Number Representation

1. Number System.
2. Conversion
3. Complement (r's, r-1's)

Numbers used → 1. Fixed Point
→ 2. Floating Point
} Number - Representation

→ 23.24, etc. 0101.110, etc.

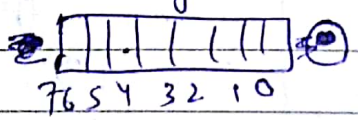
- **Fixed point**: fixed radix/decimal point. {Integers or without radix}.
  Has no decimal point.
  eg: 43, 2, 31, etc, 1010, etc.

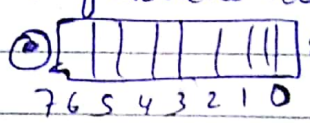
Stores:- 1.) Number → (use registers as store binary).
→ [Comb^n of flip flops].
eg: 8 bit register

(when store a value)
2.) Sign of number
3.) Decimal points (Radix).

→ use 0 & 1 also. If [0 ⇒ positive number.
1 ⇒ negative number.]

→ (a) **Fixed point**:- eg: 8 bit reg. Assume decimal point is to
Dec. point is fixed. extreme right or extreme left. (fixed).
⇒ integer no. ⇒ fractional no (only assume not store fractional here).

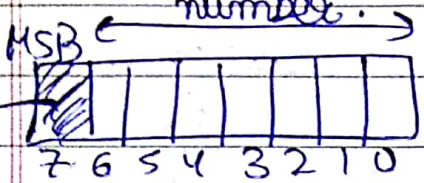




- [Only assumption, not actual store decimal point.]

→ (b) **Floating point**:-

→ On next page (2nd next)



0000

+2 ⇒ 0010

-2 ⇒ 1000 0010

+2 ⇒ 0010

-2 ⇒ 1010

0 ⇒ 0000
⇒ 1000
⇒ 1001
⇒ 1010

2 separate represⁿ (It must not be).

lso not gives effective lt in addⁿ/subtrⁿ

3-1 ⇒ 3+(-1) = 2

0011
+1001
1100 = -4

not equal.

t used.)

X

Store 1's complement instead of magnitude of positive one.

+2 ⇒ 00000010

⇓ 1's comp.

-2 ⇒ 11111101

{sign bit always 1 for -ve}

-0 ⇒ 1111
0 ⇒ 0000

2 repsⁿ (Must not be).

1 = 0001
2 = 0010
3 = 0011
4 = 0100
-1 = 1111
-2 = 1101
-3 = 1100

eg: 3+(-1) = 3-1 = 2

0011
+1110
10001 (overflow)

if carry, so add +1

0010 (Ans).

(But less used).

Store 2's complement instead of magnitude of positive one.

+2 ⇒ 00000010

⇓ 2's.

-2 ⇒ 11111110.

0 ⇒ 0000

[only one representation for zero here]

Range :- $[-2^{n-1} \text{ to } 2^{n-1}-1]$

eg: -1 = 1111
-2 = 1110

for 4 bit ⇒ [-8 to +7].

(-8 also to be stored here)

eg: 3+(-1) = 2

0011
+1111
10010 (2). Ans

(discard carry)

(-7 to 7 go only)

use it in m/c (Best)

✱ Decimal Number :- [Number = ±M × r^±e]

exponent.

radix / base.

Mantissa (Significand)

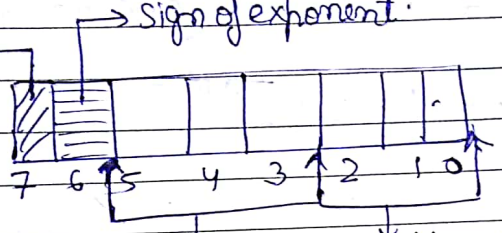
(Floating point repr^n).

(Single point precision) 32 bit → 64 bit (Double point precision)

⊙ Store all 3 (Mantissa, exponent, sign of mantissa, decimal point, e can be -ve or not) what to store or not?

{Not store radix}.

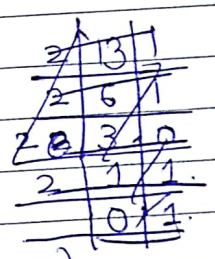
"Representation":



eg :- $(-13.9)_{10}$ ⇒ $-(1101.11100)_2$

⇓

$M \times r^e \Rightarrow M \times 2^e$



X [Denormalized form]

- Decimal point shifted to extreme left.

→ eg: 0.1101111100 ;

{0.M}.

↳ Mantissa. (9 bits here) reqd.

[Normalized form] (Use it)

- Shift decimal point shifted to extreme left + MSB must be 1.

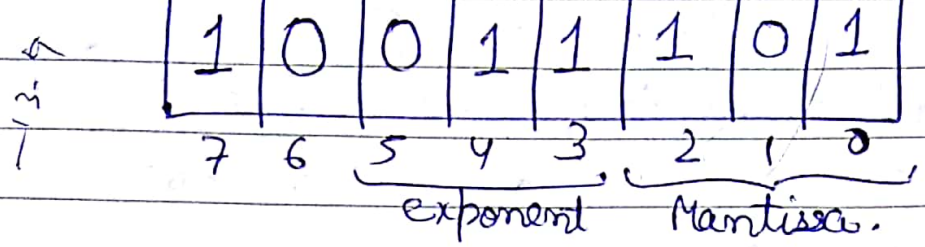
eg : 1.[01 11100] (only it stores)

{1.M} → not stored in register.

↳ Mantissa. (8 bits here) reqd.

Adv : Only store M. not 1.

{Shift that much left only till where get 1 one before point}



Rest are lost.
In 8 bit, store initial 3 bit of M

Number $\Rightarrow 1.M \times 2^{e}$ } 9 bit number stored using 8 bit as "1" not stored

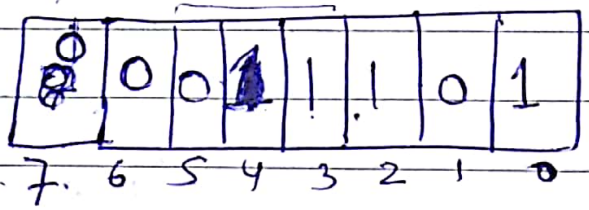
$-1.101... \times 2^{+011}$ (Not accurate).

∴ [• More in exponent bits ⇒ ↑ in range.
• More in mantissa bits ⇒ ↑ in accuracy] #

eg: $13 \Rightarrow (1101)_2 \Rightarrow 1.\underbrace{101}_{M} \times 2^{3} = +1.101 \times 2^{+011}$

[Normalised form].

$\therefore +13 \Rightarrow$



{Here as only 8 bit ⇒ So, -13 same repr^{s} as -13.9 as accuracy less}

Both fixed point & floating point ⇒ Stored / Implemented using floating point representation Only

☆ Biased Exponent:- {bits of exponent & range}

If we remove sign of exponent, we can expand bits of exponent by adding biased number of exponent.

Biased exponent = Biased No. + (±) exponent

1. Biased number = $\lfloor \frac{15}{2} \rfloor = 7$.
2. $2^{n-1}-1 = 2^{4-1}-1 = 8-1 = 7$.
3. Biased exponent = Biased No. + exp

[unbiased exponent → -7 to 8]

eg:- $(-13.9)_{10} = -(1101.11100)_2$

1	1	0	1	0	1	0	1

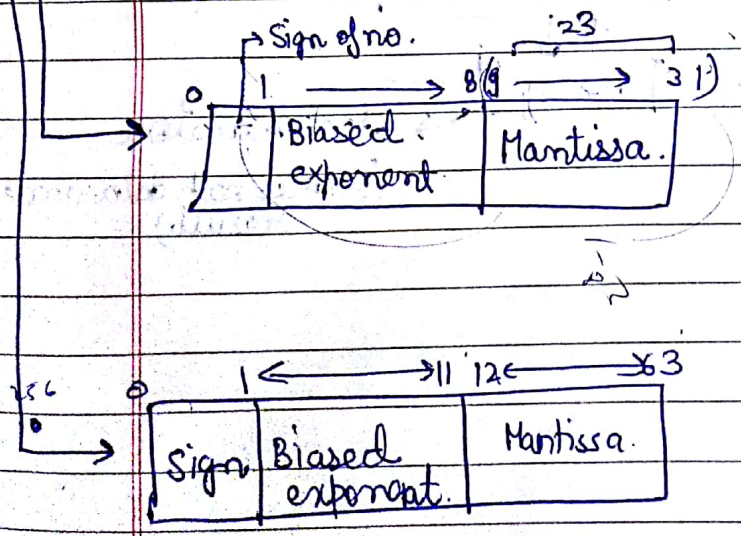
Biased exponent | M

Unbiased exponent = (011) = 3

Biased exponent = 3 + 7 = 10 (1010). Ans

☆ IEEE standards for floating point representation :-

1. Single Point Precision (32 bit)
2. Double Point Precision (64 bit).



Biased number = $\lfloor \frac{255}{2} \rfloor = 127$

[for normalized form.]

