Computer is a set of programmes
set of instructions

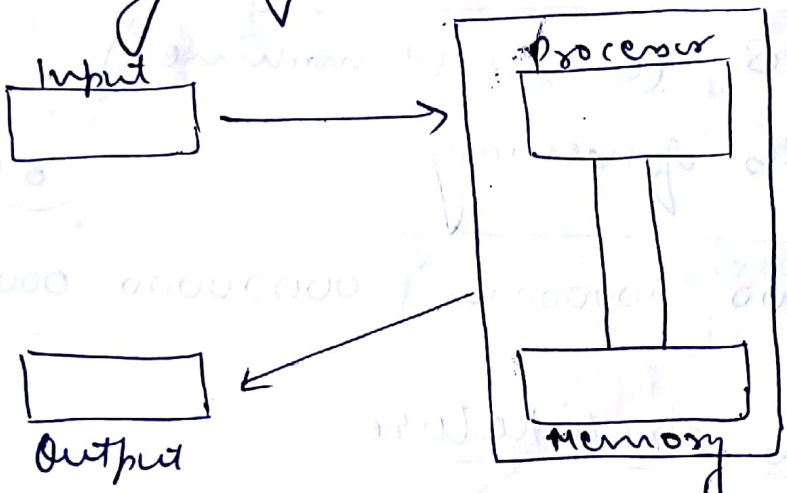
Computer needs a memory to store the program (processing unit → CPU)

Mechanical machines → Vacuum tubes
Transistors
Integrated circuits → VLSI
(very large scale integration)
(large no. of chips in small area)

Embedded computers

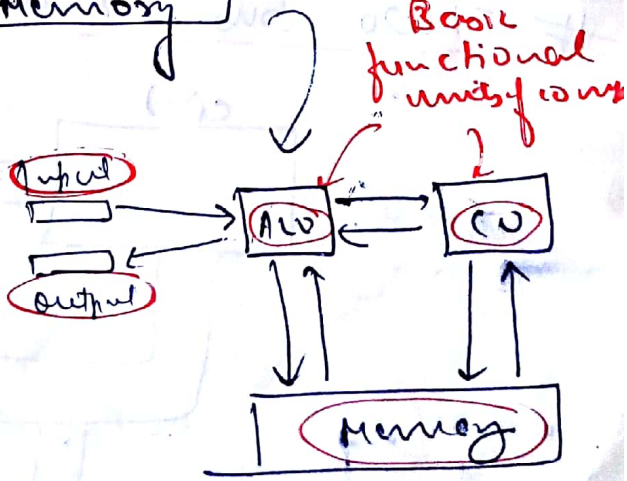
Moore's law
The no. of transistors we put in a area will double every 2 years.

#



Von Neumann Architecture

GOOGLE GLASS

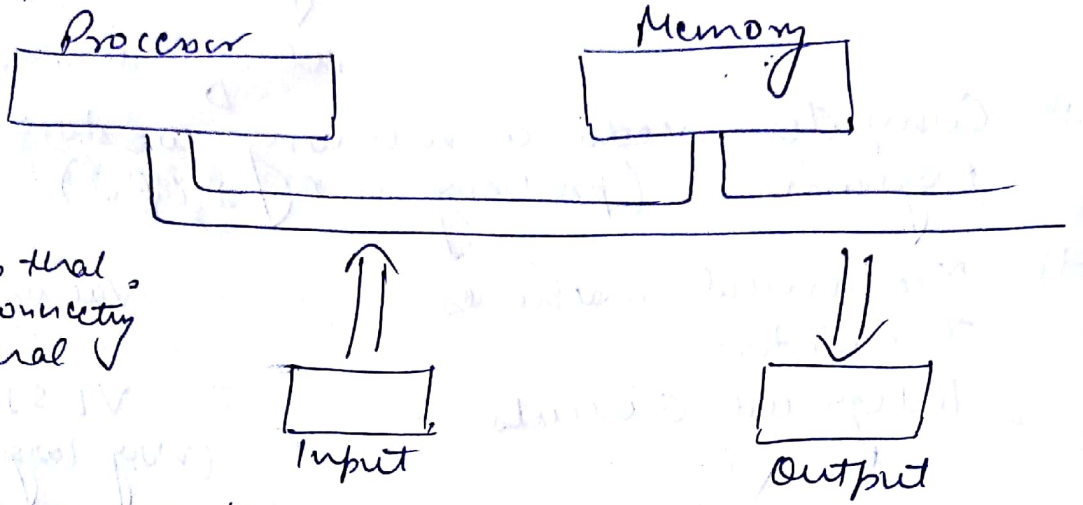


Functional Units

Combinational elements (elements that operate on data value)

State elements (elements that contain state)

Single bus Architecture



Group of lines that serves as a connecting path for several devices.

Bus is a set of wires
↳ data is travelling over these

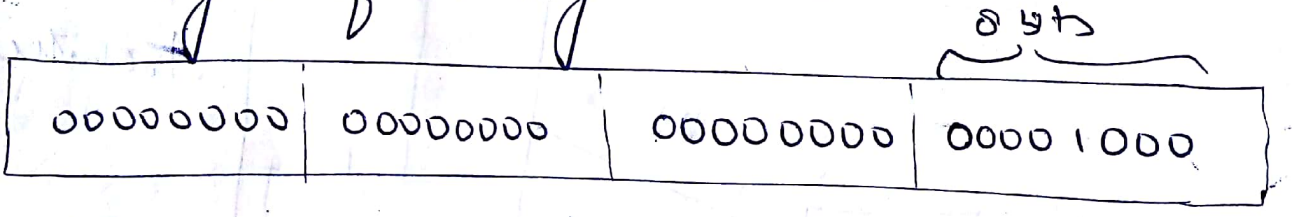
64-bit Machine

i.e. can process 64 bits of data in one go.

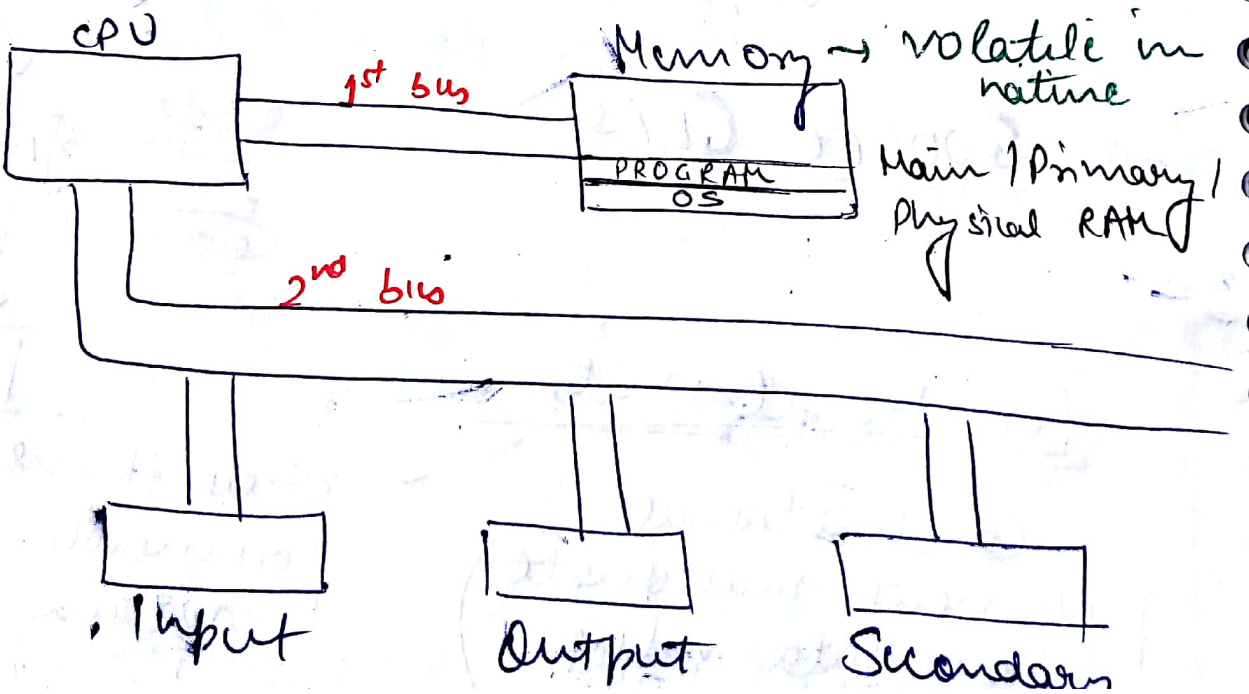
nibble = 4 bits

int a = 8 (in 32-bit machine)

↳ 4 bytes of memory



Two bus Architecture



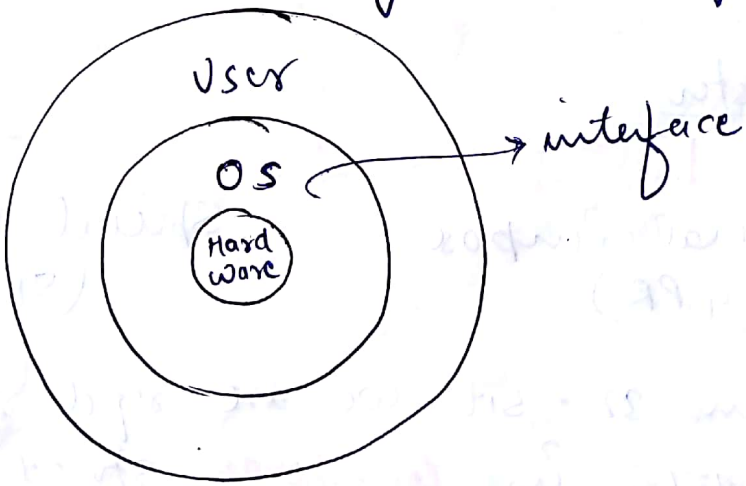
Program gets stored in Hard Disk (secondary device)

non-volatile in nature

data will not get rubbed if we ~~dor~~ close a file / remove that device from PC

OPERATING SYSTEM (a software)

→ Manages each and everything in PC.
→ also gets a memory in RAM.
→ When we give a call (i.e. run a program), we ask OS or tell it that we are running a program. Now it's the job of OS to assign a memory to that program in RAM.
→ file is stored in storage device, when we execute it, it go to RAM (duplicate) and one by one instructⁿs go to CPU & get... run



HLL (high level lang.)

Compiler → Assembly lang. (mnemonics) → machine dependent lang.

