcodes after 1-Addr instⁿ $= 128 - 120$

$= 8$

6.



Total no. of 0-addr Instⁿ $= 8 \times 2^7$

$= 2^{10} = 1024$

Inst^n Set:-

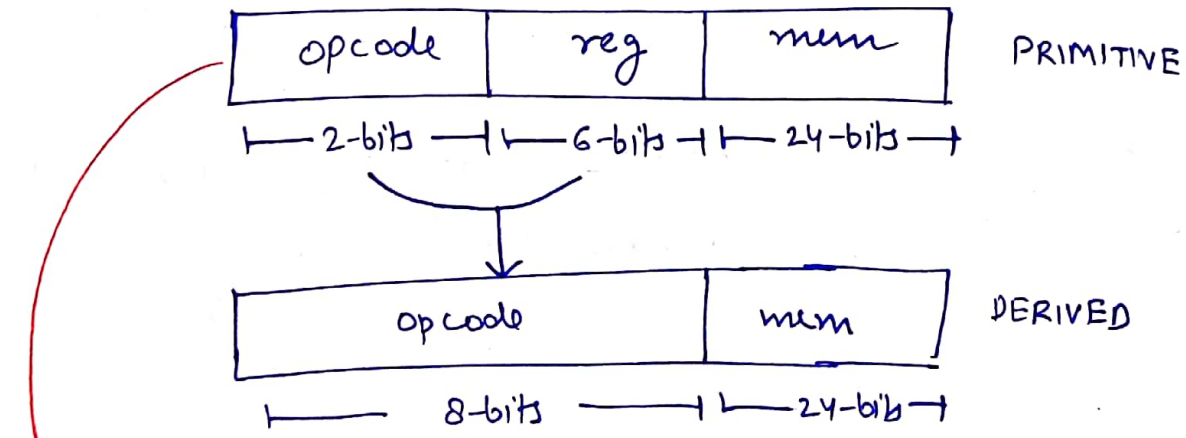
2 - Addr Inst^n : 3 } Given

1 - Addr Inst^n : 120 }

0 - Addr Inst^n : 1024 — Calculated

Q. Consider a hypothetical computer which supports 32-bit instruction. Register file size is 64. System supports 16 MB memory space. If there exist a two 2-Addr Inst^n which uses both register & memory reference then how many 1-Addr Inst^n are possible.

mem-ref ~~memory ref~~



possible opcodes

$2^2 = 4$

00 } used

01

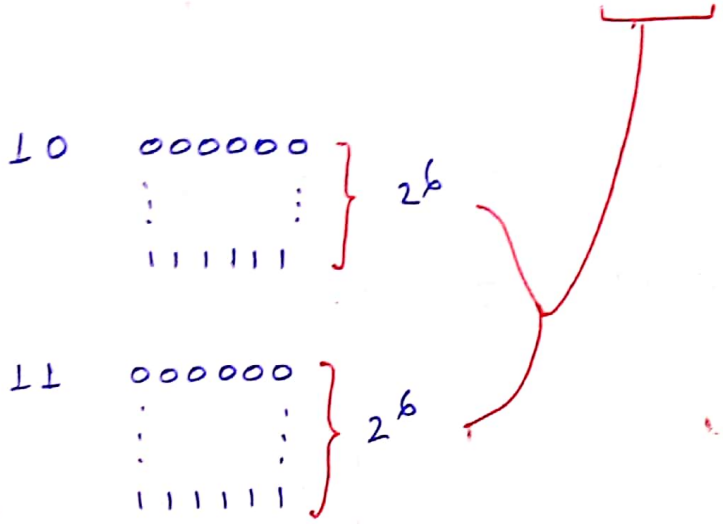
10 } Free

11

Free opcodes after 2-Addr inst^n = 4 - 2 = 2

Now merge reg field bits -

No. of 1-Addr instruction = 2×2^6 opcodes



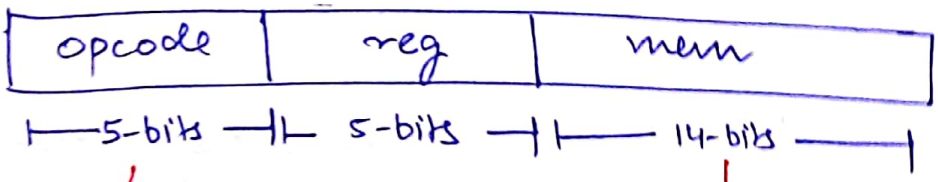
Inst^n set :-

2-Addr Inst^n : 2

1-Addr Inst^n with MEM refrence : 2×2^6

Q. A hypothetical CPU which supports 24-bit inst^n, reg file size is 32. System support 16KB mem space. If there exist 'x' - 2 addr inst^n which uses both reg & mem refrence and 'y' - 1-addr reg ref instruction, then how many 0-addr inst^n are possible formulated.

2-Addr inst^n

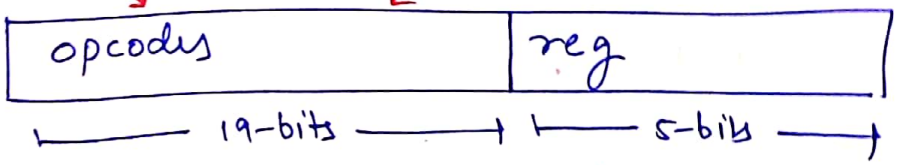


total possible opcodes = $2^5 = 32$

Free opcodes = $32 - x$

1-Addr inst^n

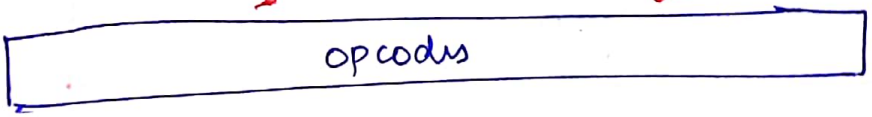
REG reference



possible opcodes = $(32-x) \times 2^{14}$

Free opcodes = $(32-x)2^{14} - y$

0-Addr inst^n



(possible opcodes = $[(32-x)2^{14} - y]2^5$)