For denormalized form:-

→ Biased number = $\lfloor \frac{(256*8)-1}{2} \rfloor = 1023$.

Ans.

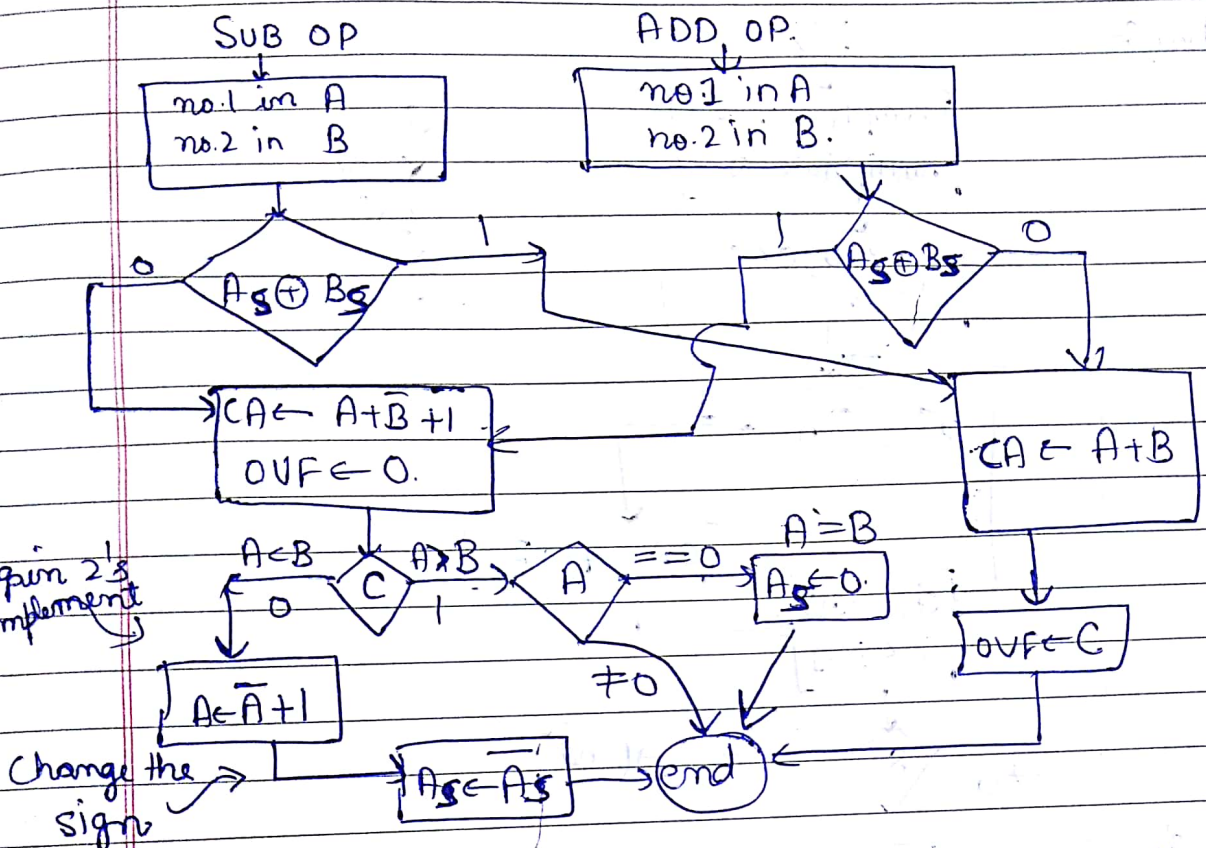
✡ Arithmetic Operations :-

1.) Addition / Subtraction :

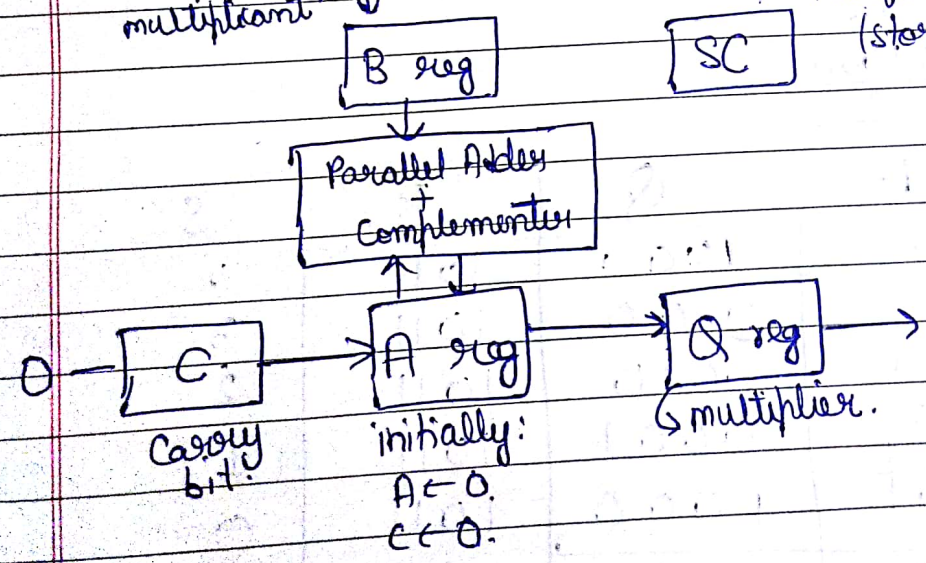
O.P.	Addition	Subtraction		
		A < B	A > B	A = B
(+A) + (+B)	+(A+B)			
(+A) + (-B)		-(B-A)	+(A-B)	0
(-A) + (+B)		+(B-A)	-(A-B)	0
(-A) + (-B)	-(A+B)			
(+A) - (+B)		-(B-A)	+(A-B)	0
(+A) - (-B)	+(A+B)			
(-A) - (+B)	-(A+B)			
(-A) - (-B)		+(B-A)	-(A-B)	0

Hardware Implementation :-

✱ Algorithm :-



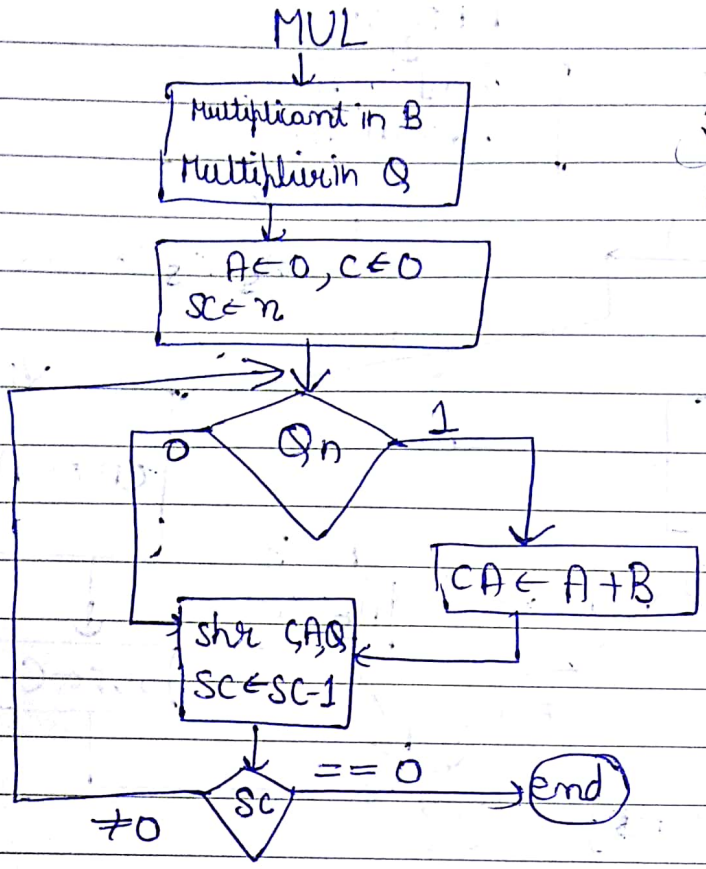
Multiplication :- "Unsigned Numbers Only"



Hardware Implementation

✲ Unsigned Number Multiplication :-

Algorithm :-



0 0 1 1 → shr = shift right (logical)

0 0 0 1

eg: 11 × 8 ⇒ 1011
× 1000
?