Assembler → Machine lang

Modern day compilers

C = a + b

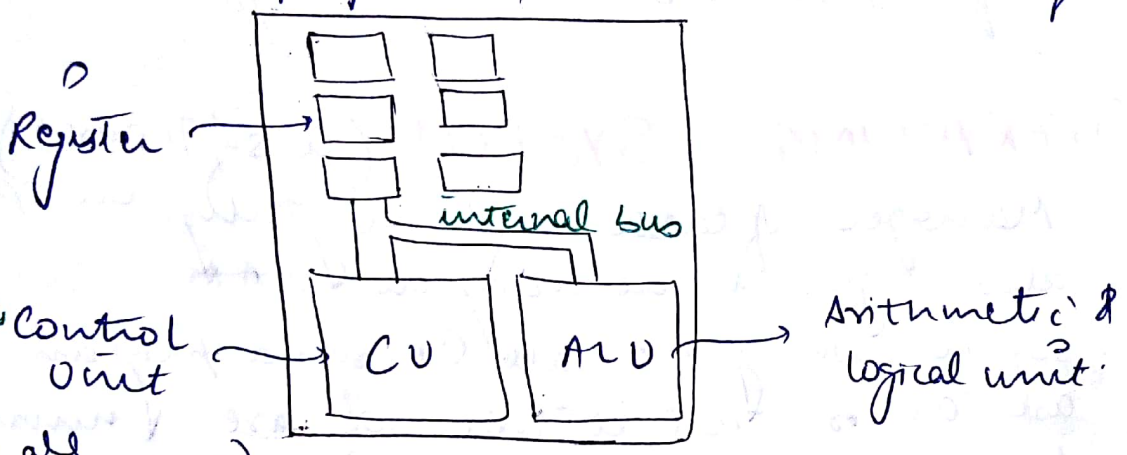
ADD R1, R2, R3

ADD → opcode
R1, R2, R3 → operands

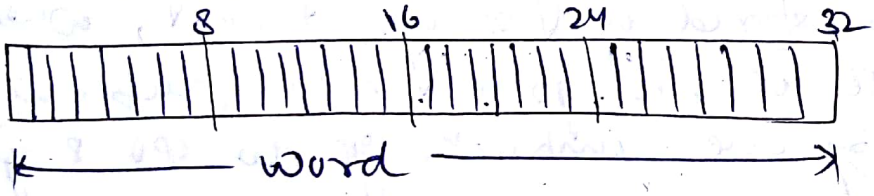
flip flop → store 1 bit of data

Register → collection of flip flops where one can store some data

CPU execute programs stored in the main memory



if 32-bit machine then



Register

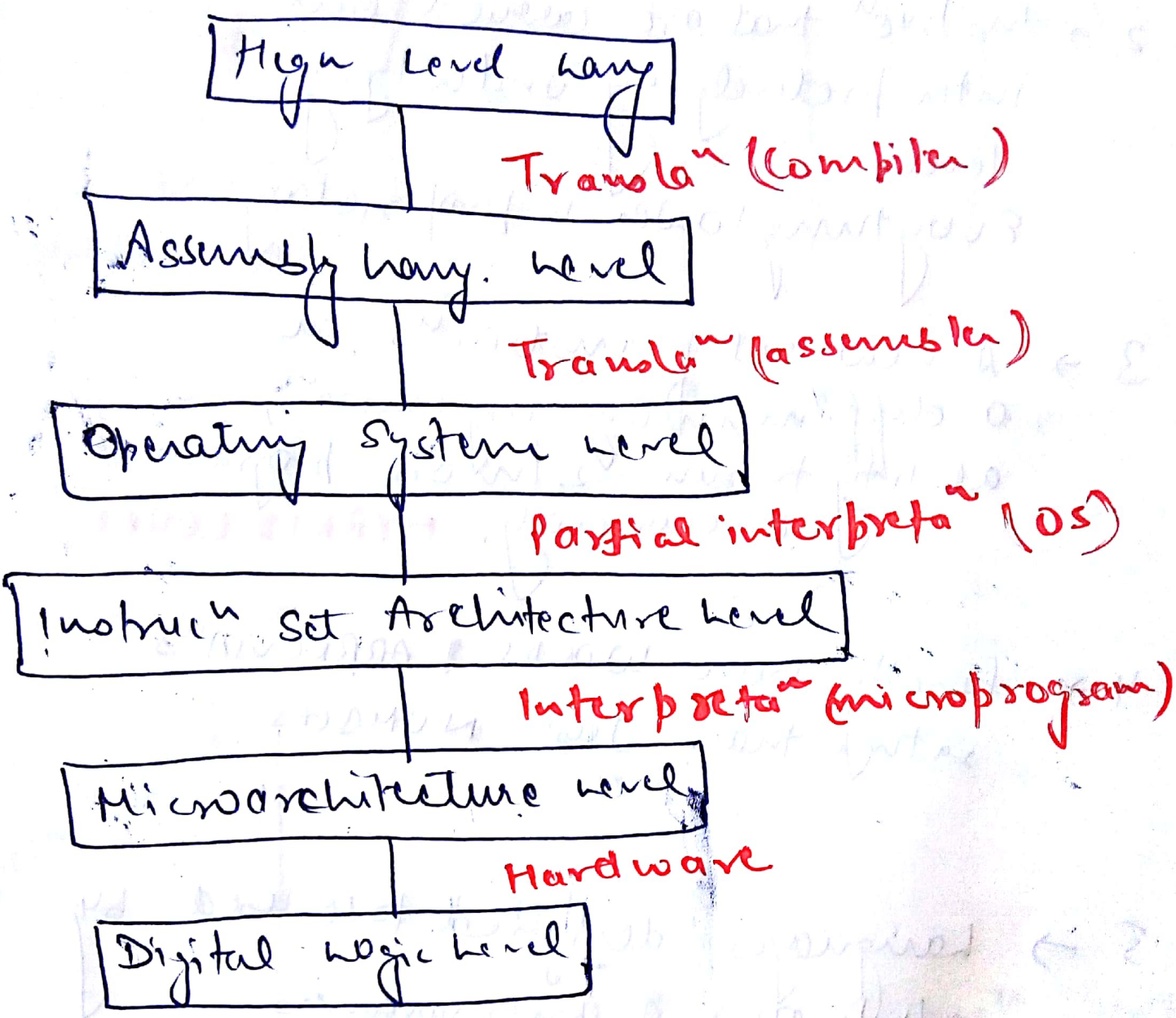
- General Purpose (GPR).
- Special Purpose (SPR)

so in 32-bit we are reqd 32 registers do not know

so data in temporary stored in registers

(1) slide 11

PC, IR, Status Reg, MDR, MAR.



0 → Gate level, made from transistors, used to form 1-bit memories.

1 → Collⁿ of 8-32 registers form → local memory & ALU

→ Registers & ALU connected forms → data path

→ Data path controlled → S/W or H/W

→ Does fetching, examining & execⁿ of instrⁿ

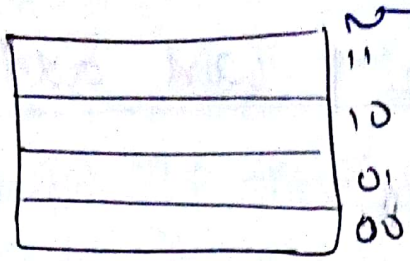
2 → Instruc^n that are carried out interpretively by underlying levels
Everything lower is proprietary

3 → A new set of instruc^n, a diff. memory organiza^n, ability to run 2 / more programs concurrently. HYBRID LEVEL

4 → finally have WORDS & ABBREVIA^n rather than JUST NUMBERS

5 → Languages designed to be used by applicans & programmers

If we have 4 words



2 bits of address

If we have 8 words ⟶ so 3 bits of address

↳ rows

n bits

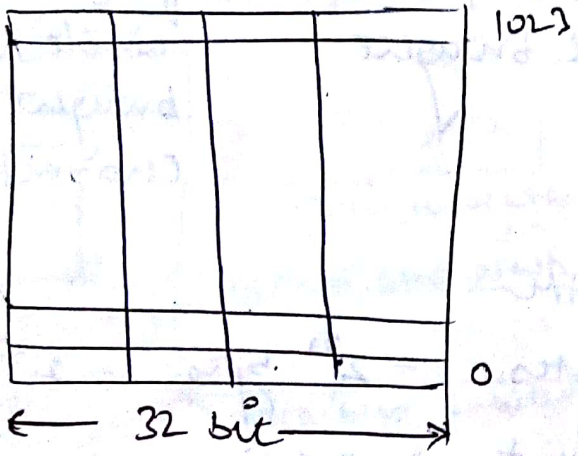
$2^n = K$

no of rows

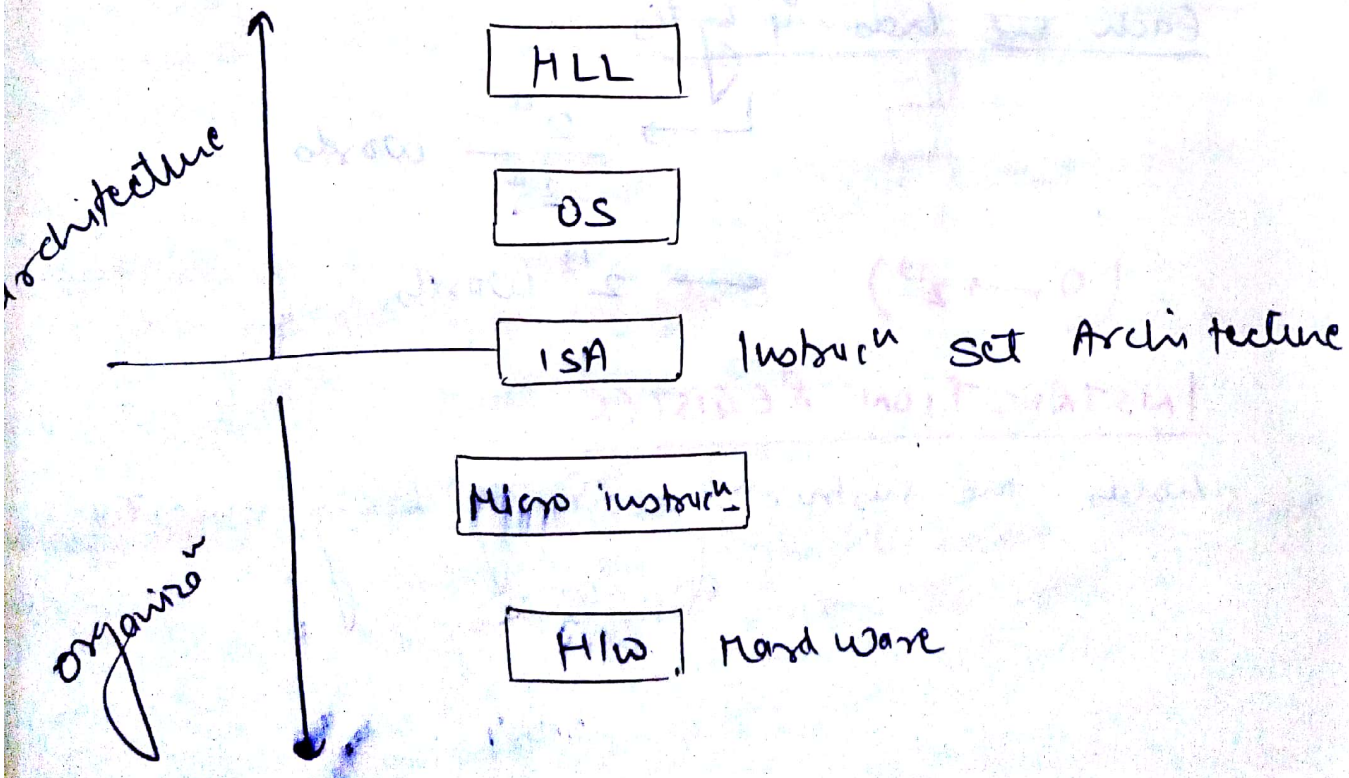
Word addressable

⇓

Each word is having an address

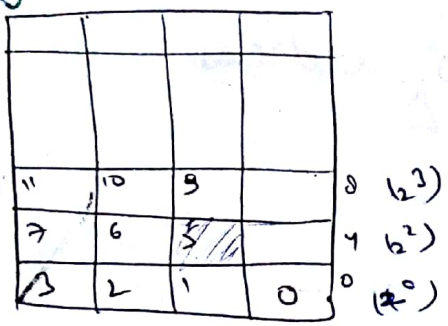


10 bits of address



Q. Diff b/w byte & word addressable

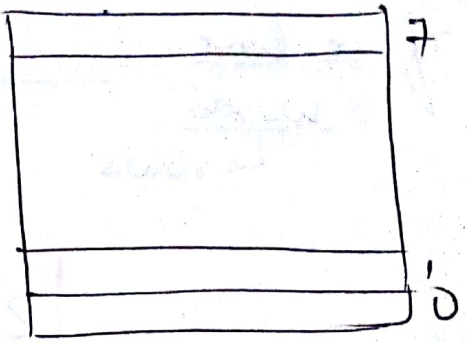
Byte addressable



int = 4 byte
char = 1 byte

if say, fetch the character from address 5, then that will be brought

Word addressable



if say, fetch the character from address 1 then, whole word will be brought and then character will be fetched

2 MB RAM

2×2^{20} bytes $= 2^{21}$ bytes $= 2^{24}$ bits

So 21 bits of address

word

Each ~~byte~~ has 4 bytes

$\rightarrow \frac{2^{21}}{2^2}$ words

$(0 \rightarrow 2^{19}) \leftarrow 2^{19}$ words.

INSTRUCTION REGISTER

holds the instruction currently being executed.

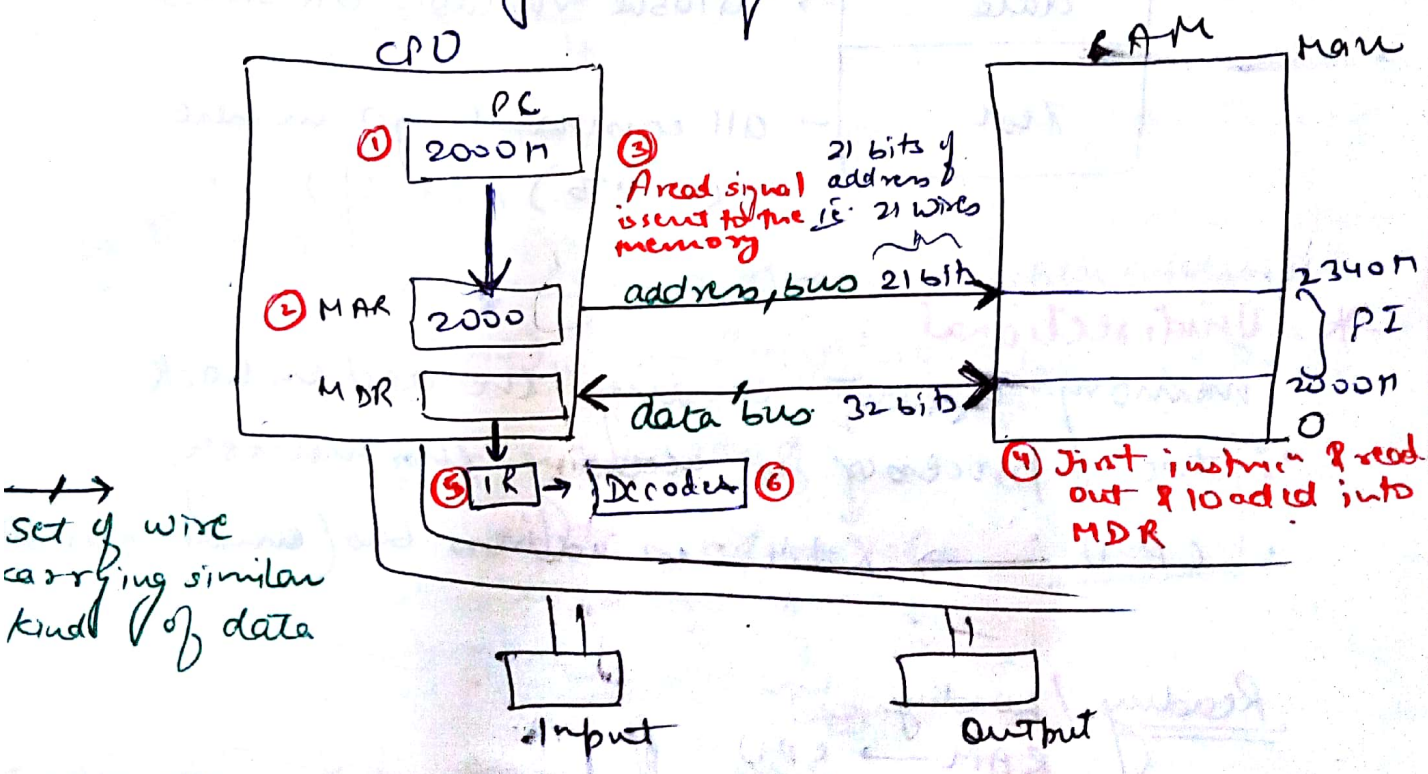
PROGRAM COUNTER (PC) → keep track of instructions