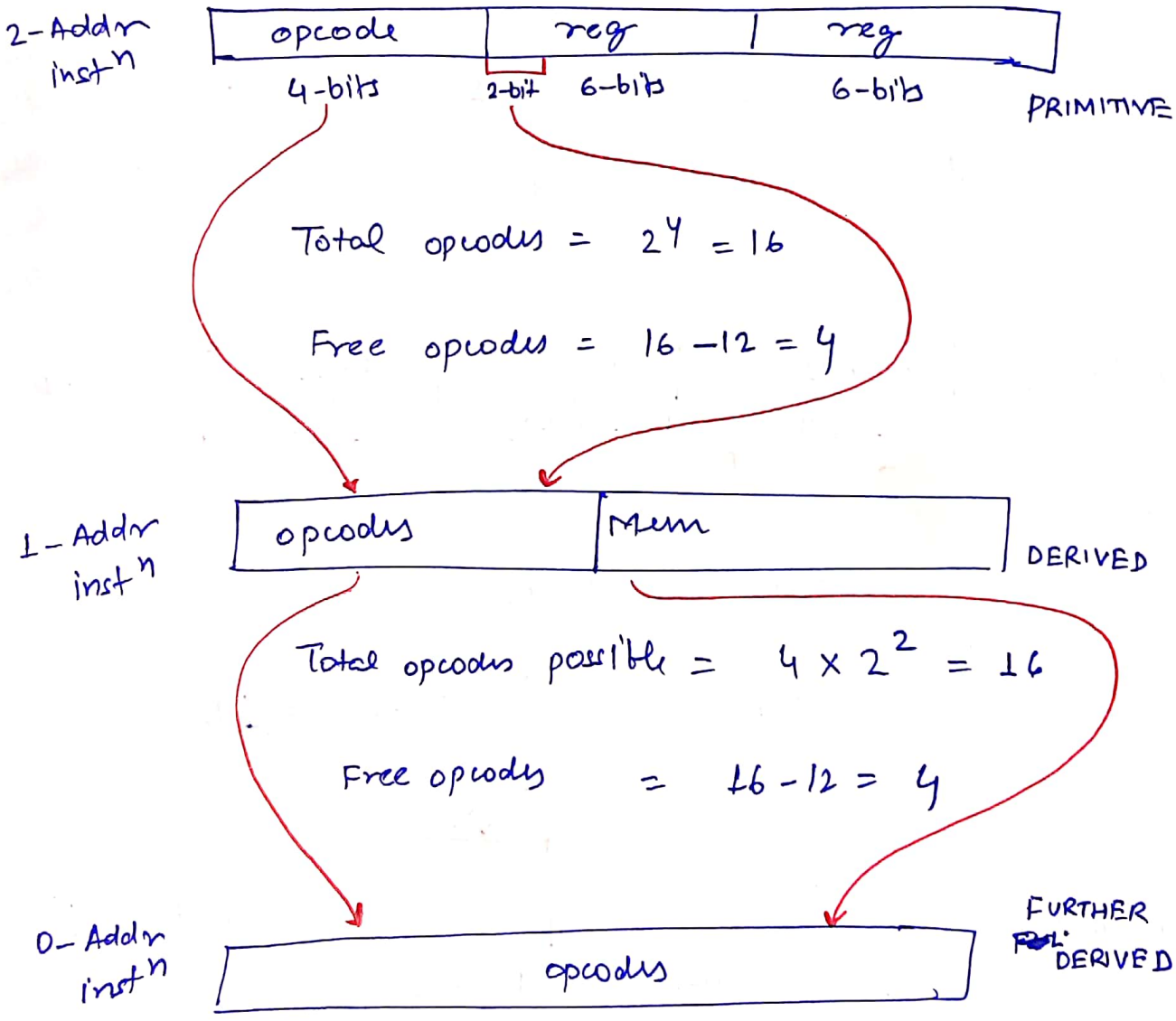
Instruction set :-

2-Addr inst^n : x

1-Addr inst^n : y

0-Addr inst^n : $[(32-x)2^{14} - y]2^5$

Q. Consider a hypothetical CPU which support 16-bit instruction, 64 registers and 1KB memory space. If there exist 12 - 2 add Instⁿ which uses register rehrence and 12 - 1 add Instⁿ with memory rehrence then how many 0-address Instⁿ are formulated.



Total opcodes possible = $4 \times 2^{10} = 2^{12}$

Q. Consider a hypothetical CPU which support 16-bit instructions, 16 reg and 8KB memory space.

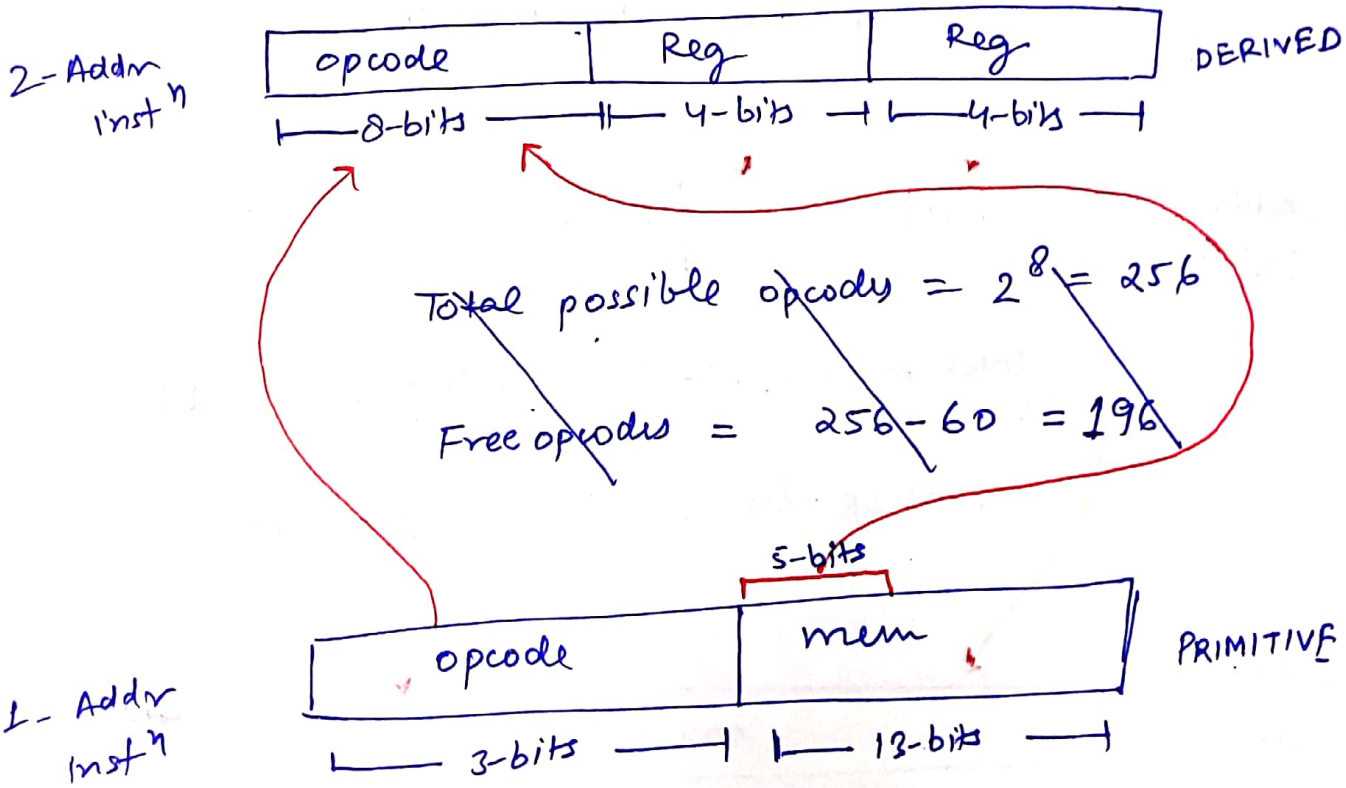
CPU supports three types of instructions i.e

Type 1: 2 reg operand,

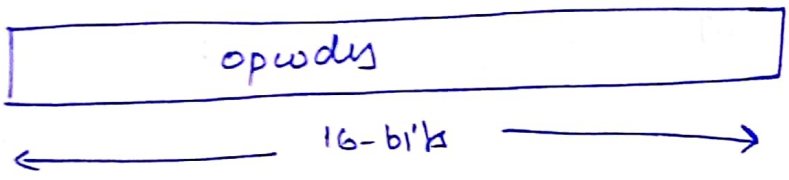
Type 2: 1 mem operand

Type 3: 0 operands

If there exist 60 Type 1 inst^n and 4 Type 2 inst^n then how many Type 3 inst^n are formulated.