C	A	Q	B	SC
0	0000	1000	1011	4
0	0000	0100	1011	3
0	0000	0010	1011	2
0	0000	0001	1011	1
0	1011 / 0101	1000	1011	0

result = 88.

ans.

0011
1100
$+1$
1101

"See on own for floating point oper"

classmate Date ___ Page ___

"Computer Architecture"

★ **Signed numbers multiplication** → (Booth's multiplication algo) Or (2's Complement multiplication)

(Use signed 2's complement representation)

2's comp	N
1101	→ −3
1110	→ −2
1111	→ −1
0000	→ 0
0001	→ 1
0010	→ 2
0011	→ 3

1100

"Properties":-

1.) ⇒ If 4 bits :- last bit (MSB) is 1 for −ve numbers :- (take −ve value)

$-3 = 1101 = 2^0 + 2^2 - 8 = -3$ Ans

2.) $14 = 01110$ ⇒ If contiguous seq of 1's :- seq of 1's in 14 ⇒ 2^3 to 2^1

($2^4\ 2^3\ 2^2\ 2^1\ 2^0$)

so; written as :- 2^k to 2^n (i)

$[N = 2^{k+1} - 2^n]$

$n = 1$ — $k = 3$

eg : 14 : 2^3 ⇐ 2^1 ⇒ $2^4 - 2^1 = 14$ Ans.

3.) If not contiguous 1's or 0's :-

eg : $0111010 = 58$ ($+2^6\ -2^3\ +2^2\ -2^1$ = 58)

For 1st 1 from RHS: subtract, put as −ve

for 1st 0 from RHS: if prev bit = 1, add it.

If continuous 0's or 1's ⇒ not do anything

↳ (a) 1 (in seq of 1's) ⇒ -2^k

(b) 0 (if prev bit is 1) ⇒ $+2^k$

(c) If 1 (prev bit is 1) } ⇒ nothing do
If 0 (prev bit is 0) }

−2 in 8 bits :-

1	0	0	0	0	0	1	0
7	6	5	4	3	2	1	0

Signed magnitude

1	1	1	1	1	1	0	1
7	6	5	4	3	2	1	0

Signed 1's

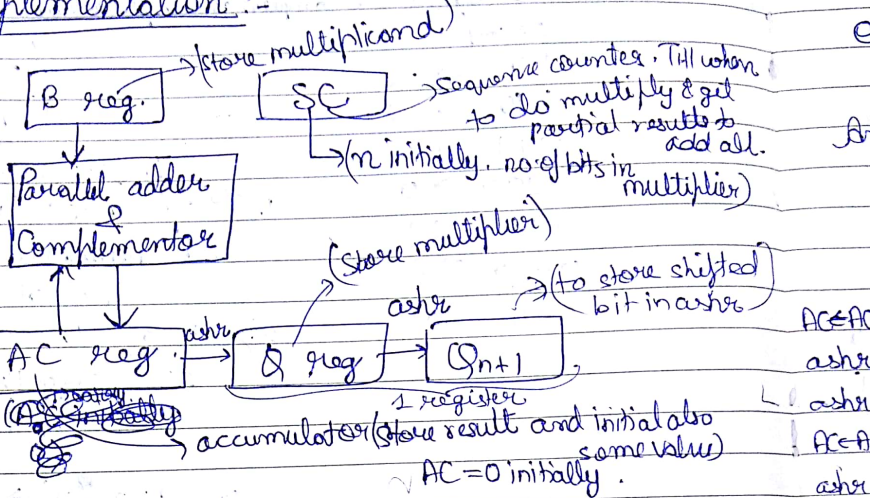
7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0

Signed 2's

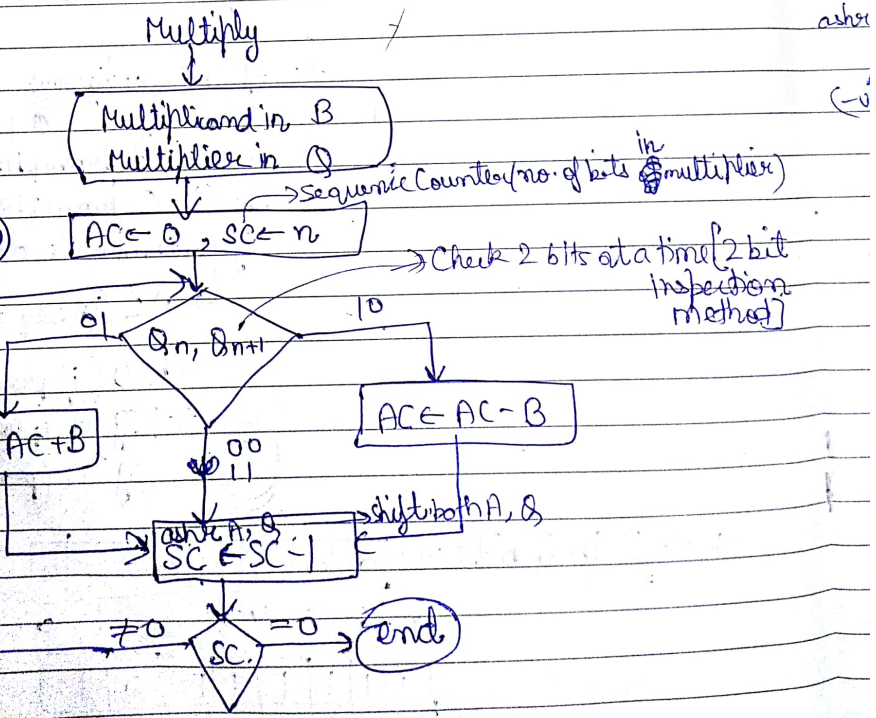
Booth's uses 3rd prop to multiply -ve numbers.

(7 → multiplicand, ×3 → multiplier, 21)

H/W Implementation :-



Algorithm :-



(Types of shift) Shifting :- circular shift :- (cshr, cshl) 0011 = 1001

(shl) (shr) Logical shift :- 0011 = 0001 (0 always here)

(ashl) (ashr) : Arithmetic : for -ve used only : acc to sign (MSB) :- 0001 (+ve), 1001 (-ve)

☆ [Result in AC, Q both]

eg :- $-7 \times 3 = ?$

$\begin{bmatrix} B = -7 \\ Q = 3 \end{bmatrix}$ = 1001 : $\bar{B}+1$ = 0111 (+7) used for subtraction.

= 0011

Ans :-

If 4 bit B, Q :- AC is of 4 bit also same for 8 bit

	AC	Q	Q_{n+1}	B	SC
	0000	0011	0	1001	4
AC←AC-B	0111	0011	0	AC←AC-B	
ashr Q,A	0011	1001	1	ashr	3
ashr Q,A	0001	1100	1	ashr	2
AC←AC+B	1010	1100	1	AC←AC+B	
ashr Q,A	1101	0110	0	ashr	1
ashr Q,A	1110	1011	0	ashr	0 (End.)

(-ve number)

Result : 3 ways :- 1.) $-128 + 64 + 32 + 8 + 2 + 1 = -21$. Ans (Using prop ①)

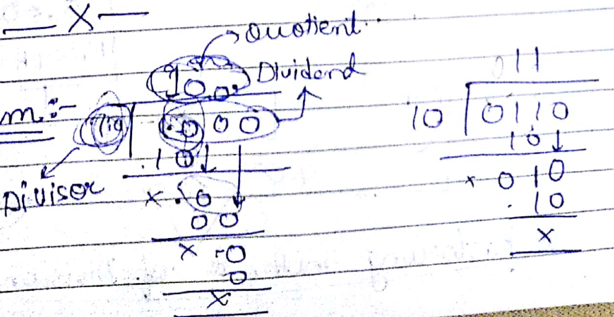
2.) $-2^0 + 2^2 - 2^3 + 2^4 - 2^5 = -21$. Ans. (Using prop ③)