→ always points to the next instruction to be executed

→ as the next file we open and the current file is running s/o the next file get stored in a specific register (PC) int in PC

→ file stored in PC has not been executed

MAR Memory Address Register
MDR Memory Data Register



2000H

Total 21 bits (see on next page)

0 0000 0010 0000 0000 0000

→ in word addressable

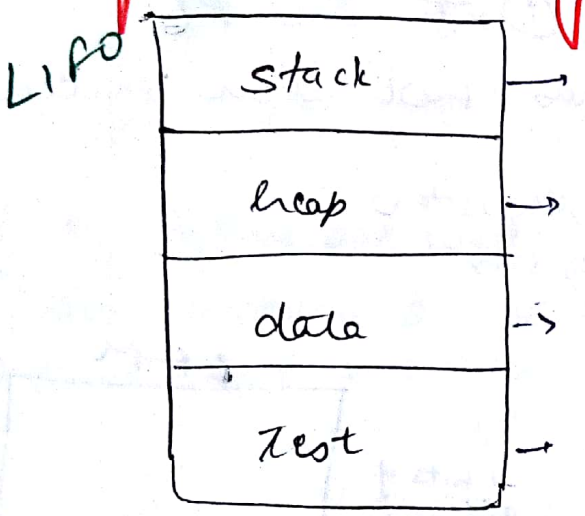
as soon as instruction is fetched from PC move to MAR so next instruction will move to PC (ie. 2001H)

if byte addressable

(move 2004H)

If 2MB RAM → 2^{21} bytes
i.e. 21 bits ∴ address
∴ Size of MAR = 21 ∴ byte addressable
= 19 ∴ word "

Program memory is divided in these 4 parts



stack → local variables are being stored
heap → if we use dynamic memory
data → Global variables are stored.
Text → all commands go in here (c=a+b), (printf), C++

Unidirectional
memory is not sending the address back
∵ the processor is accessing the memory
∴ CPU → RAM is address bus (unidirectional)

Reading / Loading
RAM → CPU
i.e. we are reading data, have access to data

Writing / Store
CPU → RAM
i.e. we are writing or storing data to memory

- Decimal to binary

Decimal = base 10
binary = base 2

eg 13 → $2^3 + 2^2 + 2^0$

1101

- Decimal to hexadecimal

Hexadecimal = base 16

eg I 2000 → 7.256 + 208 (13.16)

A = 10
B = 11
C = 12
D = 13

$16^0 = 1$
$16^1 = 16$
$16^2 = 256$
$16^3 = 4096$

7 (256s) D (16s) 0 (1s)

$(2000)_{10} = (7D0)_{16}$

eg II 2000

Decimal → binary → hexadecimal

$(2000)_{10} \rightarrow (0111\,1101\,0000)_2 \rightarrow (7D0)_{16}$

nibbles of 4

0111 / 1101 / 0000

7 / 13 (D) / 0

Hexadecimal to Decimal

$(7D0)_{16} \rightarrow$

0	$16^0 \times 0 = 0$
D	$16^1 \times 13 = 208$
7	$16^2 \times 7 = \ldots$
	2000

$(1AB)_{16} \rightarrow 16^0 \times 11 + 16^1 \times 10 + 16^2 \times 1$

$= 427$

- Decimal to Octal

25 ⟶ Binary ⟶ Octal

11001

nibbles of 3

011 | 001

3 1

$(31)_8$

Octal to decimal

$(42)_8$ ⟶ $8^0 \times 2 = 2$

$8^1 \times 4 = 32$ ⟶ 34

fetch - Decode - Execute

2400 H address ⊕ int in PC

from here it moves to MAR & PC get incremented by 4 (byte addressable)

Then MAR pass that address through address bus and get the access to data.

Data is placed to MDR.

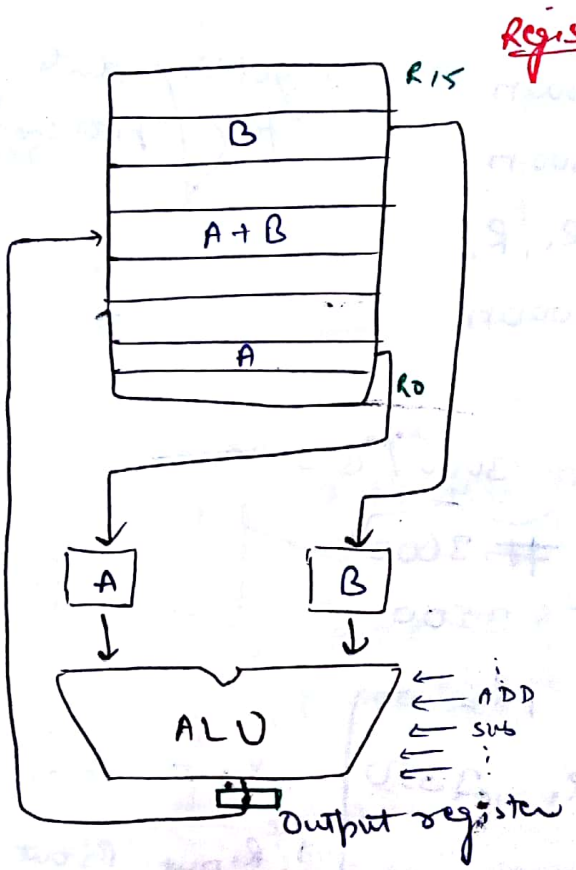
MDR passes that data to IR

Decoder decodes the instruction ⊕ int in IR.

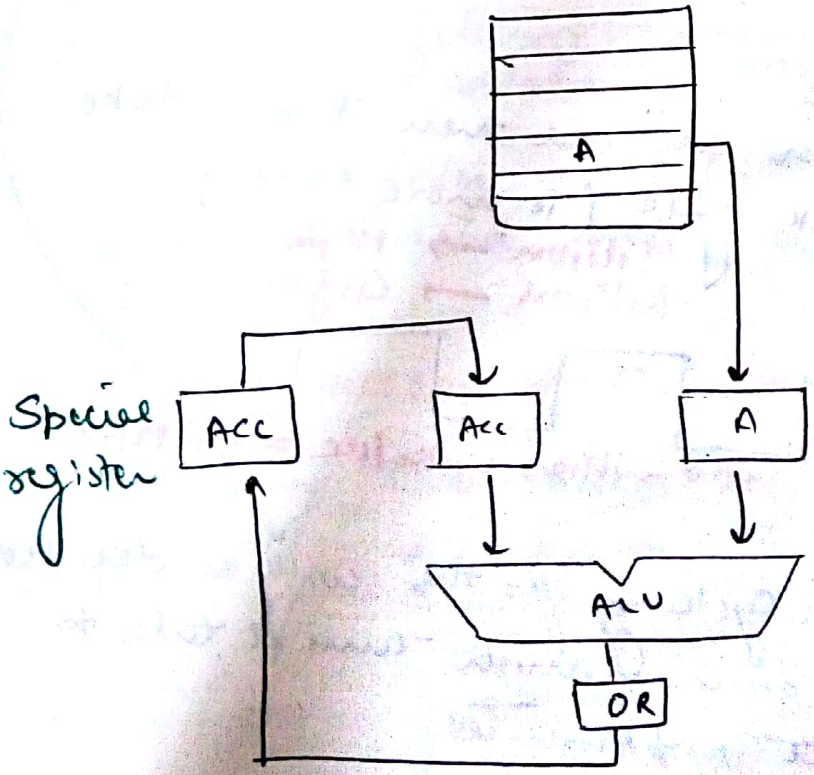
	opcode	operands	destination	source
3 address instruction (due to ⊕ nce of 3 operands)	ADD	R_1, R_2, R_3	$R_1 \leftarrow R_2 + R_3$	
			$R_3 \leftarrow R_1 + R_2$	
2 address instruction	ADD	R_1, R_2	$R_1 \leftarrow R_1 + R_2$	
			$R_2 \leftarrow R_1 + R_2$	
1 address instruction	ADD	R_1	$Acc \leftarrow Acc + R_1$	
0 address instruction	ADD			

Accumulator (Acc)

Special place register where data is ⓟut like in case of 1 operand ⓣut in ADD other operand is taken from accumulator



for 1 Address



Q. How value get stored in ACCUMULATOR

```
int x, y, z
x = 200
y = 250
z = x+y  ⟶ HLL
```

```
LOAD  R1 , 3000H
LOAD  R2 , 3600H
ADD   R3 , R1, R2
STORE R3 , 4000H
```

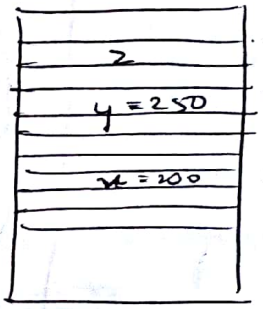
→ getting data from memory to processor

⇒ z = x + y

3000 / @3000

Assembly lang. instruction

```
LOAD  R1 , #3000
LOAD  R2 , 3200
ADD   R1 , R2
STORE R1 , 3350
```



Micro instructions

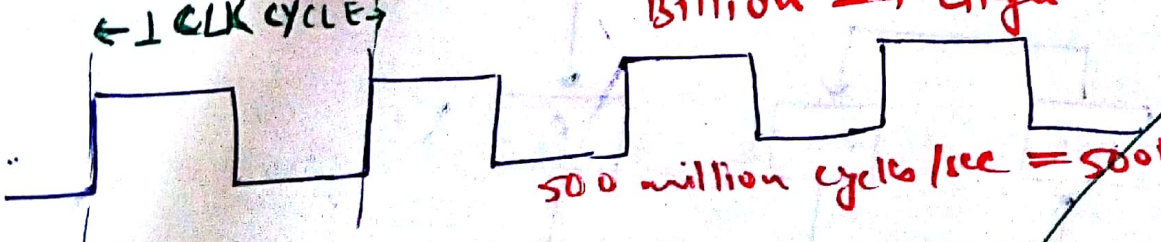
→ R1out, R2out, ADD, ALOtempIN

→ ALUtempout, R1in, END

CLOCK (CLK) CYCLE

If memory has been accessed then it will take more than 1 CLK cycle (ie more time)

Million → Mega
Billion → Giga

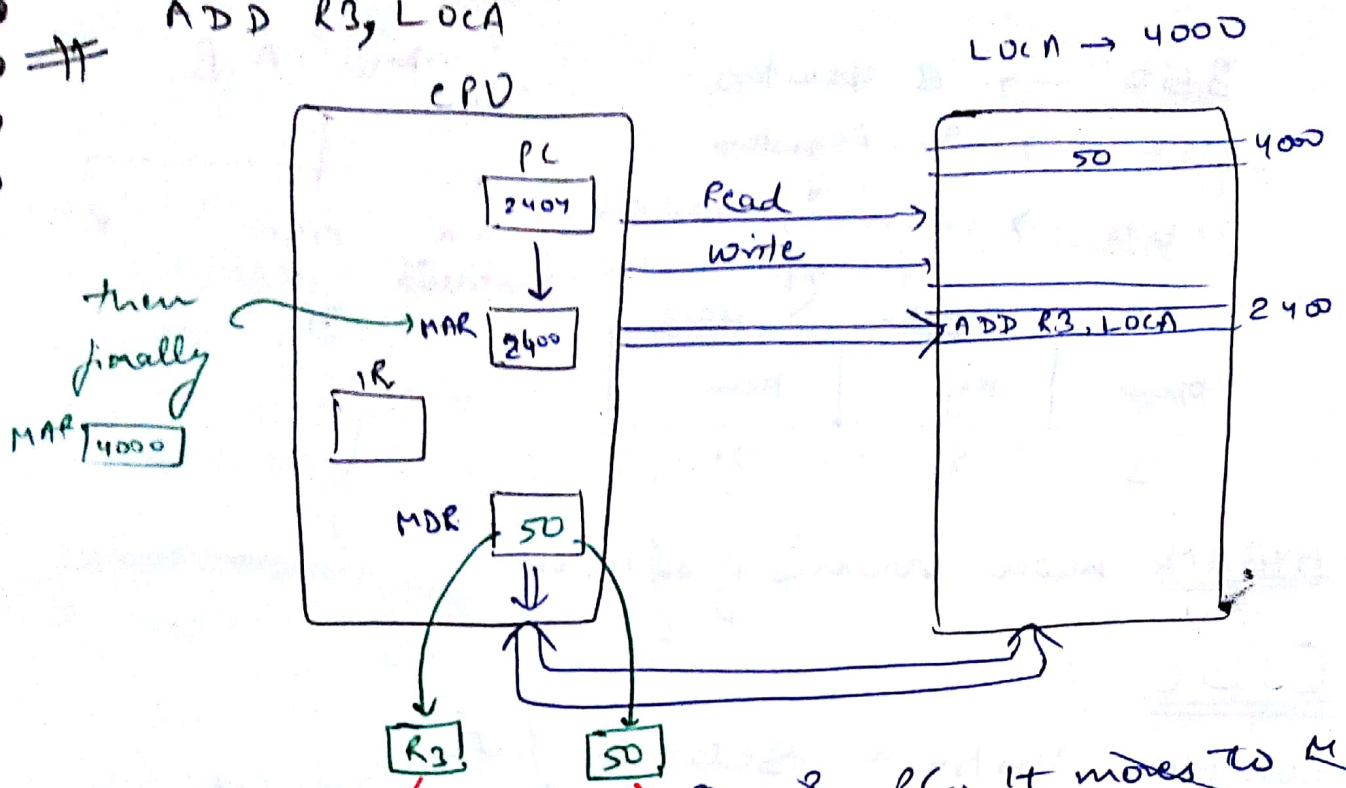


500 million cycles/sec = 500 MHz

Clock rate
↳ 1/p cycles/sec
Hertz

~~time~~ CLK cycle ~~by~~ for this will be decided by how much time it takes to get implemented.