0- Addr Instⁿ



FURTHER DERIVED

so, possible opcodes of Type-2 = $2^3 = 8$

Free opcodes in Type-2 = $8 - 4 = 4$

possible Type-1 opcodes = $4 \times 2^5 = 2^7$

Free opcodes = $2^7 - 60 = 68$

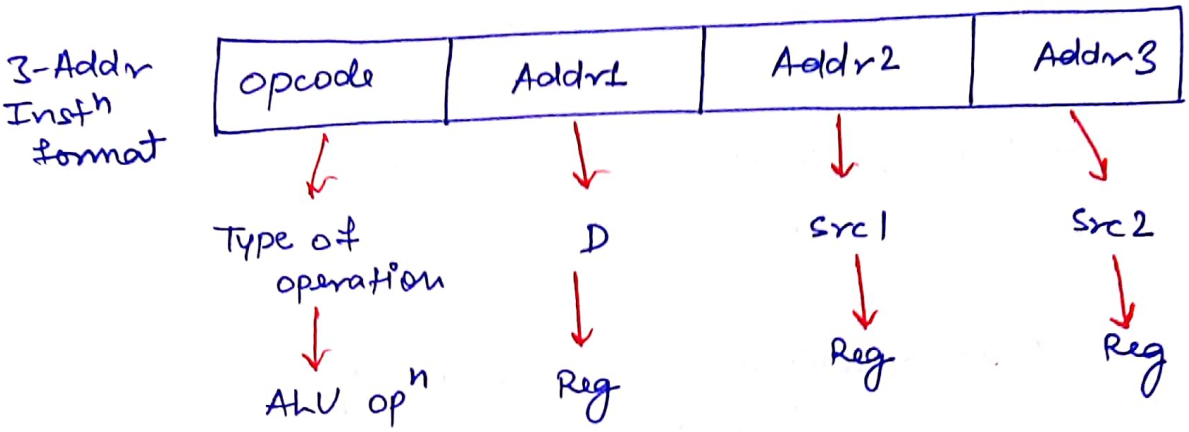
possible Type-3 opcodes = 68×2^8

Register - to - Register Refrence :-

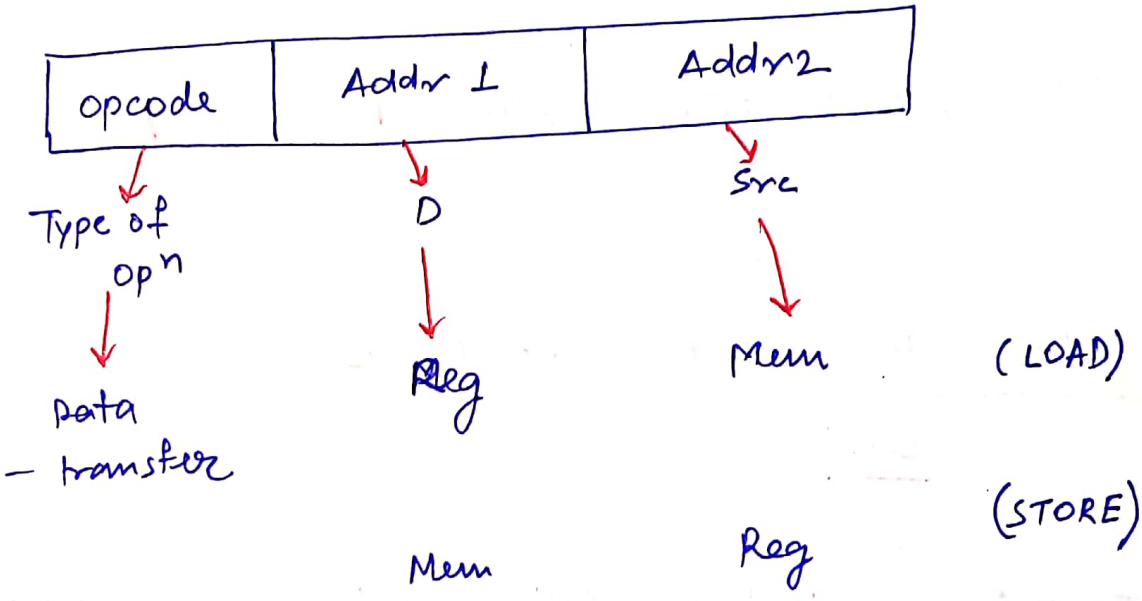
CPU

- In this organization ALU operations are performed only on a register data so both of the operands are always required in the register.
- After the data processing result is also placed into a register.

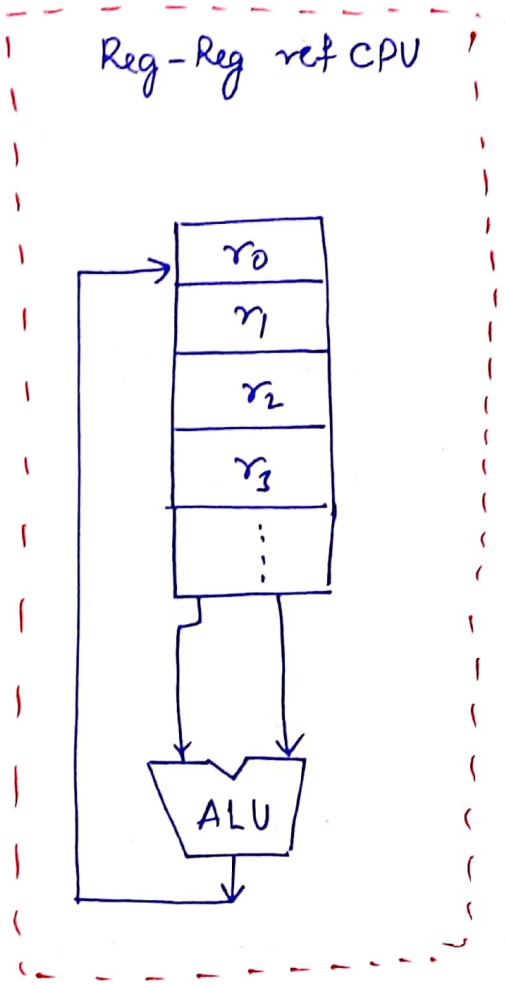
Compatible Instⁿ format :



- In this organisation Data Transfer instⁿ are designed as 2-Addr Instⁿ. LOAD & STORE instⁿ are used as a data - transfer instruction in this CPU.



- Register file is large as more no. of registers are required.

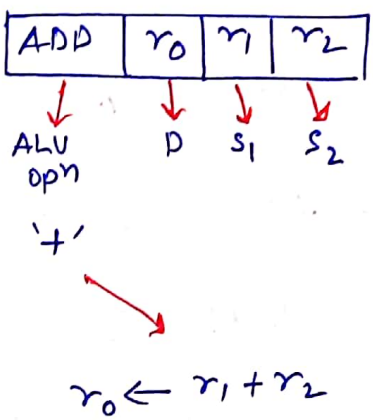


Inst Add r_0, r_1, r_2; mem request

Add r_0, r_1, r_2, GR, CPU

PC: seq

FC



EC

Eg:- $(A * B) + C$ variables are in main memory

Code:-

I_1 : LOAD r_0, A $\quad r_0 \leftarrow M[A]$

I_2 : LOAD r_1, B $\quad r_1 \leftarrow M[B]$

I_3 : MUL r_2, r_0, r_1 $\quad r_2 \leftarrow r_0 * r_1$

I_4 : LOAD r_3, C $\quad r_3 \leftarrow M[C]$

I_5 : ADD r_4, r_2, r_3 $\quad r_4 \leftarrow r_2 + r_3$

Eg:- $X = (A+B) * (C+D)$ variables are in the main memory

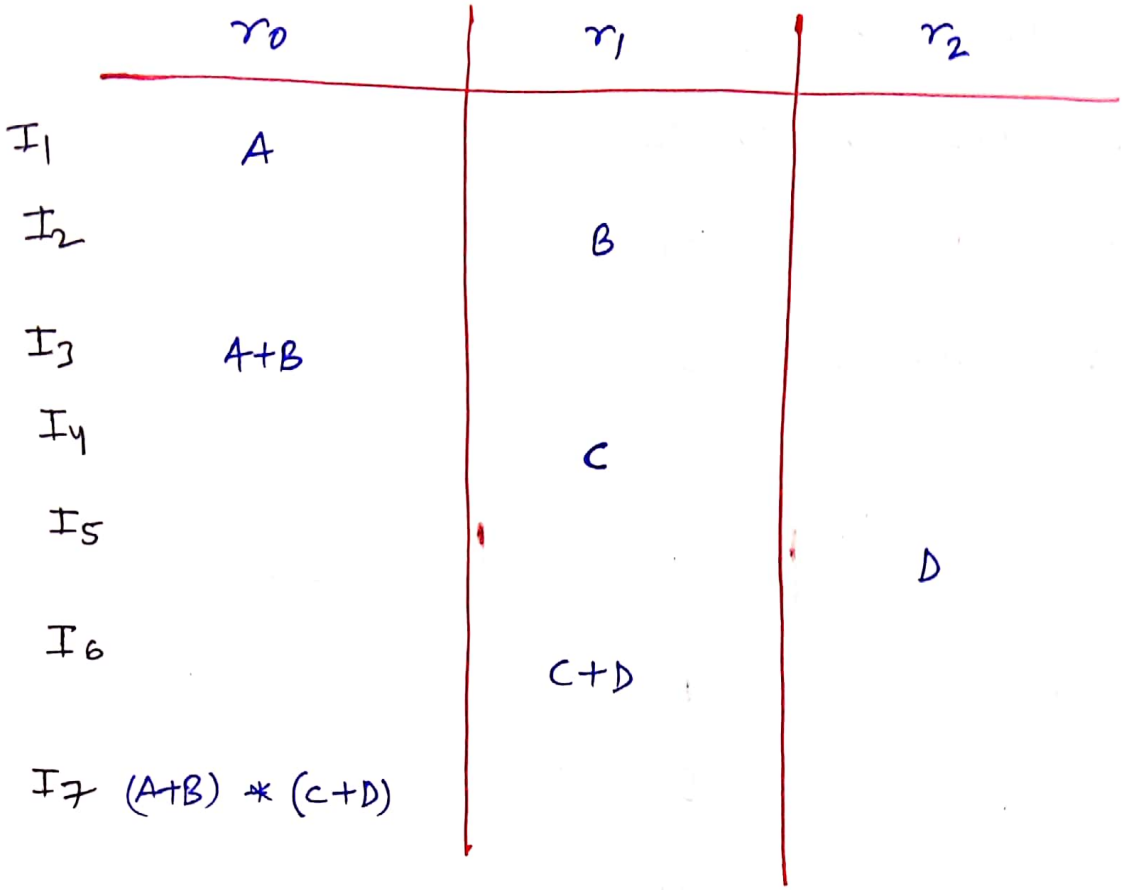
Code :-

I_1 :	LOAD r_0, A	$r_0 \leftarrow M[A]$
I_2 :	LOAD r_1, B	$r_1 \leftarrow M[B]$
I_3 :	ADD r_2, r_0, r_1	$r_2 \leftarrow r_0 + r_1$
I_4 :	LOAD r_3, C	$r_3 \leftarrow M[C]$
I_5 :	LOAD r_4, D	$r_4 \leftarrow M[D]$
I_6 :	ADD r_5, r_3, r_4	$r_5 \leftarrow r_3 + r_4$
I_7 :	MUL r_6, r_2, r_5	$r_6 \leftarrow r_2 * r_5$
I_8 :	STORE X, r_6	$M[X] \leftarrow r_6$

NOTE :- To minimize the registers re-usability is used.