3.) 2's Comp :- 00010100

00010101 (21) Ans = -21

— X —

☆ Division Algorithm :-

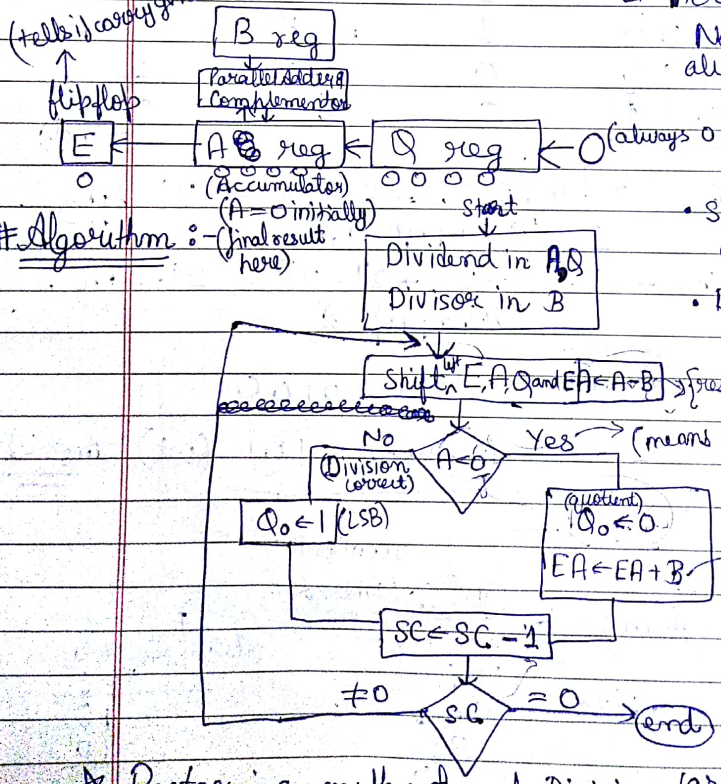


- Scan dividend from left → right. Perform only if equal or greater than. If not, move right & add 0 to quotient & so on. → (add 1 to quotient)

☆ **Divide overflow:** If quotient bits > divisor bits.
Happen when first n bits of dividend ≥ divisor.

(No overflow) { divisor bits = quotient bits.
If first n bits of dividend < divisor }

Hardware implementation:- { Here shift leftwards, so increase number. Non arithmetic as always 0 is added. }



- SC ⇒ initial ⇒ no of bits of divisor.
- Dividend stored in 2 times size reg as of divisor.

Algorithm:-

☆ **Restoring method of Division** (as restore minus one by adding if subtracted wrongly).

eg: 7/3 = ?

7 = 0111 ⇒ 11 ⟌ 0111

3 = 0011

A,Q ⇒ 7 = 0111

B ⇒ 3 = 0011 (4 bits) ⇒ AQ size = 8 bits (double of B).

∴ AQ = 7 (00000111)

(complement) $\bar{B}+1$ = 1101 (2's comp. of B).

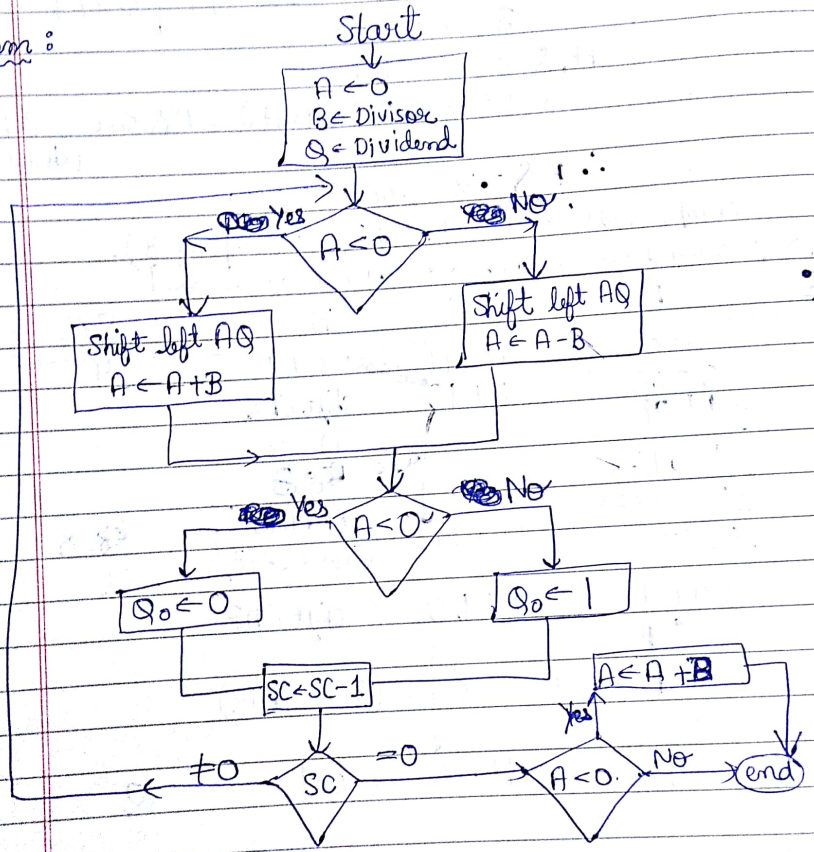
[If Carry generated, A is +ve, if not ⇒ A is -ve]

E	A	Q	B	SC
0	0000	0111	0011	4
0	0000	1110	shift EAQ	
	+1101		A−B	
	1101 (-ve)		Q_0=0	
	+0011 (B)		(EA←EA+B' ; Q_0←0, A+B)	
1	0000	1110		3
0	0001	1100	shift EAB	
	+1101		Q_0=0 (A+B)	
	1110 (-ve)			
	+0011			
1	0001	1100	0011	2
0	0011	1000	shift E,A,Q	
	+1101		Q_0=1	
1	0000 (+ve)	1001	Q_0←1	1
0	0001	0010	shift EAQ	
	+1101		A−B	
	1110 (-ve)		Q_0=0	
	+0011 (Restore)			
1	0001	0010	Q_0←0, A+B	0

Remainder (1) — Quotient (2) — Ans

☆ Non-restoring method of Division :-

Algorithm :



P.T.O.

eg: 7/3 $\quad\quad$ $\bar{3}+1 = 1101$.

A	Q	B	SC
0000	0111	0011	4
0000	1110		
+1101			
1101	1110		
	$\uparrow Q_0 = 0$		
1101	1110		3
1011	1100		
+0011			
1110	1100		
	$\uparrow Q_0 = 0$		
1110	1100		2
1101	1000		
① +0011	$\uparrow Q_0 = 1$		
0000	1001		1
0001	0010		
+1101	$\uparrow Q_0 = 0$		
1110	0010		0
① +0011			
0001			
Remainder	0010 Quotient		

$\therefore$ Quotient = 0010 = 2
Remainder = 0001 = 1 } Ans

—X—

Floating point Arithmetic :- (no numericals in exams from it)

"HW"

↳(commⁿ line b/w devices)

"Bus interconnection" :- Memory is sequential.

(1) Data bus/line (2) Address Bus (3) Control Bus (4) Power dist.

data sent. controlled by.

↳ Data width ⇒ word size/data/register size of uP. (No. of data lines).

↳ {No. of lines ∝ 1 / no. of cycle to transfer data}

(2) Address line :- Addⁿ of mmry location put here.

↳ [no. of address lines ∝ Size of memory]

[no. of address line ∝ addressable words in mmry]

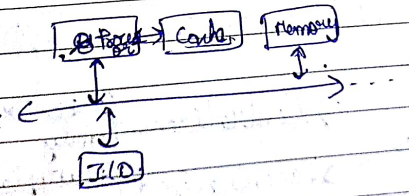
↳ (Controlled by memory size)

(3) Control bus :- control signals. Signals which control commⁿ b/w uP and memory.

(Cache → Prim.)

Processor → Memory → I/O devices
(fastest) (slowest).

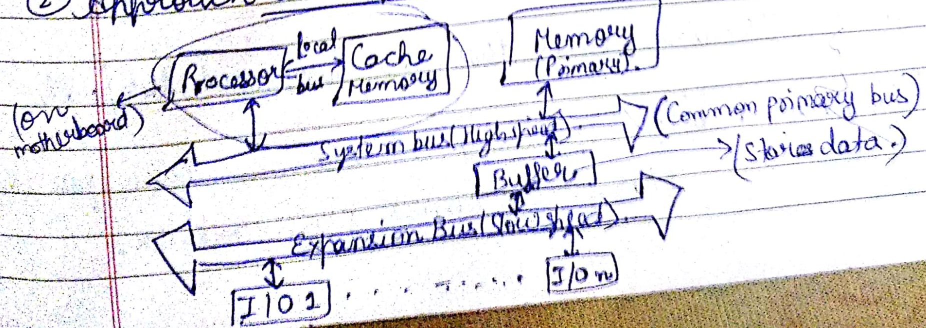
⇒ (1) Approach : Centralised Bus Architecture :-



Disadv:- • As ↑ devices (I/O), bus length go on ↑, so ↑ delay of transfer.