ADD R3, LOCA



1. ~~Instruction will~~ Address ④ Out ~~on~~ PC. It moves to MAR ~~then~~ ~~thru~~ address bus it process the

2. ~~Then instruction will go to~~ MDR and then to IR ~~and then it will get executed~~ decoded

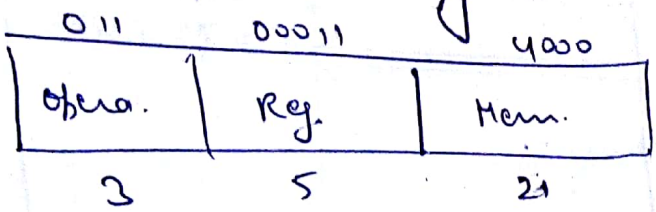
OS gives instruction to PC
PC sends the address to MAR

MAR $\xrightarrow{\text{address bus}}$ fetching the address $\xrightarrow{\text{sending the data to}}$ MDR $\xrightarrow{\text{data bus}}$ IR decode

Instruction register → decoded right here
Memory Read access again to (LOC A) 4000

→ 2 times access of Memory
① fetching the instruction
② Accessing the location

3 bits → 8 operations
5 bits → 32 Registers
21 bits → memory address



MUL A,B

micro instruc's — CISC — RISC

HYBRID newer versions of software

CISC

Complex Instruction Set Computer → lot more difficulties

It was consuming much time and was a problem since it was a repetition process that come. This was not so easy always backward compatible

now a days use

RISC

easy to decode

Reduced Instruction Set computer

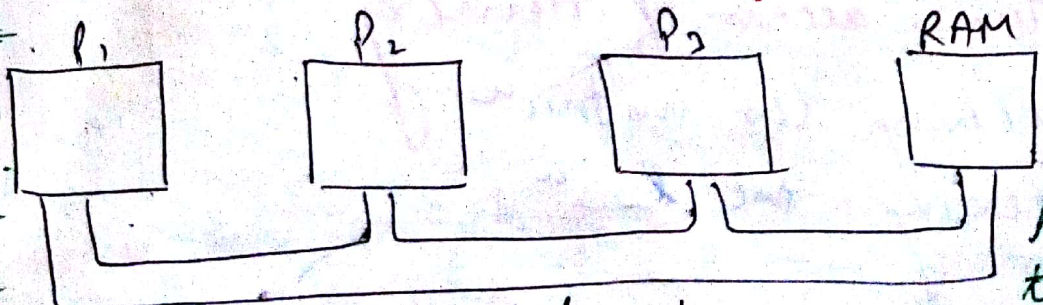
Any new machine like RISC are always backward compatible ie: all things which could be done earlier could be done now too.

- less use of memory
- More use of GPR
- More storage space inside the processor

MULTIPROCESSOR

(easier to programs)

Less time → relates to one micro-instruction directly executed on hardware



processor has to connected together

At one time only one processor can access

Pipelining:

overlapping the execn of successive instrucns.

Superscalar operation

multiple instrucn pipelines are implemented in the processor

To improve T

① Increase R
- Improve IC technology to make the ckts faster
- ↓ the amount of processing done in one basic step

② S → I

ISA

CISC

1. 1 instrucn may equate with several RISC instrucns.
2. Hard to pipeline
3. Multiple no. of cycles per instrucn
SMALL CODE SIZE

RISC

1. Instrucns are highly efficient, due to their simplicity
↓
many lines of code
2. Easy to Pipeline
3. 1 cycle per instrucn
LARGE CODE SIZE

→ 8 operation
CISC

- Goal
- 4. longer time to process instruction bcoz of complex instruction
- 5. emphasis mostly on hardware

↓

Primary goal is to complete a task in few lines of assembly instruction as possible

Big program → more storage

↓

more more

to do multiple operation within a single instruction

MULT

RISC

4. less time
5. emphasis mostly on software

↓

Primary goal is to speed up individual instruction

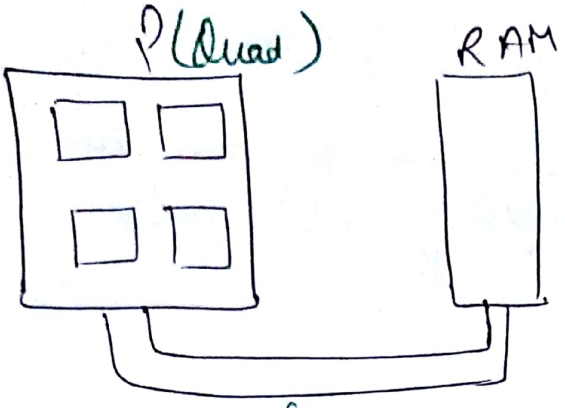
LOAD
LOAD
PROD
STORE

Performance eqn

$$\frac{\text{Time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

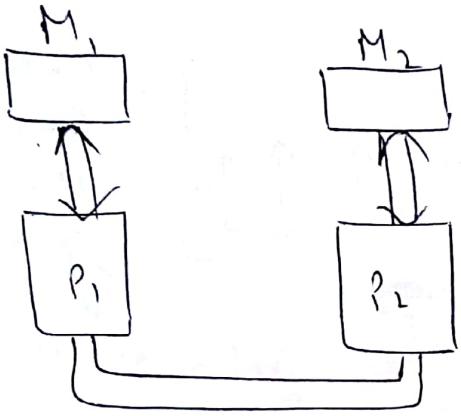
RISC → minimize no. of cycles/instruction

CISC → minimize no. of instructions per program



Multicore Single Processor

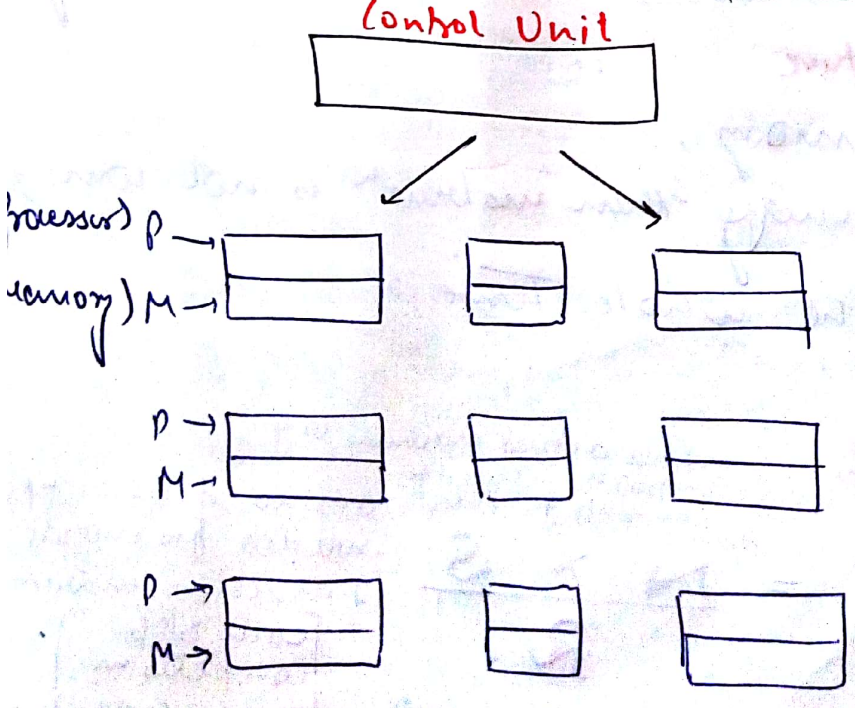
If one processor is communicating others have to wait.



Multicomputer system
A processor connected to a memory is computer

A work station is a high end computer/machine
More RAM, more speed

- One particular machine will control all network (i.e. Mainframe / Server)
  handling the network



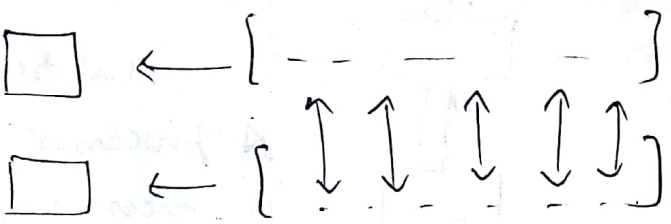
Array processors

Roll no. 45 for whole data then use of array processor

ARRAY PROCESSOR

A grid of processor ~~having~~ that perform the same sequence of instrucⁿs on diff. sets of data

VECTOR PROCESSORS



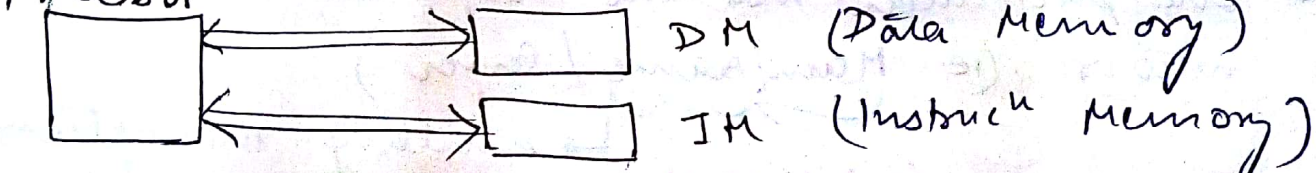
Instrucⁿ should be less given

FOR a GOOD PROCESSOR

1) Maxᵐ GPR's
2) Minᵐ access of memory
3) Multi-Processors

Performance
- Hardware design
- Instrucⁿ set
- Compiler

SPLIT MEMORY



Harvard Architecture

Before split Memory,
if data is coming then instrucⁿ is not coming
Both work the whole time.

Performance eqⁿ

processor time reqd. to execute a program that has been prepared in HLL.

$$T = \frac{N \times S}{R}$$

N → no. of actual machine lang. instrucⁿs needed to complete the execⁿ

S → avg. no. of basic steps needed to execute 1 machine instrucⁿ [each step completes in 1 clock cycle]