I_1 : LOAD r_0, A	I_8 : STORE X, r_0
I_2 : LOAD r_1, B	
I_3 : ADD r_0, r_0, r_1	
I_4 : LOAD r_1, C	
I_5 : LOAD r_2, D	
I_6 : ADD r_1, r_1, r_2	
I_7 : MUL r_0, r_0, r_1	

So, only 3 reg are used



Q. Consider the following statement executed on a hypothetical CPU. CPU accepting the ALU data only from the register and also produces the result in the register. How many minimum register are required to evaluate the statement without spilling.

$$X = (A+B) * (A+C)$$

variables are in memory.

I_1 : LOAD r_0, A

I_2 : LOAD r_1, B

I_3 : ADD r_1, r_0, r_1

I_4 : LOAD r_2, C

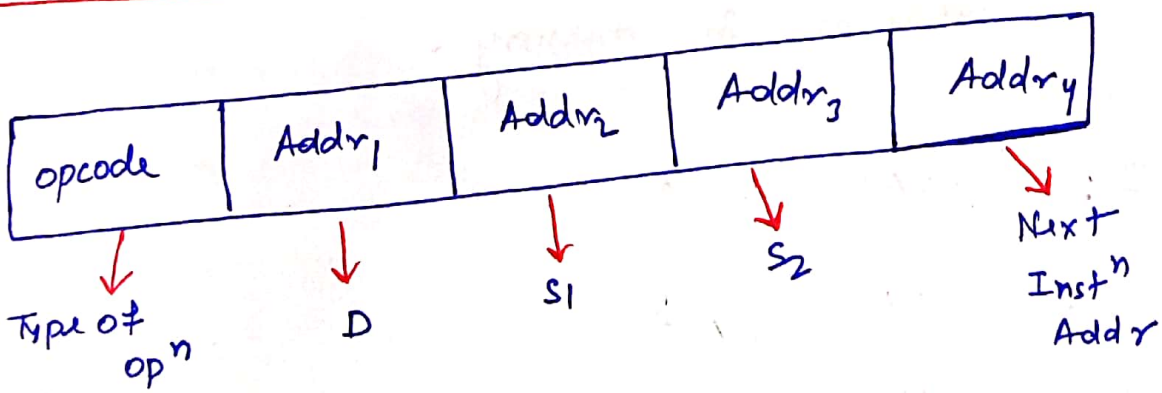
I_5 : ADD r_0, r_0, r_2

I_6 : MUL r_0, r_0, r_1

I_7 : STORE X, r_0

	r_0	r_1	r_2
I_1	A		
I_2		B	
I_3		A+B	
I_4			C
I_5	A+C		
I_6	(A+C) * (A+B)		

4- Addr Instruction Format :-



- PC is a mandatory register in a CPU, used to hold the address of a next instruction during the execution of current instruction. Therefore this concept is not in practise.

Instruction Execution Sequence :-

- Consider the accumulator CPU as a refrence modal to analyze the instruction execution sequence i.e 8085 CPU.

Expression : (A+B)

Variable	Addr	cell
A	2040	EC
B	8060	F2

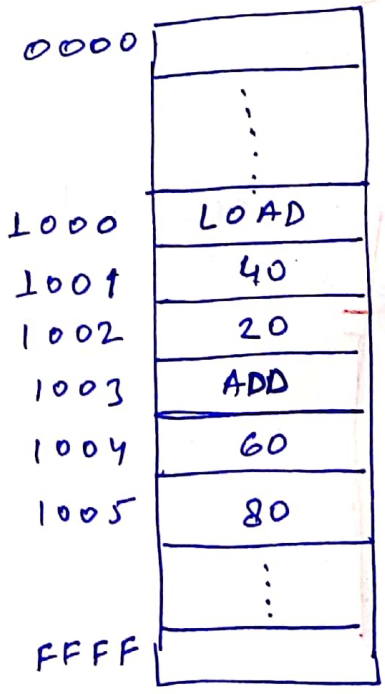
Type of CPU : ACC-CPU { 8085 µP }

Code org 1000;