• I/O have generally slow speed as processor & mmry. So increase delay here as Processor wait for I/O devices.

(2) Approach : Multiple Bus Architecture :-



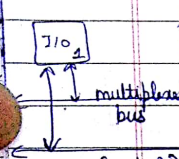
- Bus design: Take in note these points →

1) Elements of bus design ⇒

(i) Bus type :- what type data convey :

Multiplexed Bus (a) ← "2 types" → (b) Dedicated bus

- Same bus is used to transfer data as well as address.



(dedicated for purpose)

- Permanently assigned either to one function; i.e, to convey data or to convey address, or we can use these lines for physical subsets of computer component. (I/O 1 & Processor etc)
- Adv :- No wait for system bus for 2 devices intended
- Throughput, efficiency ↑.
- Disadv : Price ↑, space ↑.

I/O copy address on Mul bus & invoke valid line. (Placed data is valid & we can access it)

- All check then address putted belong to that device, then data at address is copied on multiplexed bus & can be read data.

Adv :- Space ↓, Cost ↓.

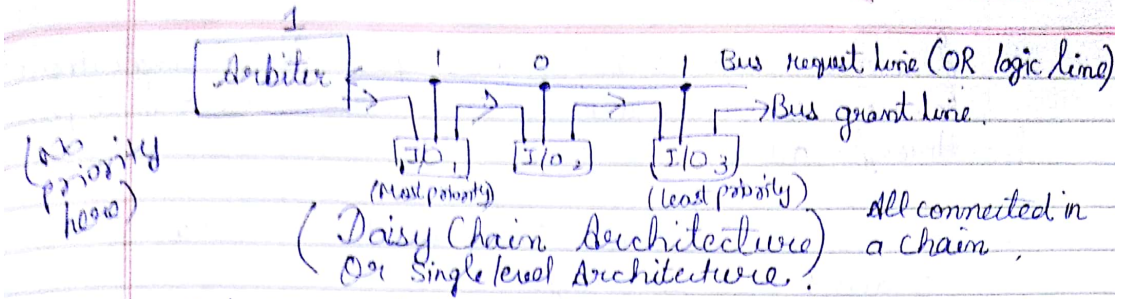
Disadv : Complexity ↑ (Need control lines more), Difficult to implement.

(ii) Bus arbitration → Centralized (1 central arbiter only)
→ Decentralized (multiple no. of arbiters)

→ process of sharing bus among multiple devices

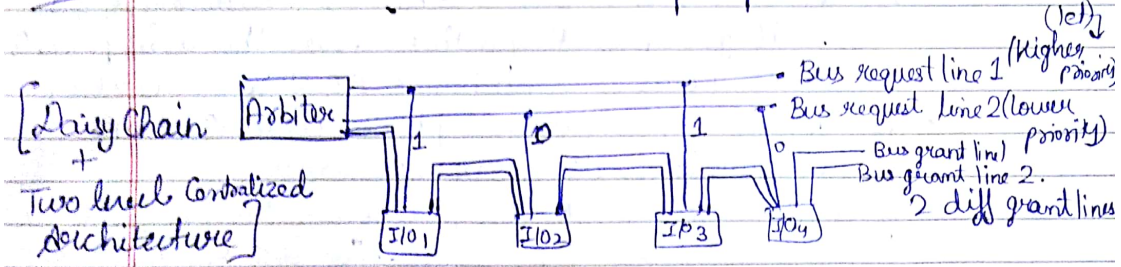
- Arbiter :- controller to do arbitration. Can be µP or other device also.
- In decentralized, each device has own controller/arbiter

Centralized Arbiter ⇒ Bus grant line in a serial fashion covers all I/O devices, comes from arbiter.



- OR logic :- multiple devices can generate bus request at same time
  ↳ Hence Arbiter can tell that someone demands bus but not tell how many demand as only 1 go to it.
- Arbiter checks timely basis. As soon as bus free, sends grant signal. As serially connected, so first go to closer to arbiter device. → It's disadv. also. Others can starve also. No matter. Starving state
- If 1st I/O not need bus, pass grant signal to next device & so on.

*Remove this disadv :- Devices divided as per 2 priorities levels.



- Here have 2 levels. Disadv :- If same priority devices need bus, then which one is closer, gets first grant.

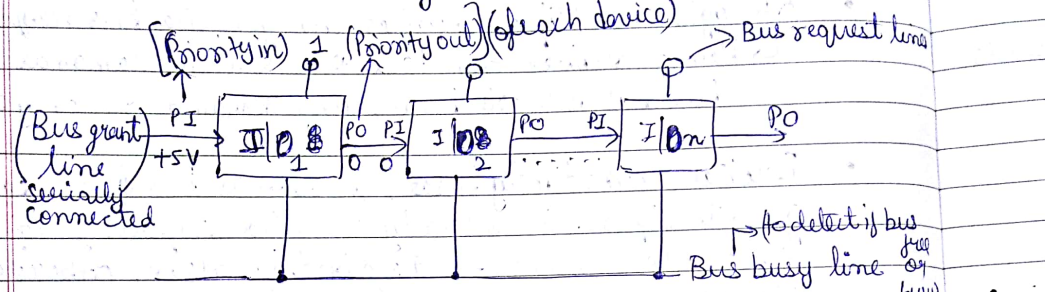
Decentralized Arbiter ⇒

1st way: All connected serially.

2nd way: Keep encoder & decoder. Requests go to encoder, decode it & generate single o/p. eg: use 4×2 encoder, then 2×4 decoder. One of them is activated.
(1 of them is activated)

Decentralized Arbitration Algo :- Every single device has own controller/arbiter. Every single bus request pass through all controllers.

1.) Serial Arbitration : Bus grant line to all devices in a chain order serially.

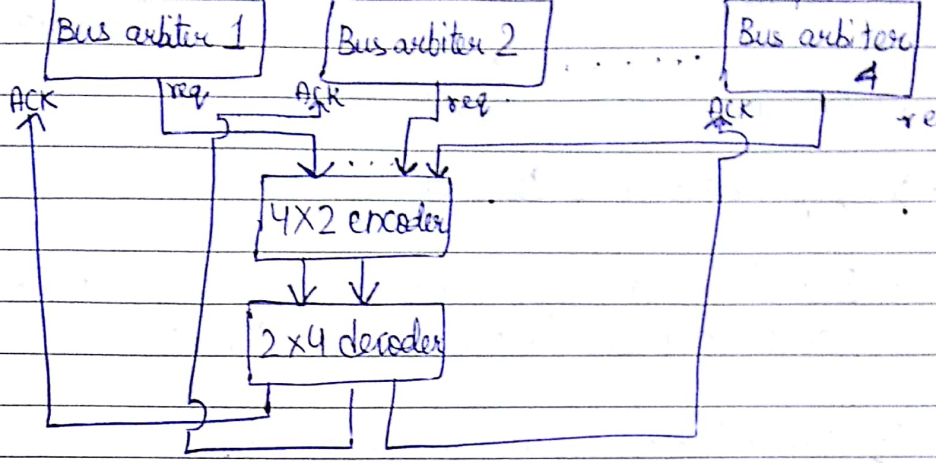


- Bus request line connected separately to each one
- PI of first device is always at high logic level (+5V)
- If ① requests bus, check if PI = high, true, If yes, so it gets bus & further down the line passes 0. Passes PO as 0 and hence PI of next = 0 also.
- Rest devices in starvation phase : until ①st one not completed its work
- If 0 for I/O_1, so PI pass down the line. So PO = 1 ⇒ next PI = 1 go. So get bus by who has [PI = 1 & requesting bus] and so will pass 0 on down the line.

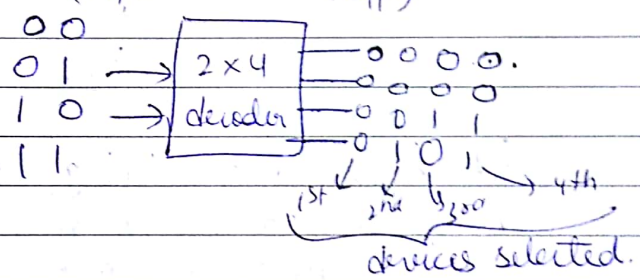
2.) Parallel bus Arbitration :- Each has 2 logics : ACK & req. (i/p pin) (o/p pin)

(Device enabled when get ACK = 1 & access bus now) (and req. them req lines)

{ All devices generate req. at same time. Encode them. Decode them and then any one line will be enabled. (eg: 0001) → activated.