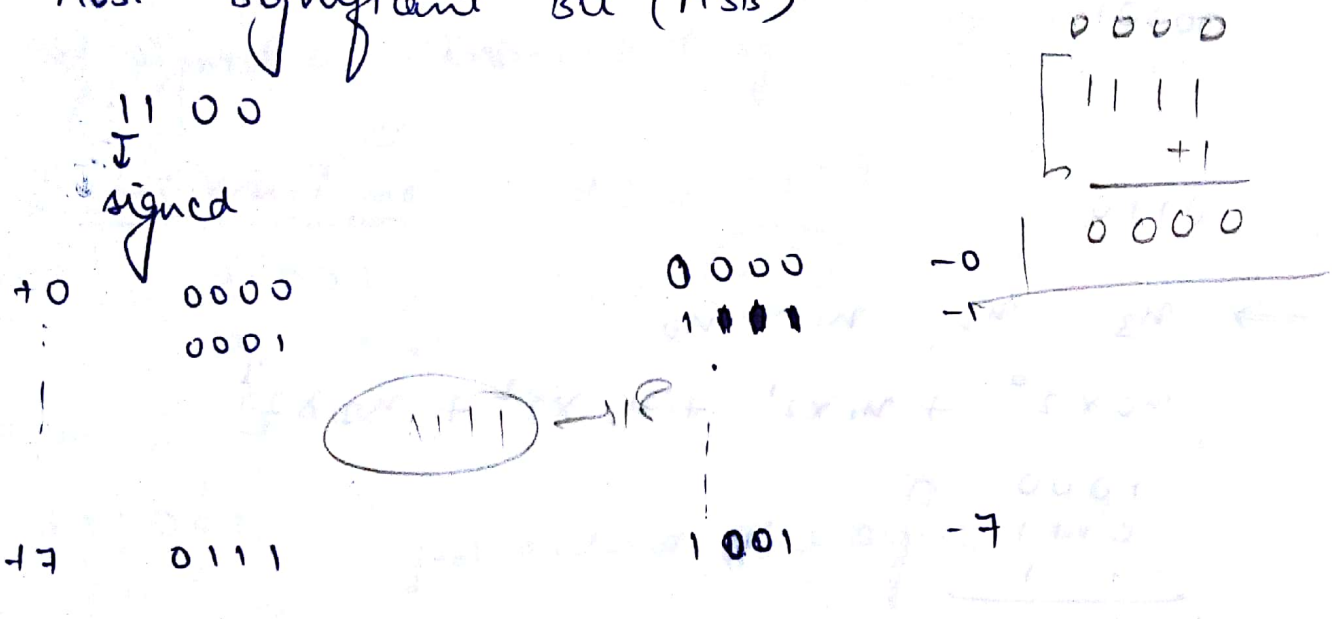
R → Clock rate

3 WAYS TO REPRESENT NEGATIVE NO.S

① Signed bit magnitude

~~The significant bit will be for sign~~

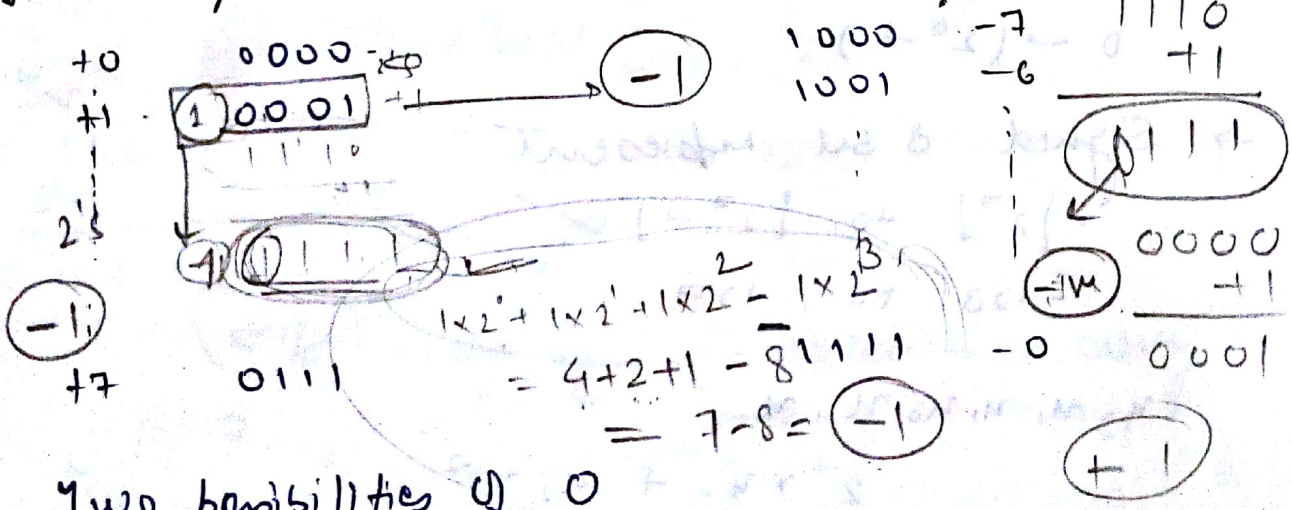
Most significant Bit (MSB)



$-(2^{n-1}-1)$ to (2^{n-1})

0010

1's complement +2 ✓ (-2)



Two possibilities of 0

So taking two's complement

Writing

0001
1110
+1
1000

Now we can represent 0 also bcoz single representation of 0

$$-(2^{n-1}) \text{ to } +(2^{n-1}-1)$$ 2's comp.

0000
0001
|
|
|
0110

$\Rightarrow$ x_3 x_2 x_1 x_0

$x_0 \times 2^0 + x_1 \times 2^1 + x_2 \times 2^2 + x_3 \times 2^3$

1000
0111
+ 1
1000

→ Unsigned 8 bit representation

$0 \rightarrow 255$

$0 \rightarrow (2^8 - 1)$

→ Signed 8 bit represent

$-[2^7]$ to $[2^7 - 1]$

−128 to 127

$x_3 x_2 x_1 x_0 x_{-1} x_{-2}$

$2^{-1} \times x_{-1} + x_{-2} \times 2^{-2}$

Unsigned

8.5

→ 1000.1

.5
+2
1.0

.68
+2
1.20

0.20

8.5

1000.1 0

0.4
×
0.0

-9 four bit using 2's complement not possible

1001 (signed no.)

$2^0 \times 1 + 1 \times 2^3 = -7$

if signed no. then but sign

1001
0110
+1
0111

- 5-bit machine

$1 \times 2^0 - 1 \times 2^3$

11001 = (-7)

$9 - 2^4 \times 1$

$9 - 16 \longrightarrow -7$

11001
-7 just extend the signed bit

- 8 bit signed no.

0000 0110

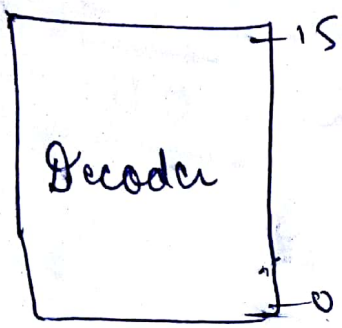
sign extension

1100
1101
10001

carry in 0 1 1 sum 1 carry out 0

1 1 0 0 0
1 1 0 1 1

4 bit → 4 full adder
8 bit → 8 full adder
n bit no. → n full adder



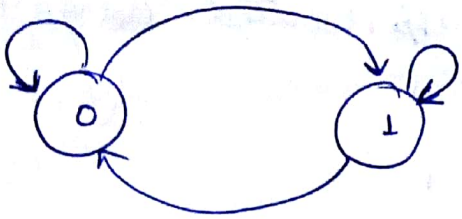
n inputs → 2^n outputs

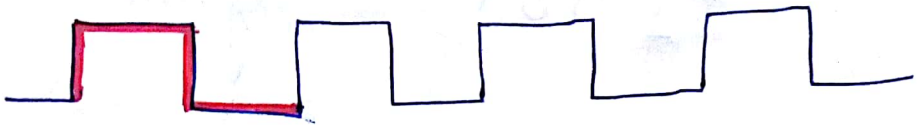
1010 (10) → would go high particular

1010
0100

1010 →
1101 ← signed bit rep.
0101 ← unsigned bit rep.

- Combinational Circuit
  current output depends only on the current input
  Just a set of logic gate implementing boolean func^ns.
- Sequential Circuit
  The current output depends on the current input and (also the previous state of input)





One clock cycle

- SR flip flop
  Not defined if the state is 1,1

S	R	Q(t+1)
0	0	No change Q(t)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	Undefined

- JK flip flop
  has defined output for 1, 1

- D

D	Q (t+1)
0	0
1	1

- Binary counter

Carry

$$\begin{array}{r} 1101 \quad -3 \\ 0101 \quad 5 \\ \hline 10010 \end{array}$$

→ Carry not overflow

7 and 6 can be represented using 4 bits but there ans can't

$$\begin{array}{r} 7 \\ +6 \\ \hline 13 \end{array}$$

0 1 1 1
0 1 1 0 0
1 0 1 → negative no. (−3)

$1\times2^0 + 1\times2^2 + 1\times2^3$
$= 8+4+1 = 13$

⇒ 32 bits →]
32 bits →] — 32 full adder

if we are having 1 bit adder

1 bit of x & 1 bit of y.

1 1 0 1 0 0 1 0 — x
0 1 0 1 1 1 0 1 — y

so we are reqd 8 full adders

if we are having 4 bit adder
so we are reqd 2 full adder

1 1 0 1 0 0 1 0
0 1 0 1 1 1 0 1

Nibble
motion that is carried forward

X − Y
adding 2's complement of Y to X

Overflow

It occurs when sign of two operands is same.

$x_{n-1}\, y_{n-1}\, \bar{s}_{n-1} + \bar{x}_{n-1}\, \bar{y}_{n-1}\, s_{n-1}$

$C_n \oplus C_{n-1}$

x y c

4 bits & 4 bits → result can't be stored in 4 bits?

Q. Multiplication of unsigned No.s.

4 bit → 1101 → 13 → multiplicand.

4 bit → 1011 (LSB) → 11 → Multiplier

1101

1101 → Partial products

0000

1101

8 bit → 10001111 → 143

Product 2 (n bit no.) is atmost 2n-bit no.

4 bit × 4 bit → 8 bit no.

Starting from LSB

→ if multiplier bit == 1
add multiplicand to the product & shift right?

→ if multiplier bit == 0
we simply shift the product to right

1011 → used LSB

↳ shifting the ~~LSB~~ used LSB so we have to check only the LSB

In a 32-bit machine →

unit ⇒ 4 byte ie (32 bits) → for multiplier / multiplicand

↓

64 bits for product (8 or 2 Registers)

Signed Multiplication

⇒

-13 1 0 0 1 1 Multiplicand (-ve)

11 0 1 0 1 1 Multiplier (+ve)

1 1 1 0 0 1 1

1 1 1 0 0 1 1

0 0 0 0 0 0 0

1 1 1 0 0 1 1

0 0 0 0 0 0

1 1 0 1 1 1 0 0 0 1 -143

11 Multiplicand (+ve)

-13 Multiplier (-ve)

2's → 1 0 1 0 1

0 1 1 0 1

proceed same as

1 1 1 1 1 1 0 1 0 1

0 0 0 0 0 0 0 0 0 0

1 1 1 0 1 0 1

1 1 1 0 1 0 1

0 0 0 0 0 0

1 1 0 1 1 1 0 0 0 1

Booth Algo

0 1 0 1 1 0 1

0 0 1 1 1 1 0 → 30

32 - 2

= 0100000 - 0000010

only 2 add's & reg. ⟷ 0 +1 0 0 0 -1 0

add 2's complement

only 2 adds & reqd.

$32 - 2$

$= 0100000 - 0000010$

$\longleftrightarrow 0\ +1\ 000\ -1\ 0$

add 2's complement of multiplicand

13 : 0 1 1 1 0

−6 : 0 −1 +1 −1 0

```
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 1 0
0 0 0 0 1 1 1 0
1 1 1 0 0 1 0
0 0 0 0 0 0
-------------------
1 1 1 1 0 1 0 1 1 0 0
```

−6 : 1 1 0 1 0

13 : +1 0 0 −1 0

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0
1 1 1 0 1 0
-------------------
1 1 1 0 1 0 1 1 0 0
```

−13 : 1 0 0 1 0

6 : 0 +1 0 −1 0

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0
1 1 1 0 0 1 0
0 0 0 0 0 0
-------------------
1 1 1 0 1 0 1 1 0 0
```

6 : 0 0 1 1 0

−13 : −1 0 +1 −1 0

```
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 0 1 0
0 0 0 0 0 1 1 0
0 0 0 0 0 0 0
1 1 1 0 1 0
-------------------
1 1 1 1 0 1 0 1 1 0 0
```

01 → +1
10 → -1
11 → 0
00 → 0

} to reduce add^{n} in booth algorithm -

00 1 0 1 1 00 1 11 0 1 0 1 1 00 → 9 add^{n}
0 +1 -1 +1 0 -1 0 +1 0 0 -1 +1 -1 +1 0 -1 0 0 → 10 add^{n}

increased (bad case)

Best Case → a long string of 1's (skipping over 1s)

Worst case → 0's & 1's alternating

Q. by booth algorithm 5 × (−6)

0 1 1 0 0 → +6
1 0 1 0 → −6
-1 +1 -1 0

		Multiplicand	Multiplier
1011	2's	0101	
		0000	+1 0 -1 0
Shift		0000	0 -1 +1 -1
		1011	
Add		1011	0 -1 +1 -1
Shift	Signed bit	1 1011	1 0 -1 +1
		0101	
Add	1	0010	1 0 -1 +1
Shift		0001	0 1 0 -1
		1011	
Add		1100	0 1 0 -1
Shift		0110	0 0 1 0
		100010	

0 1 0 1 1 0 -1 Shift (3)

Bit-pair recoding (more faster)

1 1 1 0 1 0
0 0 -1 +1 -1 0 ← booth algo
0 -1 -2 → 2 shifts in product
(-2+1) (-2+0)

→ 13
-6

2's | 0 1 1 0 1 |
1 1 0 1 0

1 0 0 1 1

⇒

0 1 1 0 1
0 +1 +1 -1 0

0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 1 1
0 0 0 0 1 1 0 1
1 1 1 0 0 1 1
0 0 0 0 0 0

1 1 1 1 0 1 1 0 0 1 0

→

0 1 1 0 1
0 -1 -2 → shift by two

1 1 1 1 1 0 0 1 1 0
1 1 1 1 0 0 1 1
0 0 0 0 0 0 0
1 1 1 1 0 1 1 0 0 1 0

0 1 1 0 1
-2
2's
1 0 0 1 1
1 0 0 1 1
1 0 0 1 1 0

Slide 44

Carry Save add^n (Set of 3) Summands

A B C D E F
S1 C1 S2 C2
S3 C3
S4 C4
↳ Add

1 0 1 1 0 1
× 1 1 1 1 1 1

1 0 1 1 0 1 A
1 0 1 1 0 1 B
1 0 1 1 0 1 C
1 0 1 1 0 1 D
1 0 1 1 0 1 E
1 0 1 1 0 1 F

BOOLEAN ALGEBRA

- Precedence Order → NOT, AND, OR

- laws →

Identity	$1 \cdot x = x$	$0 + x = x$
Null	$0 \cdot x = x$	$1 + x = x$
Idempotent	$xx = x$	$x + x = x$
Inverse	$x\bar{x} = 0$	$x + \bar{x} = x$
Absorption	$x(x+y) = x$	$x + xy = x$
De-Morgan's (easy way of finding the complement of a given func^n)	$(\overline{xy}) = \bar{x} + \bar{y}$	$(\overline{x+y}) = \bar{x}\bar{y}$

Q. To find complement of a given
Replace each variable → complement
change all ANDs → ORs &
all ORs → ANDs

$f(x,y,z) = (xy) + (\bar{x}z) + (y\bar{z})$ → SOP

$\bar{f} = \overline{(xy) + \bar{x}z + (y\bar{z})}$

$= (\overline{xy})\,(\overline{\bar{x}z})\,(\overline{y\bar{z}})$

$= (\bar{x} + \bar{y})\,(x + \bar{z})\,(\bar{y} + z)$ → POS

⇒ Easy to convert a func^n to SOP using its truth table

Boolean func^n

Eg: $f(x,y,z)$

xyz	$x\bar{z} + y$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	1
1 0 1	0
1 1 0	1
1 1 1	1

- $\bar{x}y\bar{z} + \bar{x}yz + x\bar{y}\bar{z} + xy\bar{z} + xyz$

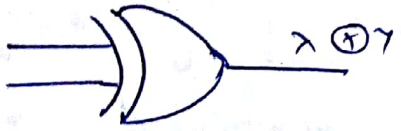
- $\bar{x}y + x\bar{z} + xyz$

- LOGIC GATES

Integrated circuits contain collection of gates suited for a particular purpose.