I_1 : LOAD A

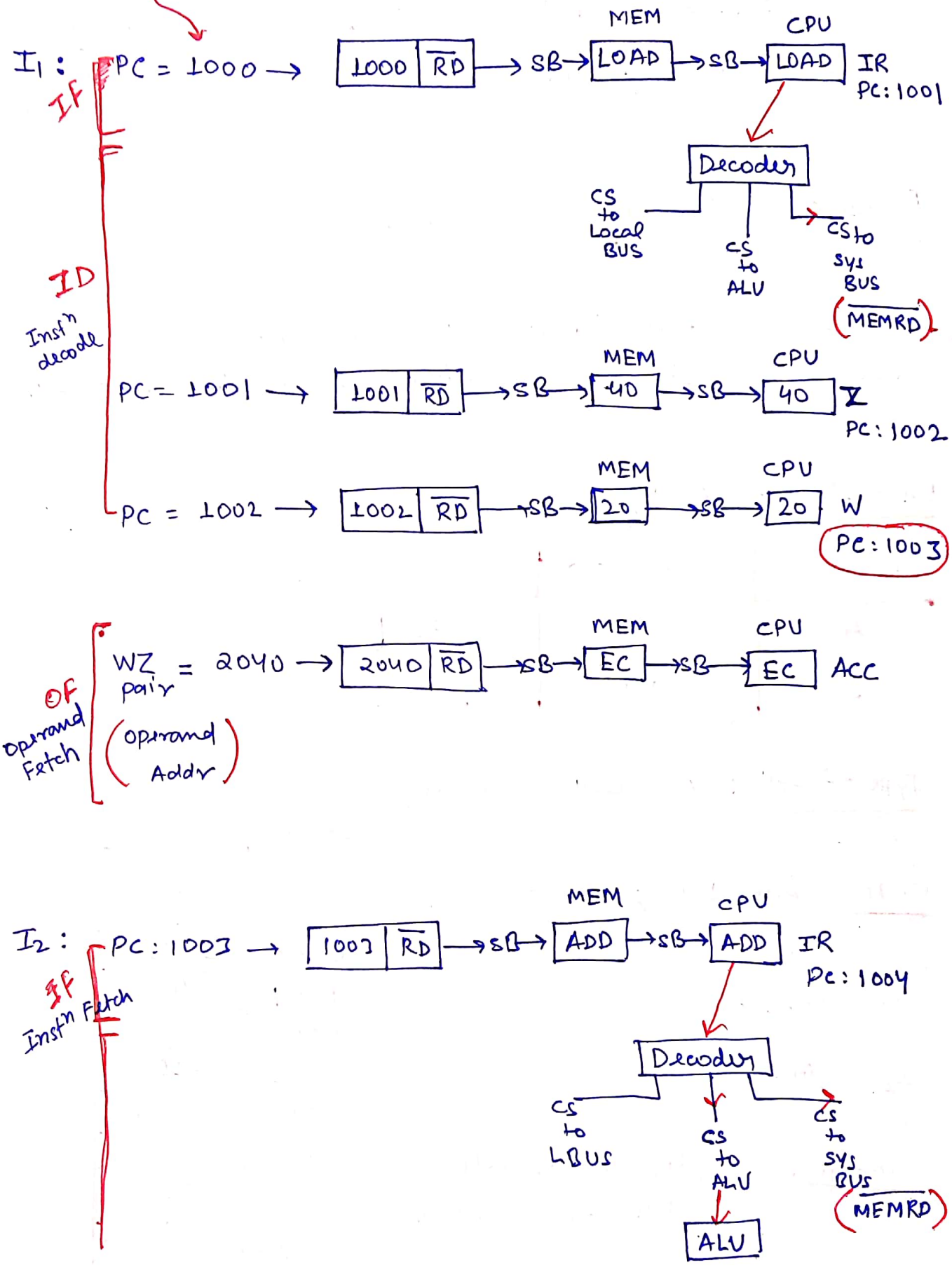
I_2 : ADD B

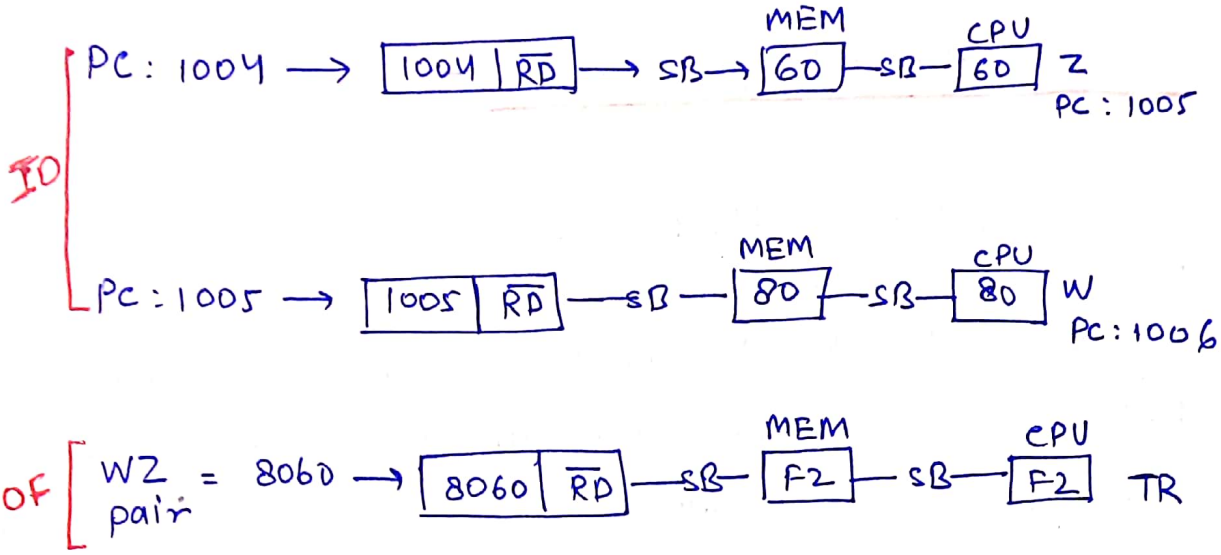
storage : Byte-Addressable with Little-Endian



Execution :-

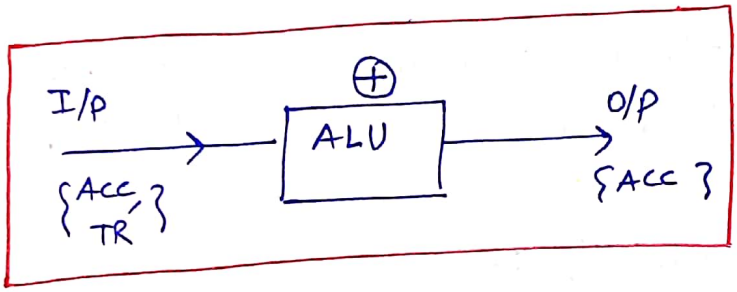
- t : 1000





PD & WB (write-back)

Processing



Instⁿ cycle :-

	FC	EC				
	IF	ID	OF		PD	WB
			S_1	S_2		D
I_1	1MR	2MR	1MR			1RR
I_2	1MR	2MR	1RR	1MR	1ALU	1RR

memory reference

register reference

- Execution (Instruction) states are -

(i) Instruction Fetch (IF) :-

In this CPU fetch the instⁿ from the memory on the basis of PC. At the end of this process PC will be increamented sequentially.

(ii) Instruction Decode (ID) :-

In this state, CPU decodes the opcode to perform the opⁿ and also fetch the remaining part of instⁿ when the instⁿ size is greater than word size.

(iii) Operand Fetch (OF) :-

In this state CPU fetch the data based on the addressing mode.

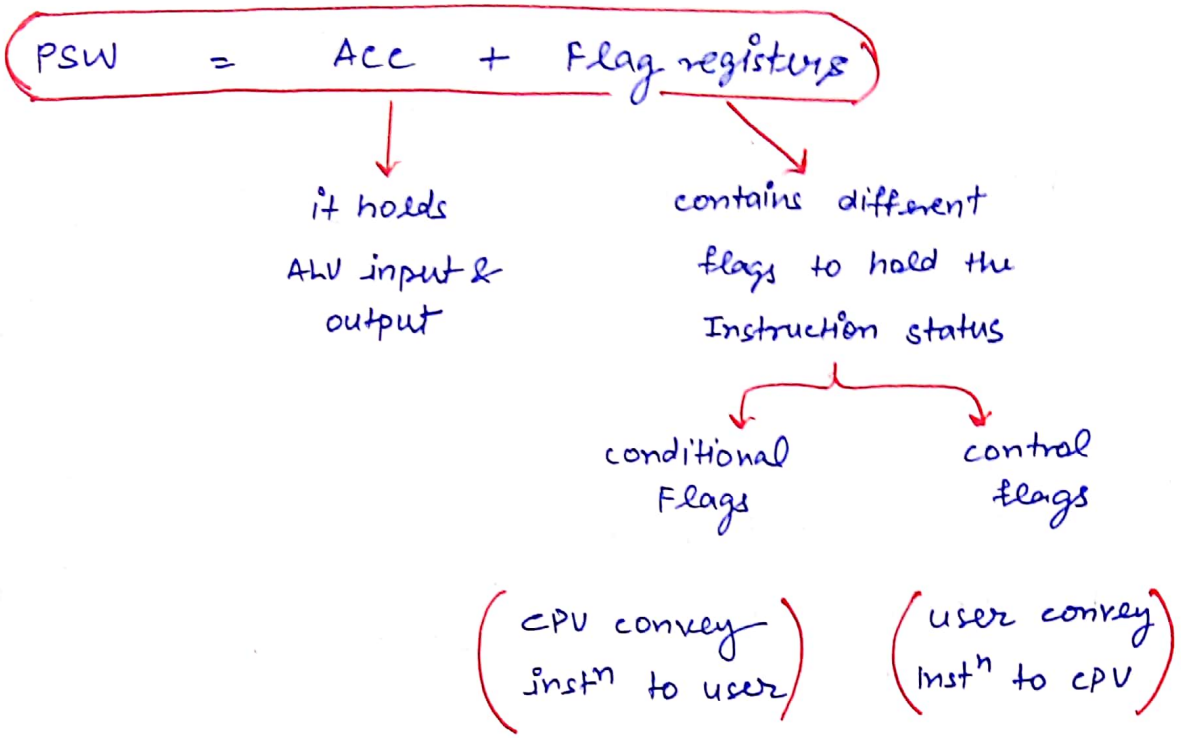
(iv) Processing data (PD) :-

In this state input data is processed on enabled hardware.

(v) Write Back (WB) :-

In this state result will be stored into distination using the addressing modes.

Program Status Word (PSW) :-



Conditional Flags :-

- These flags are SET or RESET based on the result nature of ALU
- According to 8086 MP flag register design, 6 conditional flags are present name as:

1. Carry Flag
2. Parity Flag
3. Auxilliary carry Flag
4. Zero Flag
5. Sign Flag
6. Overflow flag

1. Carry :- 'Is there any extra bit out of MSB'

T set=1 = C

F reset=0 = NC

2. Parity :- 'Is the ALU o/p (Acc) contains even no. of 1's'

T = set=1 = PE (Even parity)

F = reset=0 = PO (odd parity)

This Flag is used in the serial data communication, to perform error correction & detection.

3. Auxilliary Carry :- 'Is there an extra bit from the lower nibble (3rd position) to higher nibble (4th position)

T = set=1 = AC

F = reset=0 = NA

This flag is used in the BCD arthmatic, to hold the range exceeding condition of a lower nibble i.e this flag is set when lower nibble is greater than F.(1111).