- No priority here. Device ACK on basis of O/p of decoder only



- Check Bus Busy line while asking for bus each time. If 0 on busy line ⇒ generate next request.

* Above all fixed algo. Cannot change logics above. All Called Static Algorithms. (Cannot change process). Here always closest one gets first. So (use them to counter it)

* Dynamic Algorithms :-

1) First come first serve: Disadv :- if each time same requests first, so get agn & agn.

2) least recently used :- B = 5 used, A = 1 used. So A get.

3) Round Robin / Time slice: Time divided. Cuiem to each device, equally time slots in round fashion. Start from A (10 sec). If need, so access, if no need, pass it & so on; (After but go to 1st agn).

Adv :- (No starvation, no priority etc)

- Elements of bus design ⇒ (Continued)

(iii) Bus Timing: (Event 1 has to be synchronized. Event 1 just started or Event 2 ended, etc.)

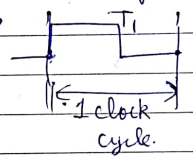
Synchronous ← → Asynchronous.

☆ Synchronous Bus Access :-

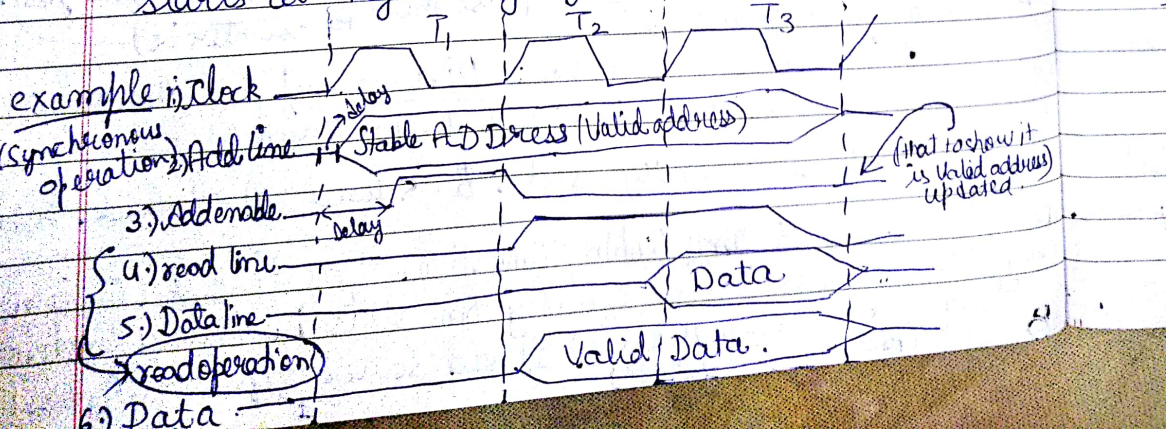
- Bus timing refers to a way in which the events are coordinated on the bus.

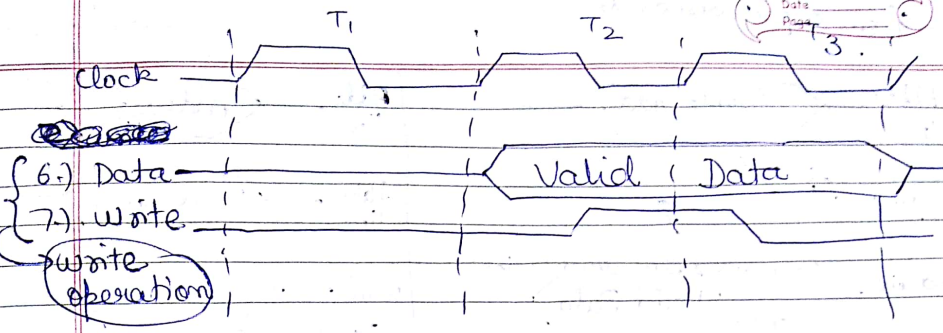
Events → coordinated with clock cycle.

↳ signal of 0 & 1.

1 clock cycle ⇒ 

- With Synchronous timing, the occurance of event on the bus is determined by a single clock. The bus design now includes a clock line which is used to transmit a regular sequence of alternating 1's and 0's of equal duration. A single 1-0 transmission is known as clock cycle or bus cycle.
- In Synchronous bus timing, all other devices connected to bus regularly reads the clock and all event starts at beginning of clock cycle.





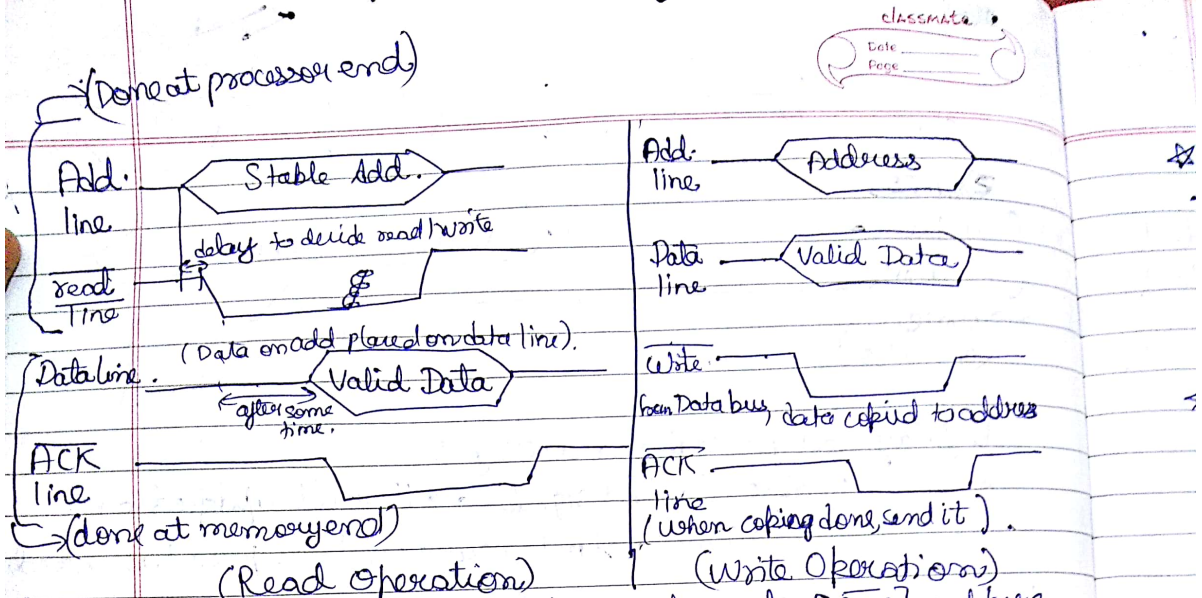
* Delay : to get stable address get.
* Add. enable : Control signal to tell valid/updated add.
  ↳ Time to start new event as prev has ended.

Read operⁿ : All device check which has that address, loads data on data line.
- Read line :- to tell that read this data. (It is a second event). After completion of prev event (Add enable), start it now. It remain high till read operⁿ not completed. As soon as get read data, it gets low.

Write operⁿ :- First need some data : So put add on add bus. As add enable high. Data load add on data bus in T_2 cycle. After some delay (data on data bus still now), now write line ↑.

Asynchronous Bus Timing :-

P.T.O.