→ AND OR NOT

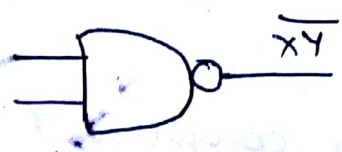
→ XOR (Exclusive OR)

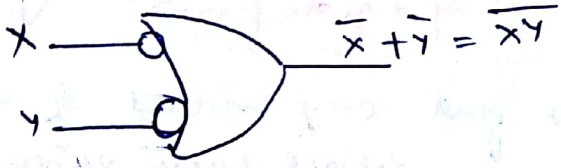


x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

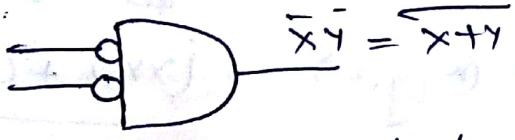
→ NAND

Universal gates





→ NOR

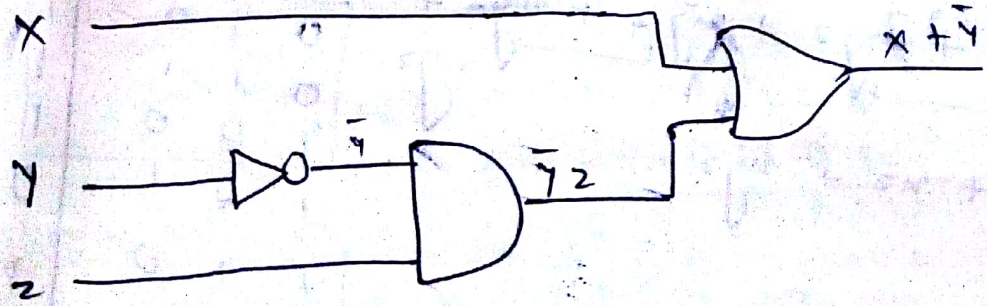


Any Boolean function can be constructed using only NAND or only NOR gates. They are inexpensive to manufacture.

- LOGIC COMBITIONAL CIRCUITS CLC

Designed ckt that implements the Boolean func^n

$F(x,y,z) = x + \bar{y}z$



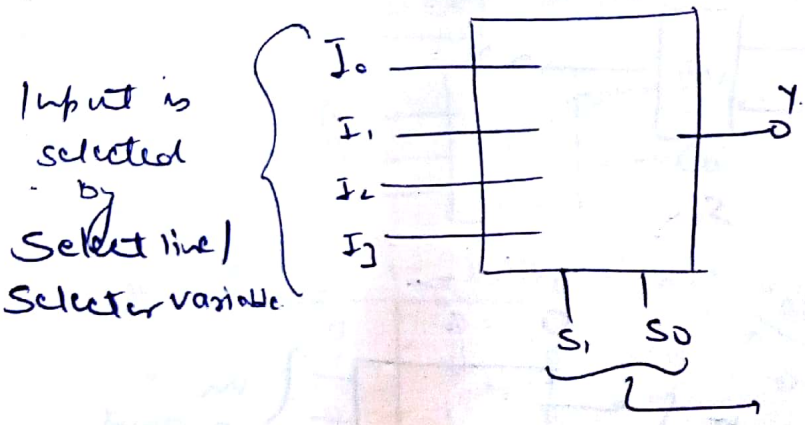
→ Multiple inputs + Multiple outputs
CLC produce a specified output at the instant when input values are applied

eg. Half adder
Full adder
Decoder
Multiplexer

① MULTIPLEXER

that selects binary information from one of many inputs and directs it to o/p line

→ DATA SELECTOR

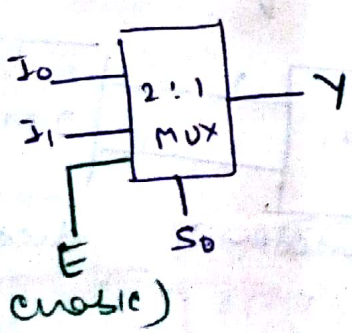


2^n inputs
n selectors
1 output

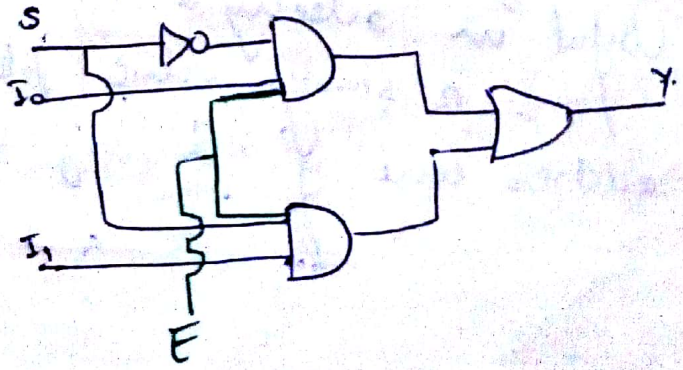
$2^n = 2^S$

0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

2:1 MUX



E	S	Y
0	X	0
1	0	I_0
1	1	I_1



$Y = E.(\bar{S}.I_0 + S.I_1)$

MUX TREE

To obtain higher order using lower order

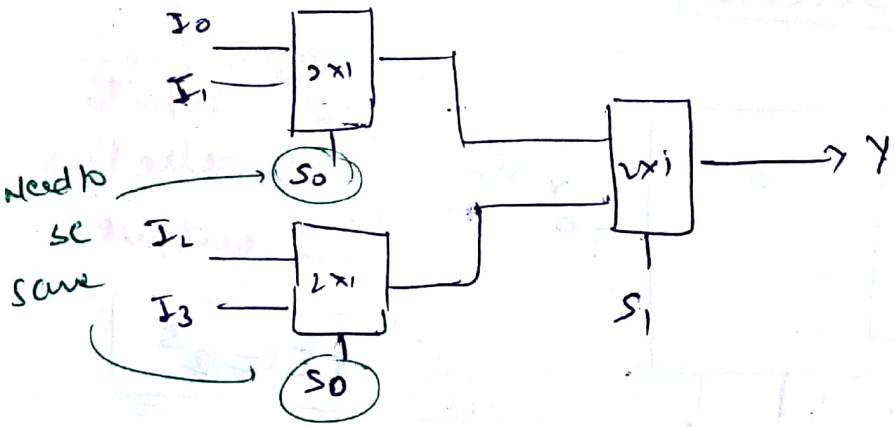
→ 4×1 using 2×1

n = 4

$\frac{4}{2} = 2$

$\frac{2}{2} = 1$

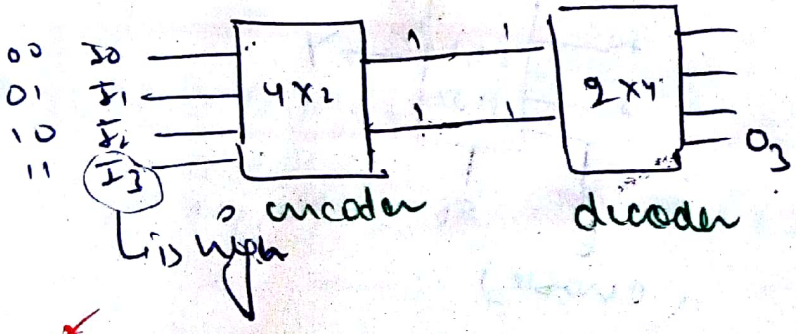
$2 + 1 = 3$ → 2×1



② DECODER

n input, n×m decoder, m output

function of decoder is opp of encoder

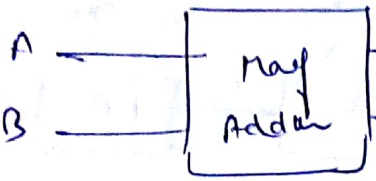


$m = 2^n$

doubt

Usual in selecting a memory location
acc. a binary value placed on the
address line of memory bus

③ HALF ADDER

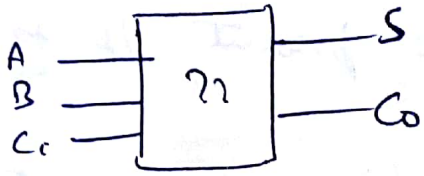


$A \oplus B$

$A \cdot B$

A	B	S	C_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

④ FULL ADDER



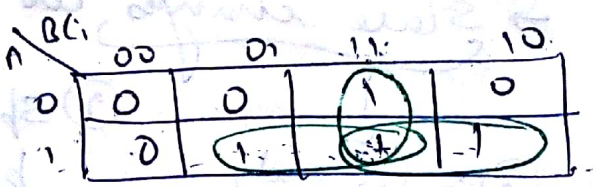
$A \oplus B \oplus C_i$

$= A \cdot B + C_{in}(A \oplus B)$

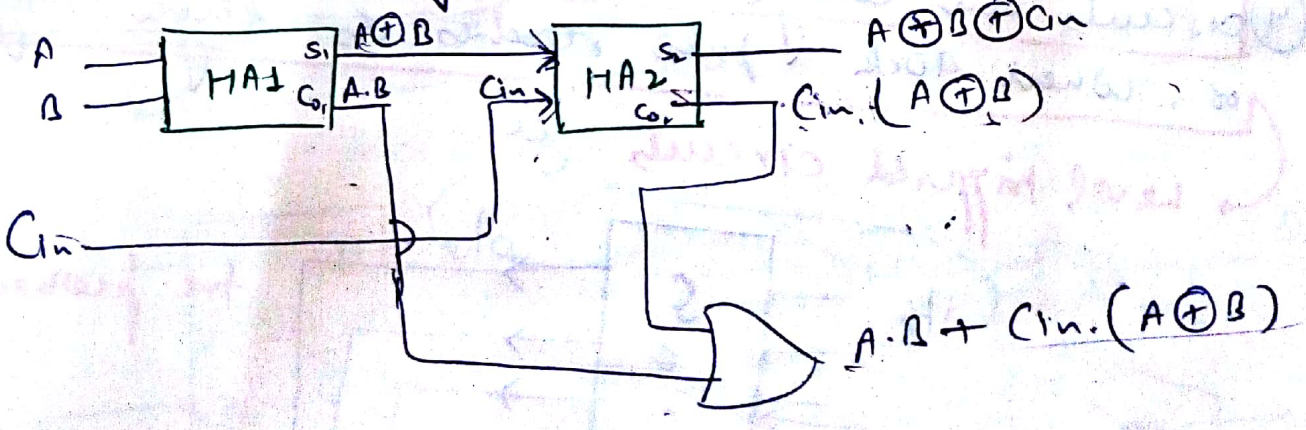
$BC_i + AB + AC_i$

thru K-map

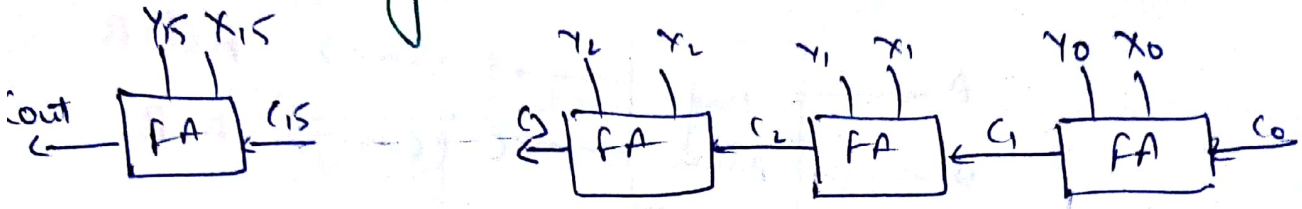
A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Full adder Using Half Adder



Ripple Carry Adder (RCA)



Carry-bit ripples from one adder to next

↳ this configuration → (RCA)