NOTE - Carry Flag is also used in BCD arthmatic to hold the range exceeding condition of higher nibble. This flag is set when higher nibble is greater than F (1111).

- BCD format is used to represent decimal data. It is of two types:

  ↳ packed BCD

  ↳ unpacked BCD

- In the unpacked BCD format each digit is represented with 8-bits i.e

0000 0000 ... 0000 1001	Decimal digits
0000 1010 ... 1111 1111	246 codes are unused

In this format more patterns are unused so alternative required i.e packed BCD.

- In the packed BCD format each digit is represented with 4 digits (nibble).

0000 0000 ... 0000 1001	Decimal codes
0000 1010 ... 0000 1111	6 unused codes

- In this format 6 codes are unused (A-F) so adjustment required to convert the invalid codes into valid i.e Add 6 to nibble when nibble is greater than 9.

Eg:-

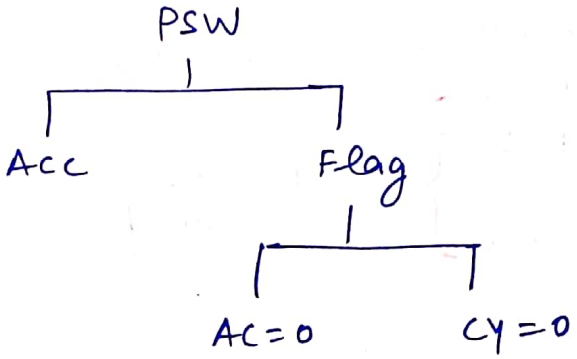
$$\begin{array}{r} 27 \\ +36 \\ \hline 63 \\ \hline \end{array}$$

$$\begin{array}{r} \text{(HN)}\quad\text{(LN)} \\ 0010\quad 0111 \\ +\ 0011\quad 0110 \\ \hline 0101\quad 1101\ \ (5D) \\ \hline \end{array}$$

{ 13 ~~X~~ F AC = ~~0~~ 1
  6 < F CY = 0 }

{ AC = 0
  CY = 0 }

Justify :-



[D > 9 and AC = 0 , actual LN is 'D' only
Add '6' to LN

5 < 9 and CY = 0 , Actual HN is '5' only
No adjustment]

So,

$$\begin{array}{lll} \text{~~5c~~} & (5D) & \text{~~0000~~} \\ +\text{~~c6~~} & & 0101\quad 1101 \\ \text{~~1011~~} & (+6) + & \qquad\quad 0110 \\ \hline & (63) & 0110\quad 0011 \\ \hline \end{array}$$

Eg:-

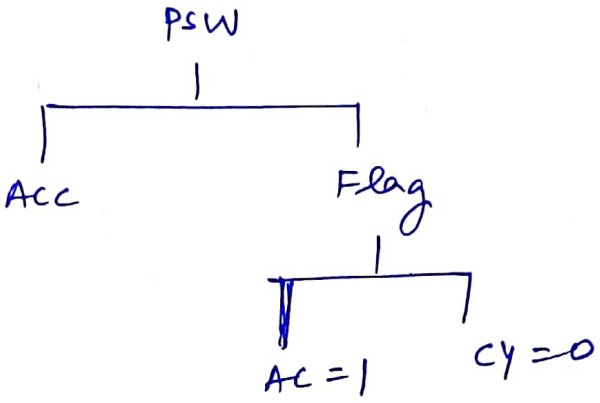
$$\begin{array}{r} 28 \\ +\ 39 \\ \hline 67 \end{array}$$

HN LN

$$\begin{array}{r} 0010\ \ 1000 \\ +\ 0011\ \ 1001 \\ \hline 0110\ \ 0001 \end{array}$$ (61)

(AC = 1, CY = 0)

Justify:-



[1 < 9 and AC = 1 so actual LN is > F
Add 6 to LN

6 < 9 and CY = 0 so actual HN is 6
No Adjustment]

$$\begin{array}{r} 61 \\ +6 \\ \hline 67 \end{array}$$

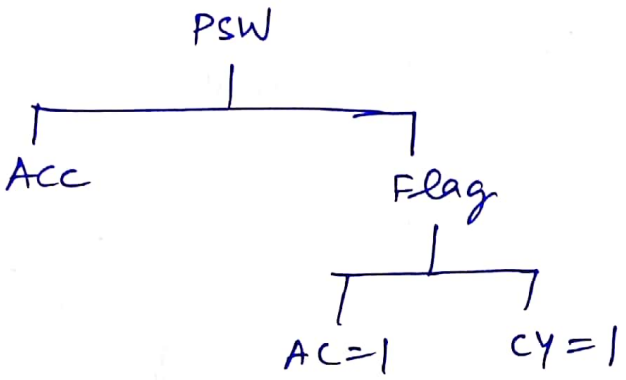
Eg:-

		HN	LN
		1	
89		1000	1001
99		1001	1001
188		0010	0010 (22)

AC = 1

CY = 1

Justify



2 < 9 & AC = 1 so actual LN > F

Add 6 to LN

2 < 9 & CY = 1 so actual HN > F

Add 6 to HN

22
66
88

CY = 1

188

4. Zero Flag :- "Is the ALU o/p (Acc) value is zero"

- T = set = 1 = Z
- F = reset = 0 = NZ

Acc_value	flag_status
0	1
≠ 0	0

eg1

```
MOV CX, #1000
LL: DEC CX
    JNZ LL
```

5. Sign Flag :- 'Is the MSB bit of ALU o/p (Acc) is 1'

- T = set = 1 = NG (-ve logic)
- F = reset = 0 = PL (+ve logic)

6. Overflow flag :-

$$OV(x,y,z) = x'y'z + xyz'$$

$x'y'z$ → +ve OV

xyz' → -ve OV

Q. Consider the following 2's complement of data and perform the addition operation, What is the status of sign, zero, carry, Parity, overflow flag after the computation.

D_1 : 1011 0110

D_2 : 1100 0101

carry

① 1011 0110

\+ 1100 0101

0111 1011

CY = 1

OV = 1

Parity = 1 (PE)

AC = 0 (NA)

zero = 0 (NZ)

sign = 0 (PL)

Control Flags :-

- Based on the status of these flags, hardware operations are controlled.
- According to 8086 flag register, 3 control flags are present.
  - TRAP Flag
  - Interrupt Flag
  - Direction Flag

TRAP Flag

- 1, Single step prog execution; -t
- 0, At a time prog execution; -g

Interrupt flag

- 1, Enable the interrupt (EI)
- 0, Disable the interrupt (DI)

(Only maskable interrupt are controlled)

Direction flag

- 1 Auto decreament, STD (set-direction)
- 0 Auto increment, CTD (clear-direction)

Addressing Modes :-

It shows the location of a required object in the program. Object may be a data or instruction.

- O/p of a addressing mode is effective address (EA)
- EA is the actual address of an object, therefore

$$\text{object} = [EA]$$

(content of)

- Addressing mode is implemented in the inst^n design in two ways :

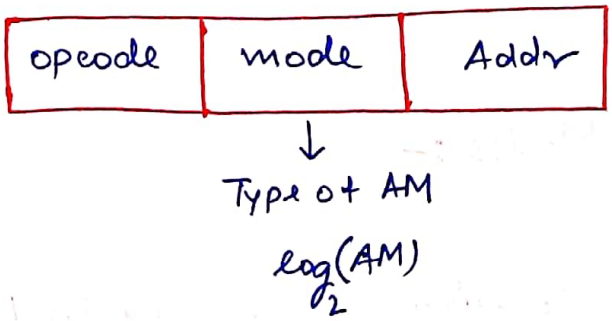
↳ Implicit { opcode itself specify the type of AM used }

eg: ADD (stack - CPU)

~~ADD [xxxx]~~

↳ Explicit { mode field is used in the inst^n to specify type of addressing mode used }

i.e



- Different symbols are used to refer the various addressing modes in the user program.