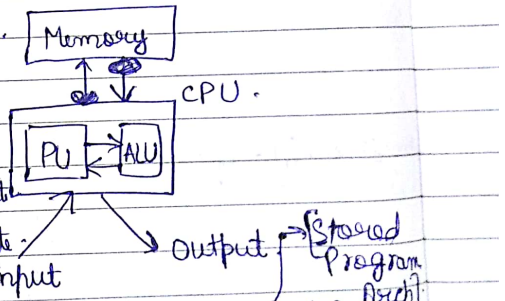


- No clock here : Use 3rd party Control signal. [ACK] used here

---

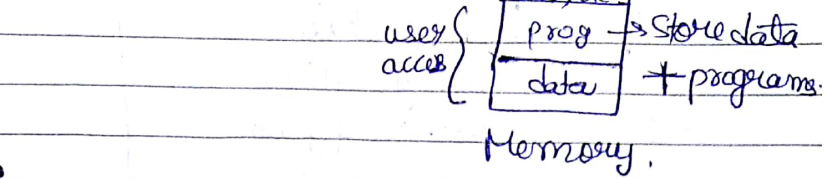
Computer Architecture :-

- Serial Arch : Von-Neumann Arch.
- Parallel Arch.
  - Instⁿ executed serially.
  - To achieve parallelism → Instⁿ parallel execute
    - → Data parallel execute :

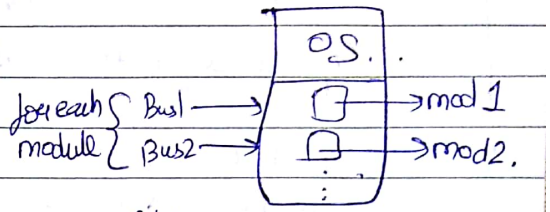


(Von-Neumann Arch.) → Stored Program Arch

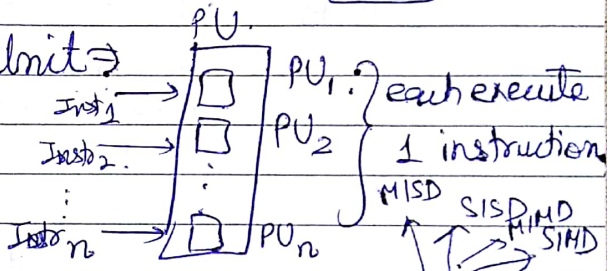
- Earlier, operators collects/groups set of instⁿ ⇒ Jobs, and then give it one by one as i/p to Control Unit (all have same regⁿ)
  Then process it & give o/p. ⇒ Before Von-Neumann
  → Instⁿ not stored in memory.
- Von Neuman ⇒ said → store program & data (variables) both in memory. Hence called Stored Program Architecture.



* **Parallel arch.** :- so that can execute multiple instⁿ at a time or fetch multiple data at a time.
  - Divide processor/memory in modules for it.
  - If fetch multiple data ⇒ memory in multiple modules



- Or can divide Processing Unit ⇒



⇒ **Model for Parallel Arch.** → Flynn's Model / Classification

- Comp. arch divided by using Instruction & Data Stream concept.
- **Instruction** → 2 parts :- (stored also in memory)

eg: C = A + B

Opcode (operation code) → assign binary codes for each oper.ⁿ

- **Memory** ⇒ sequential. A, B can be in anywhere memory.

Sequence to execute instⁿ
1.) Fetch ~~multiple data~~ instⁿ.
2.) Decode
3.) Fetch data.
4.) Execute instⁿ
5.) Store result in memory location. (C)

eg:

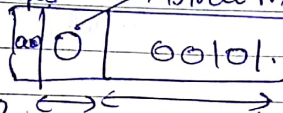
- **Instruction Stream**
  ⇒ Flow of instⁿ (processor demands it from memory location)
  - One time flow. Only from memory to processor.
  - Flow of instⁿ from memory to CPU established.

- **Data Stream**
  ⇒ Flow of data (processor demands it from memory loc.) or vice versa
  - from proc to memory & memory to processor both
  - Flow of operands.

- **Per instⁿ**: Access/flow of memory for instⁿ fetch ⇒ 1
  Access/flow of memory for data fetch ⇒ 0.5

- Operand further divided in 2 parts:-

Instⁿ (Parts):-



→ How to fetch data from location specified

eg:- Direct: (0) } let use.
Indirect: (1) }
- Direct: get data directly at location.
- Indirect: it is memory location where address of data stored.

Flynn Model

- If arch has 1 instⁿ & 1 data stream ⇒ at a time can only fetch 1 instⁿ/data executed or fetched at a time only ⇒ Serial Arc. (SISD): Single Instⁿ Single Data: (Called).
- SIMD
- MISD
- MIMD ⇒ Pure Parallel architecture. Both executed parallelly.

→ Memory accesses per instruction = 1 + 0.5 (1 instruction access + 0.5 data access).

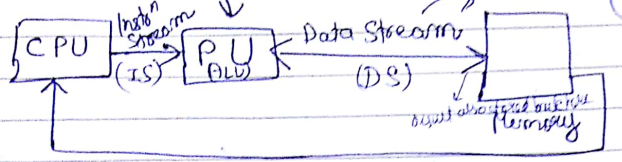
#Imp

SISD :-

(Classical Von Neumann Arch.)

C = A + B

Passed here from CPU to it.



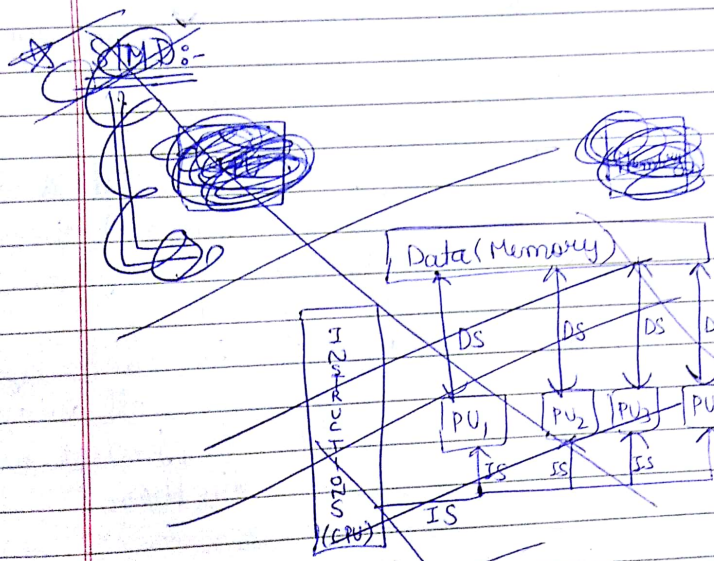
2 clock cycles to access A, B

(IS = DS = 1)

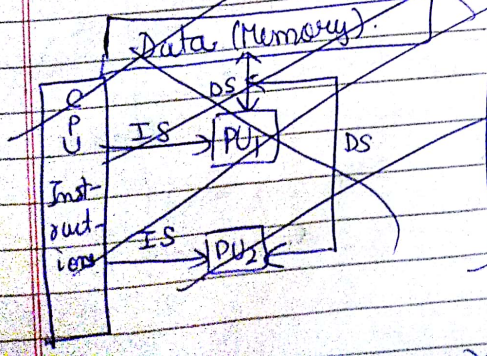
(IS) Instruction Stream (Unidirectional)

Draw rest arch" :-

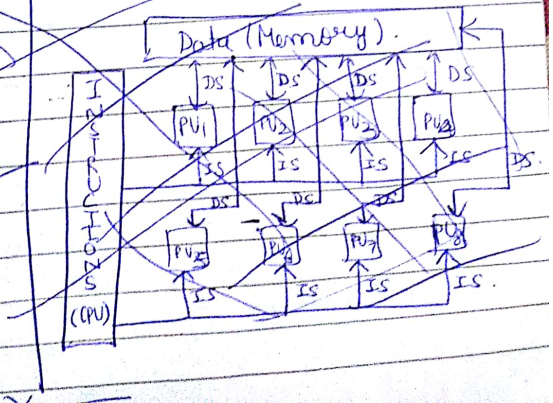
SIMD :-



MISD :-



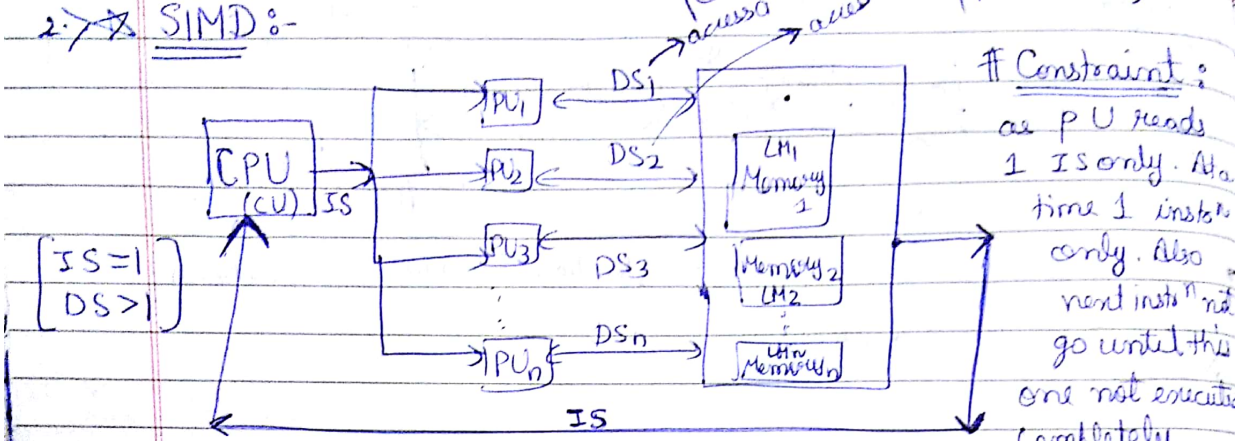
MIMD



all processors in locked down state. Until prev. instⁿ not completed, not go other instⁿ. Waits. Cannot jump to next instⁿ