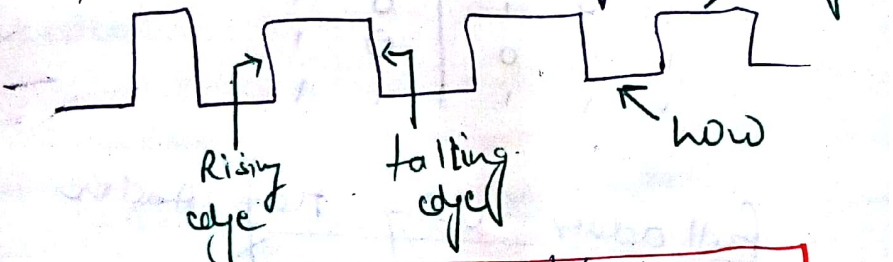
• SEQUENTIAL CIRCUITS

o/p depends on the present i/p as well as prev. o/p or o/ps

① State changes are controlled by clocks

③ occur only when clock ticks

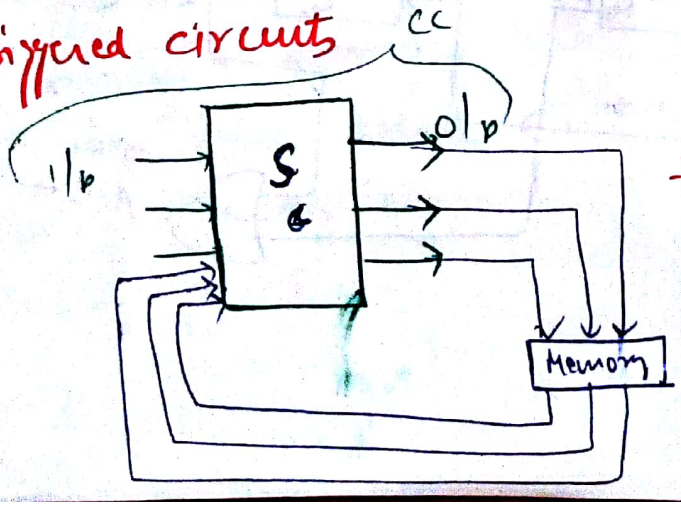
② special circuit that sends electrical pulses thru a ckt. produce electrical waveforms



edge triggered

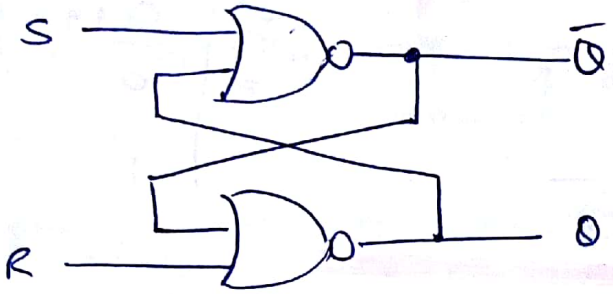
④ circuits can change state on rising / falling edge or when clock pulse reaches its high-voltage

↳ level triggered circuits



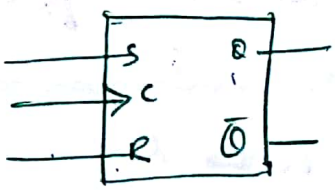
the feedback

① SR flip flop set / reset



S	R	Q(t+1)
0	0	Q(t) no change
0	1	0 reset to 0
1	0	1 set to 1
1	1	Undefined

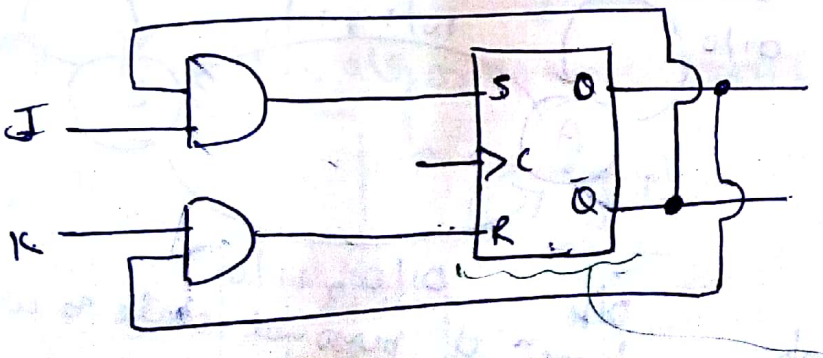
in real 3 inputs



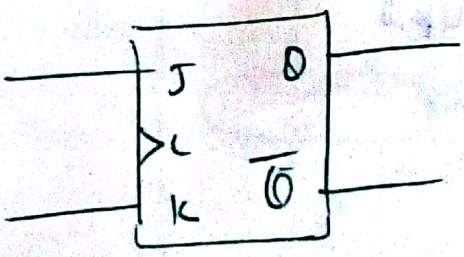
S	R	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	—
1	1	1	—

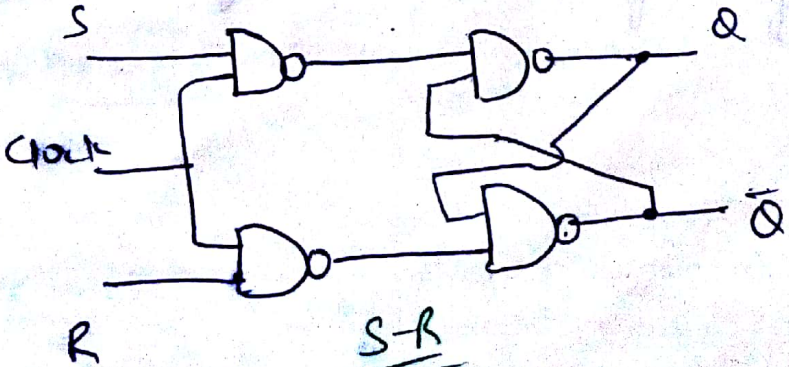
modified to provide a stable state when both inputs = 1

② JK flip flop Jack Kilby



J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q̄(t)

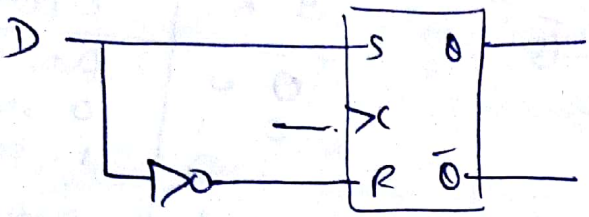




S-R

③ D - flip flop

fundamental ckt of computer memory



D	Q(t+1)
0	0
1	1

Behaviour of sequential ckts

① can be explained using characteristic table / FSMs

finite state machines

consist of a set of nodes that holds the states of the machine & a set of arcs that connect the states.

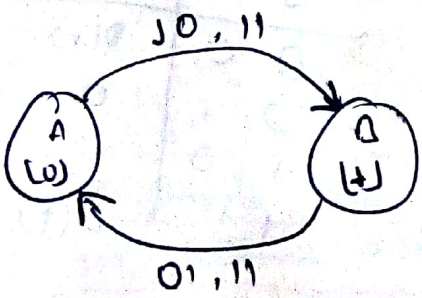
(FSMs) → only in the way the o/p is generated

Moore

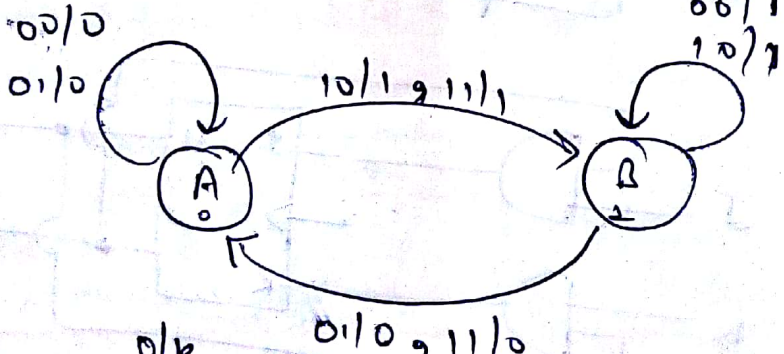
place outputs on each node

Mealy

present their outputs on the transitions

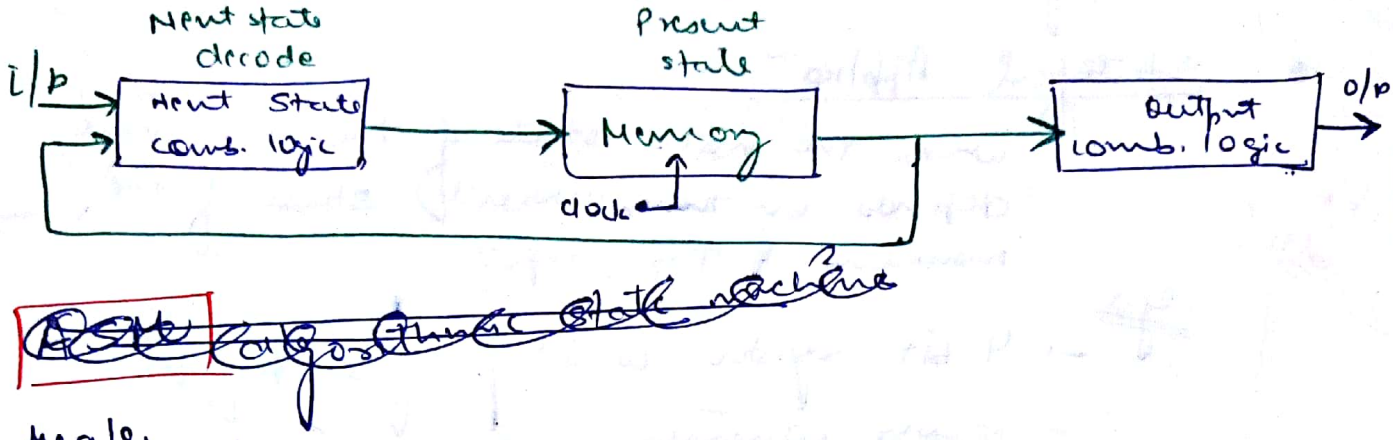


fun^n of Present state only



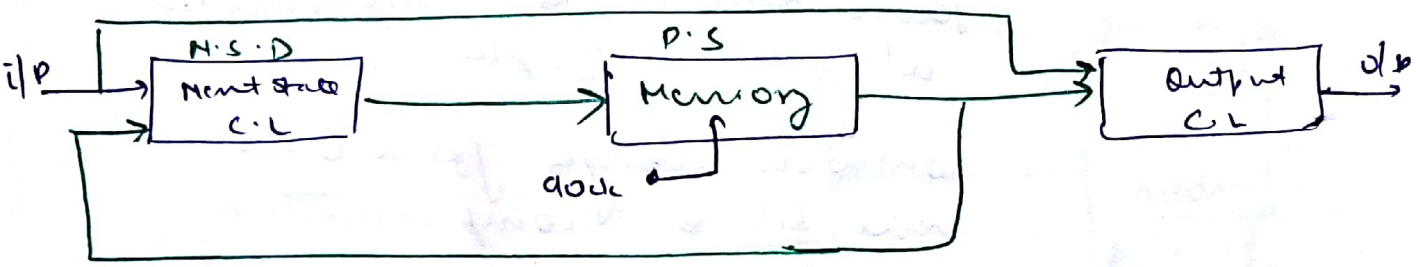
o/p fun^n of present state as well as i/p.

Moore



~~ASM (Algorithmic state machine)~~

Mealy

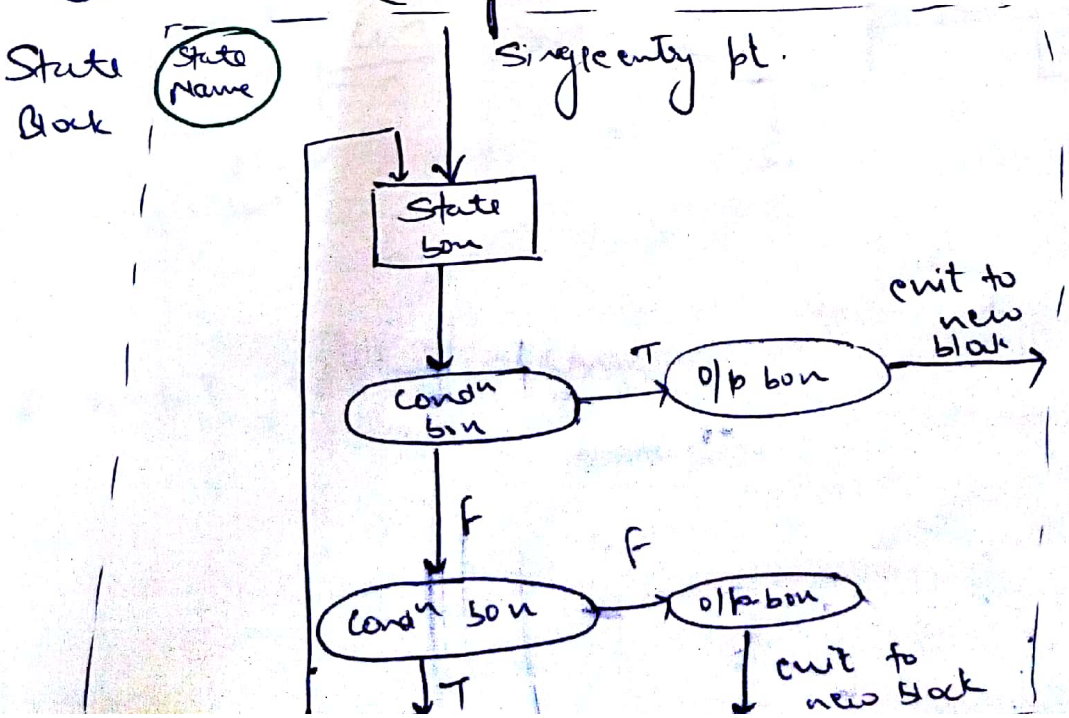


→ They do not address the intricacies (complexities) of timing very well.

→ Introduce numerous signals in regd. to advance a machine from one state to another.