/ I → Immediate mode

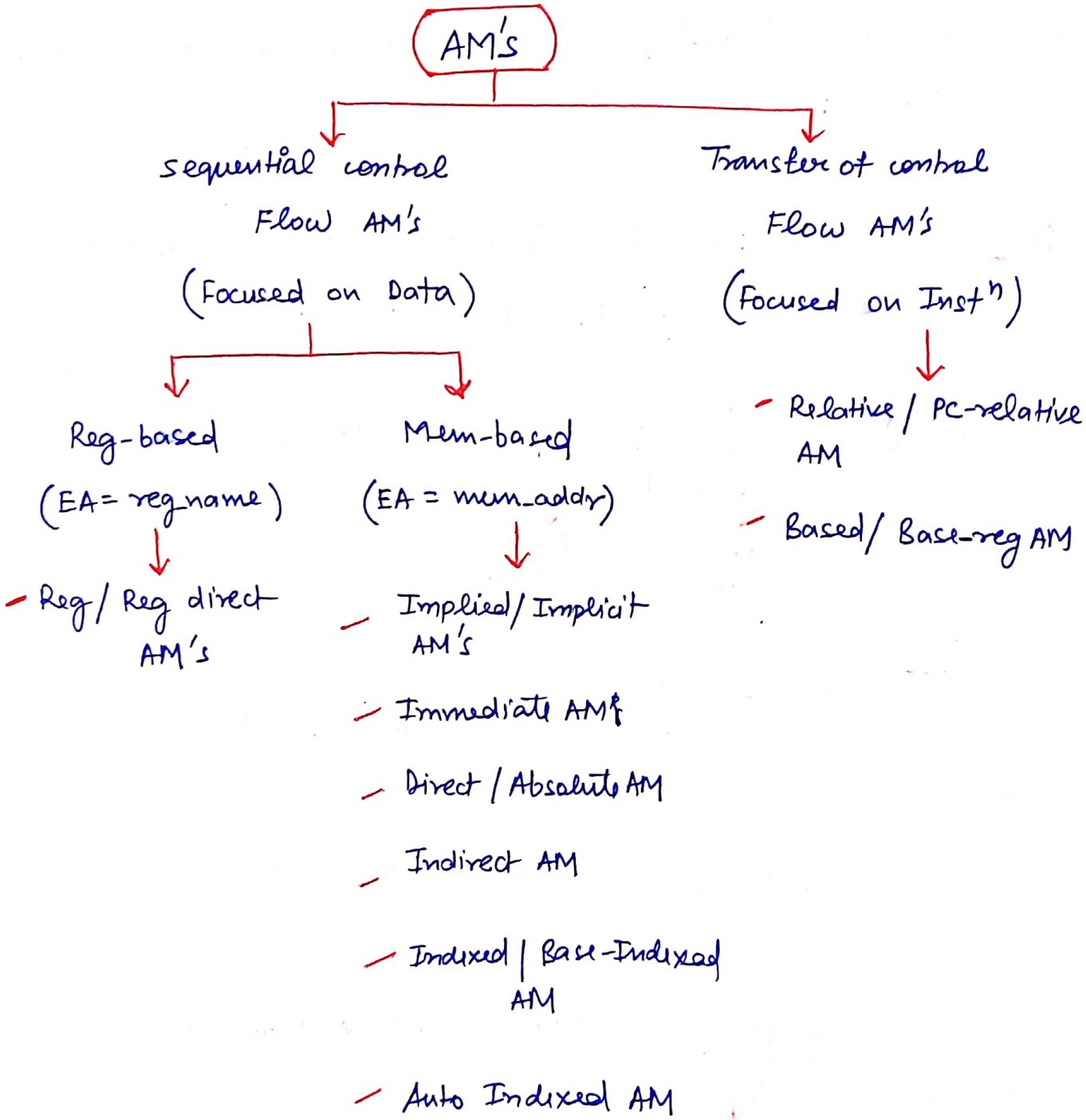
Reg_name → Register AM

@ / ( ) → Indirect AM

Index by name → Indexed AM

+/− → Auto indixed AM

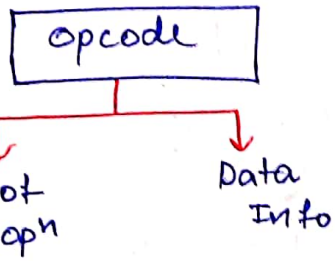
- Different AM's used in the computer design is as follows :



Sequential control flow AM's :-

- These modes concentrate on location of data ∴ data transfer & data manipulation Inst^n are designed with these AM's.
- Different addressing modes (AM's) is deployed in this category :

1. **Implied AM** :- In this mode data into is present in the opcode itself.



0-Addr-Inst^n format

eg-

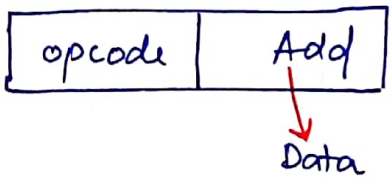
STC	set carry CY=1
CLC	clear carry CY=0
ADD	stack - CPU ALU addition

ADD → POP, POP, +, PUSH

TOS ← TOS + TOS

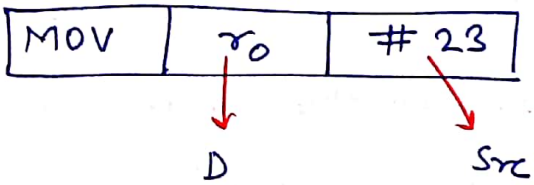
2. Immediate AM :- This mode is used to access the constants. In this mode data is present in the address field of an inst^n.

Inst^n design



Data = Addr field value

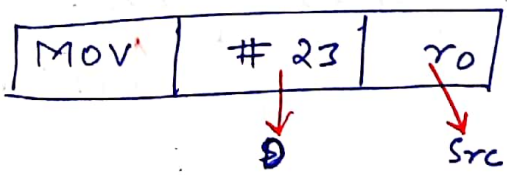
eg:-



✓ VALID

($r_0 \leftarrow 23$)

eg:-



✗ INVALID

$23 \leftarrow r_0$

constant value can't be an address or location

* Immediate value is only allowed in source field

Limitation :-

Range of constants are restricted by the size of an address field i.e n-bit address field support

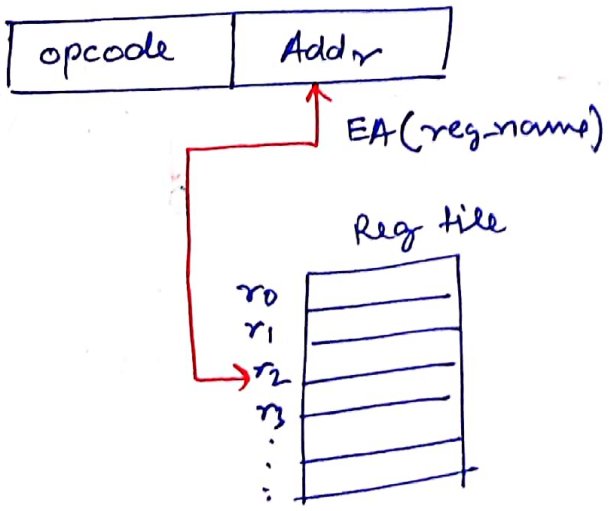
$(0 \text{ to } 2^n - 1)$ unsigned constants

$(-2^{n-1} \text{ to } +2^{n-1} - 1)$ signed constants

It is a source addressing mode, it can't be used as a destination addressing mode.

3. Register AM :- This mode is used to access the local variables. In this mode data is present in the register, the corresponding register name will be maintained in address field of an instruction.

Inst^n design :



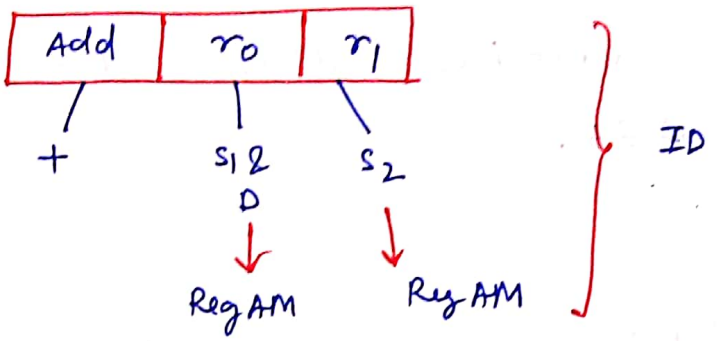
EA = Addr field value

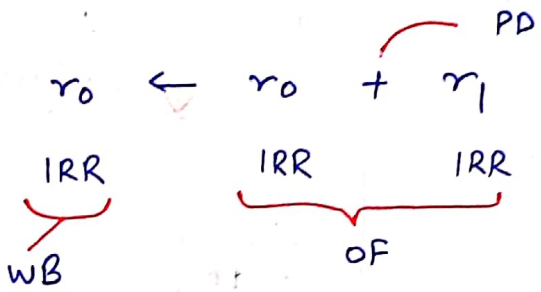
↓

Reg_name

Data = [EA] = [Reg_name]