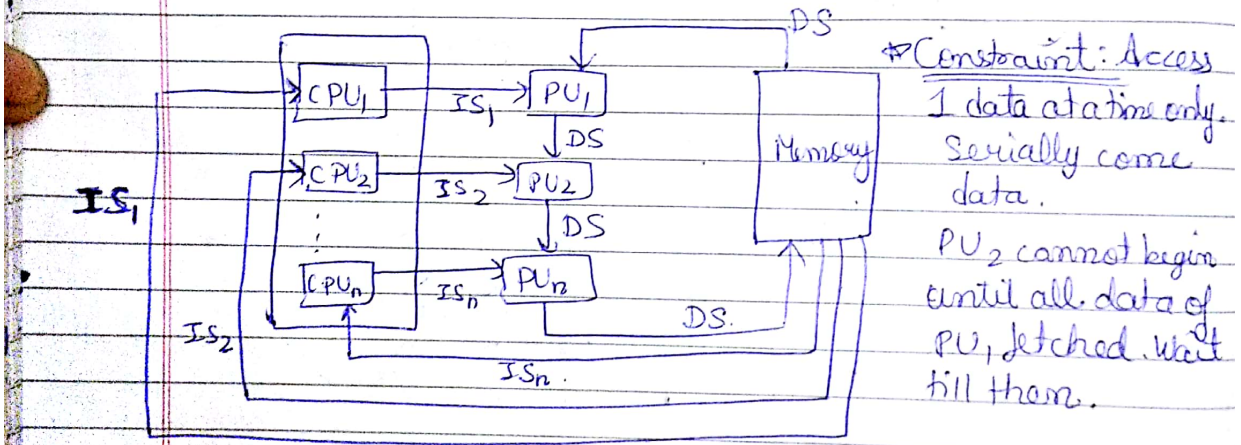
↳ "More correct"

2) SIMD :-



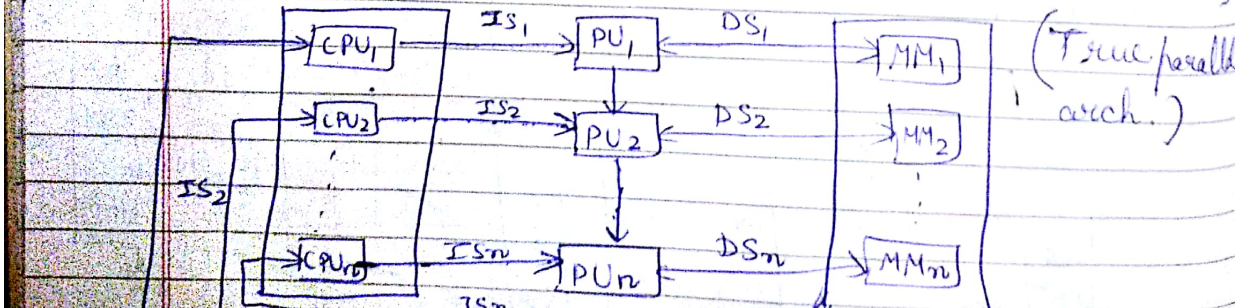
Constraint : as PU reads 1 IS only. At a time 1 instⁿ only. Also next instⁿ not go until this one not executed completely.

3) MISD :- IS>1 , DS=1



Constraint: Access 1 data at a time only. Serially come data. PU_2 cannot begin until all data of PU_1 fetched. Wait till them.

4) MIMD : IS, DS>1 . [(Parallel purely) : No locked stepped here]



* Computer cycles per instruction (CPI) is 1.0 when all memory accesses are cache hits. The only data accesses are loads and stores, representing a total of 50% of instructions. "Imp"

—X—

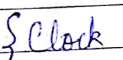
* SIMD also called { Synchronous execution arch. Or Locked stepped arch. }.

—X—

* System performance Attribute :-

1) CPI (Clock rate) :- [Cycle per instruction.]

signal of alternating 0, 1 to synchronize.

{Clock cycle →} 

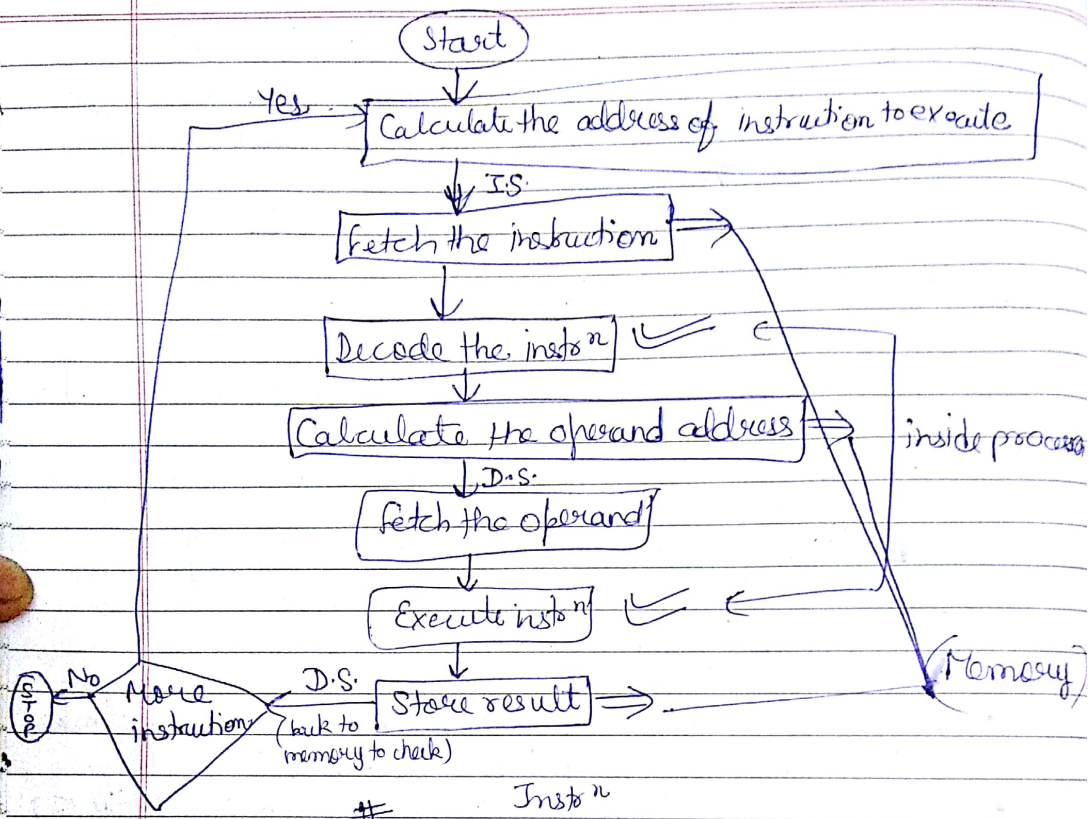
- The processor is driven by a clock with constant clock time measured in nano-seconds (n sec).
- Clock rate is defined as inverse of clock cycle time and is expressed in MegaHz (Mz). (τ)
- Clock frequency = $\frac{1}{\tau}$ Mz.

[I.C. (Instⁿ count) ⇒ Defines size of prog.]

- Size of prog. determined by instⁿ count (I.C.) and diff instructions may require diff clock cycles to execute.

P.T.O.

"Flow Chart" : "Flow of Instⁿ execution"



Instⁿ

Processor cycle.	Memory cycle.
Decode instⁿ, Execute instⁿ.	Fetch instⁿ, Calculate operand add, Store result.

requires (Processor cycle) — requires (Memory cycle)

Time to execute a program (total) = (I.C. (total instⁿ) × CPI) × T.

(CPI: total no. of clocks)

→ Depends on types of instⁿ in prog. (Processor, Memory)

Memory cycle slower than processor cycle.

If more processor cycles ⇒ as they are more fast. So less time

Time = I.C. × CPI × τ (depends on type of instn)

Time = I.C. × (P + M × K) × τ.

P → Processor type instn cycle (fast).

M → (Memory cycle) (slow)

K → ratio b/w processor & memory cycle.

factor (constant).

To equalise P, M ⇒ M × K done. As M already less fraction.

—X—