eg:- Assume 4W Inst^{h} given

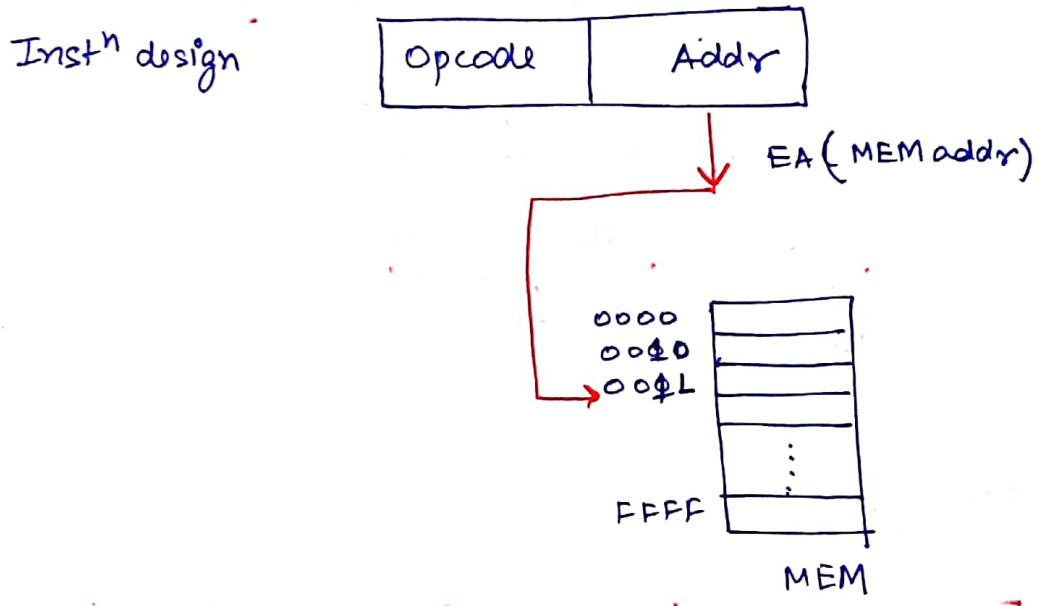




Inst^{h} cycle :-

FC		EC		
IF	ID	OF	PD	WB
		S_1 S_2		D
IMR	3MR	IRR IRR	IALU	IRR

4. Direct Addressing :- This mode is used to access the static variables. In this mode data is present in the memory, the corresponding memory address will be maintained address field of an instruction as an EA.



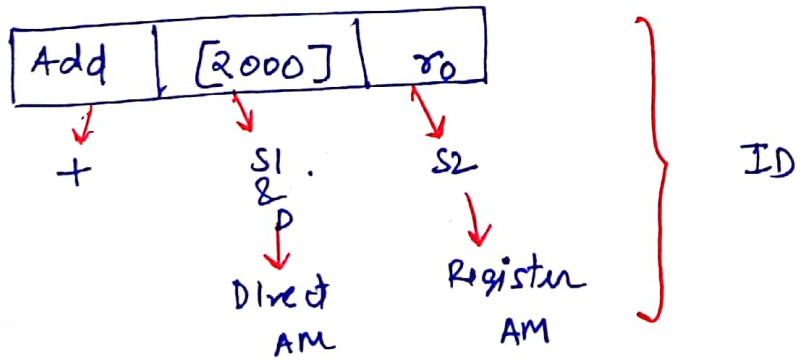
EA = Addr filed value

↓

MEM addr

(Data = [EA] = [MEM addr])

Eg:-



PD

M[2000] ← M[2000] + r_0

(IMR) (IMR) (IRR)

WB OF

Inst^n cycle :-

FC	EC				
IF	ID	OF		PD	WB
		S_1	S_2		D
IMR	2MR	IMR	IRR	ALU	IMR

5. Indirect Addressing :- This mode is used to access the pointers. In this mode address field contains the address of EA.

[MEM - indirect AM]

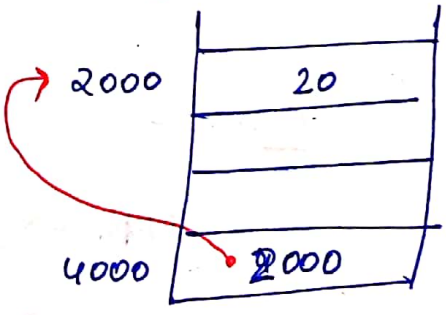
Eg: MOV r_0, @4000

(IMR) r_0 ← M[[4000]]

(IMR) r_0 ← M[2000]

r_0 ← 20

(2MR)



MOV r_1, #2000 [reg- indirect AM]

MOV r_0, @r_1

(IRR) $r_0 \leftarrow M[[r_1]]$

(IMR) $r_0 \leftarrow M[2000]$

* so this is enhanced version of Indirect Addressing

$r_0 = 20$

(IRR + IMR)

- Mem -indirect AM , EA is in the memory : Address field of Inst^n contain address of EA.

EA = [Addr field value]

Mem Addr

Data = [EA]

[[mem-addr]]

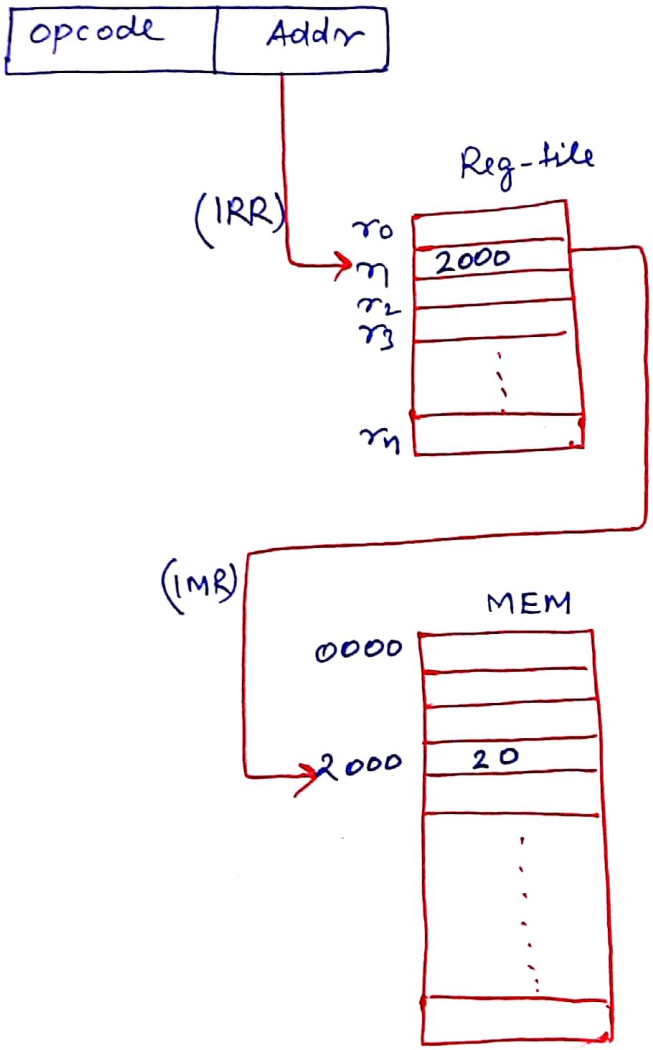
Actions -

IMR → to get EA

IMR → to access the Data

NOTE: This consumes more time to access the data so, enhancement required i.e reg-indirect AM.

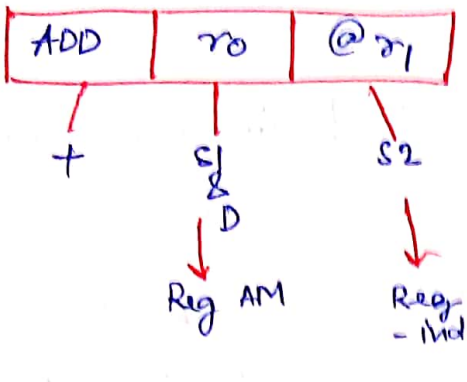
- Reg-indirect AM :- In this mode EA is in the register. In this mode, address Field of an Instn contain address of a effective address i.e register name has the EA.



Actions:

IRR — to access EA

IMR — to access Data



[reg_name]

this means reg contains some special info

$r_0 \leftarrow r_0 + M[[r_1]]$

IRR IRR IRR + IMR