↓ due to this

② ASMs (Algorithmic state machine)



ASM for microwave oven →

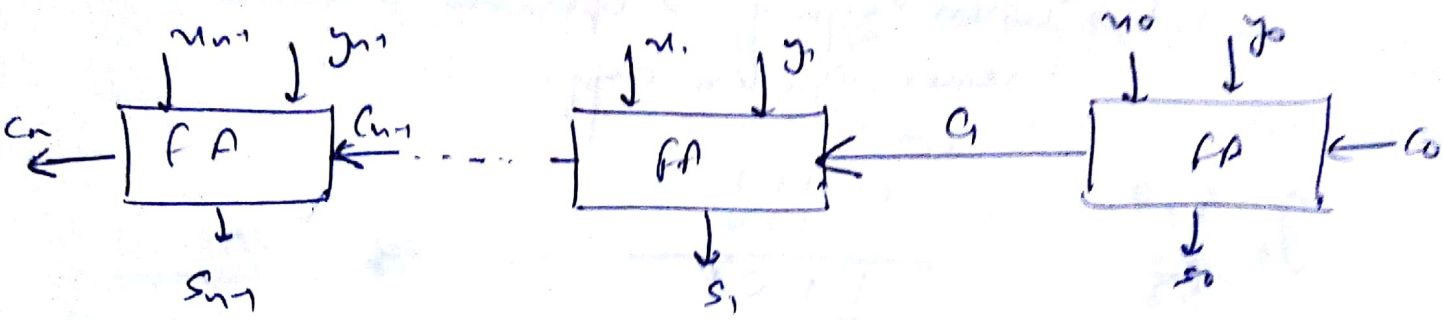
- **Stateful Applica~**

  where the next state of the machine depends on the current state of the machine & the input.

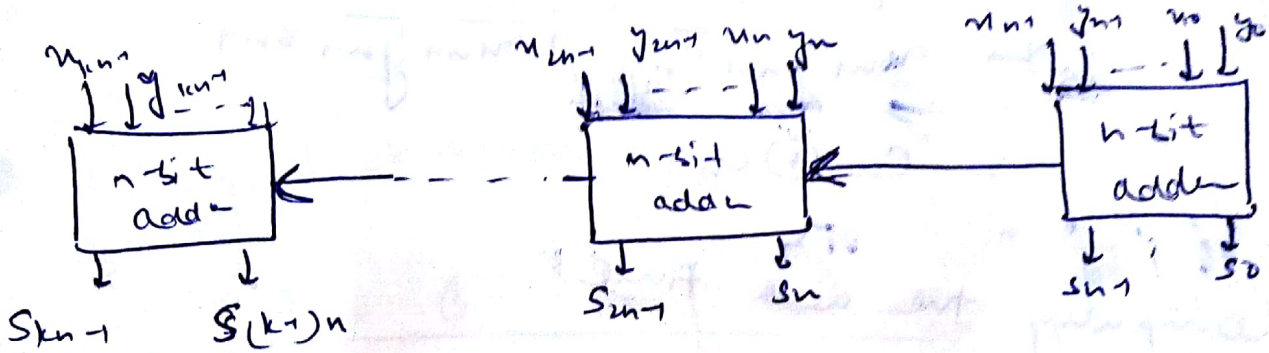
see slides

Eg:
- 4 bit register consisting of D flip-flops
- binary counter
  - lower order bit is complemented at each clock pulse
  - whenever changes from 0 → 1 next bit is complemented.

- n - bit adder / n-bit ripple carry adder



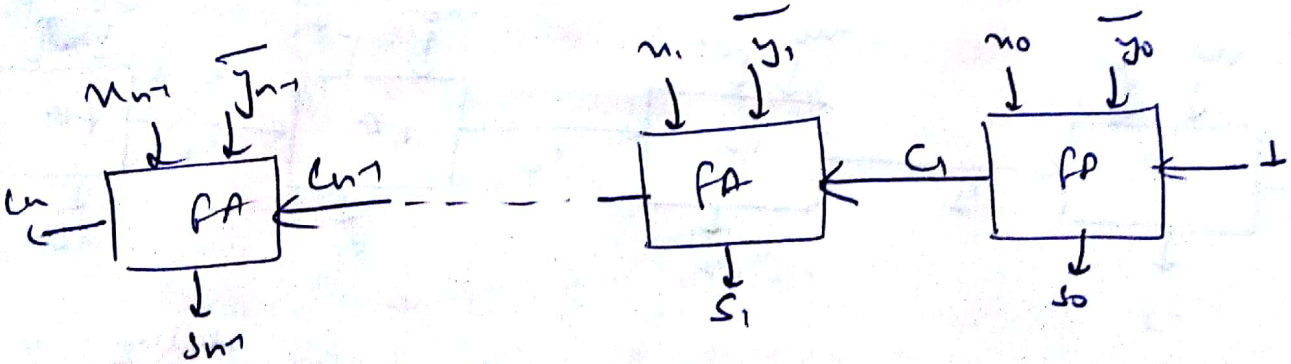
- k n-bit adder / Blocked Ripple Carry Adder



- n-bit subtracter

$X - Y = X + \bar{Y} + 1$

↳ add 2's complement of Y



- n-bit adder / subtracter

↳ slide

- Detecting Overflows
  - ↳ when sign of 2 operands is same & result sign is diff.

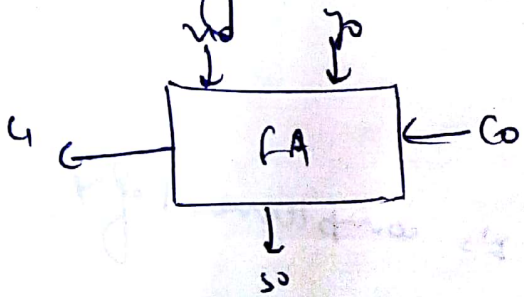
```
 7          +ve          0 1 1 1
+6          +ve          0 1 1 0
---                      -------
13                       1 1 0 1
                         ↓
                        -ve no.
```

Condn for overflow

↳ $x_{n-1} y_{n-1} \bar{s}_{n-1} + \bar{x}_{n-1} \bar{y}_{n-1} s_{n-1}$

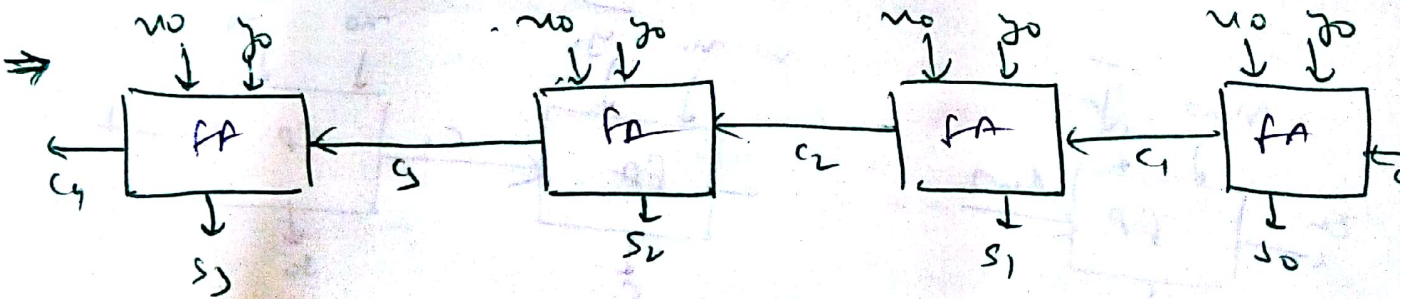
→ $c_n \oplus c_{n-1}$

- Computing the add time



C_1 available after 2 gate delays

S_1 " " 1 " "



$S_0 \longrightarrow$ 1 gate delay

$S_1 \longrightarrow$ 3 " "

$S_2 \longrightarrow$ 5 " "

$C_1 \longrightarrow$ 2 gate delays

$C_2 \longrightarrow$ 4

$C_3 \longrightarrow$ 6

for a n bit adder

$S_{n-1} \rightarrow (2n-1)$ gate delays | $C_n \rightarrow 2n$ gate delays

- Fast adder

for a full adder →

$S_i = x_i \oplus y_i \oplus c_i$

$C_{in} = x_i y_i + x_i c_i + y_i c_i$

$= \underbrace{x_i y_i}_{G_i} + c_i \underbrace{(x_i + y_i)}_{P_i}$

$C_{i+1} = \boxed{G_i + P_i c_i}$

G_i & P_i can be computed in one gate delay.

C_{i+1} = generate func^n + propagate func^n · (carry)

- Carry lookahead

A	B	C_n	C_0
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$A \oplus B = 1$, $C_n = 1$

$A \cdot B$

$C_0 = \underbrace{A \cdot B}_{G} + \underbrace{(A \oplus B)}_{P} \cdot C_n$

$A \cdot B$ → carry generator

$A \oplus B$ → carry propagator

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_0 C_0$$

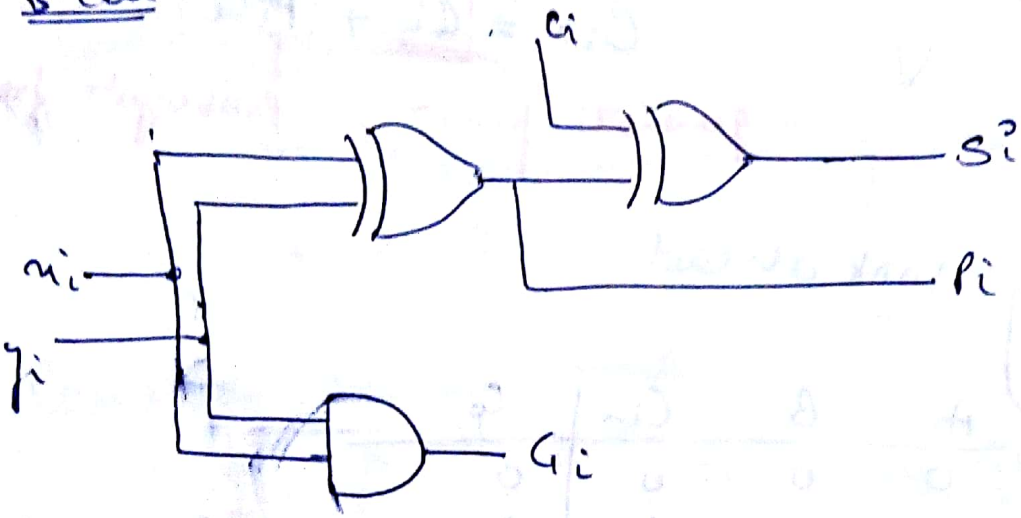
→ 1 gate delay for P_i & G_i

→ all carries can be obtained by 3 gate delays
(↓ x, y, c_0 are applied)

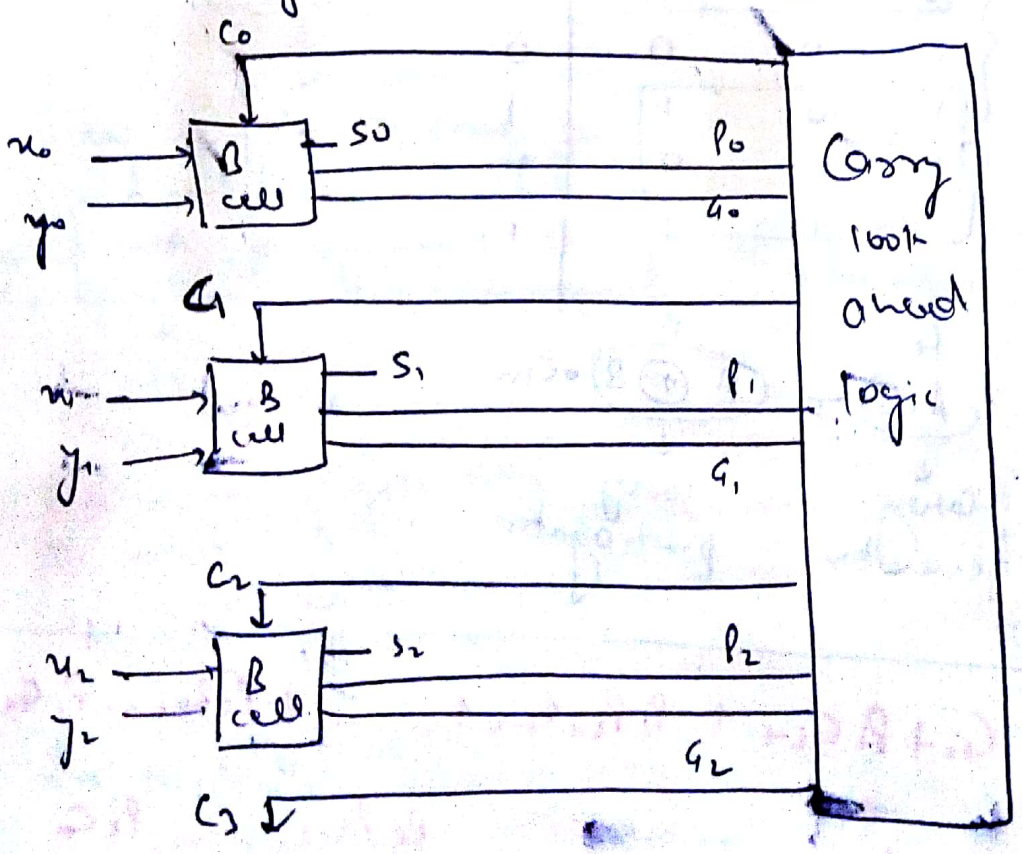
→ 2 gate delays in the AND-OR ckt for c_{i+1}.

∴ S = XOR of inputs = $A \oplus B \oplus C$

B-cell



4 bit Carry Look Ahead Adder



fan-in → Max. no. of inputs that a logic gate can accept.

Slide 19 doubt

Bit-pair recoding of Multiplier

→ 0 0 0

0 0 booth

0 pairing

00	0
01	+1
10	-1
11	0

→ 1 0 0

-1 0

-2 magnitude

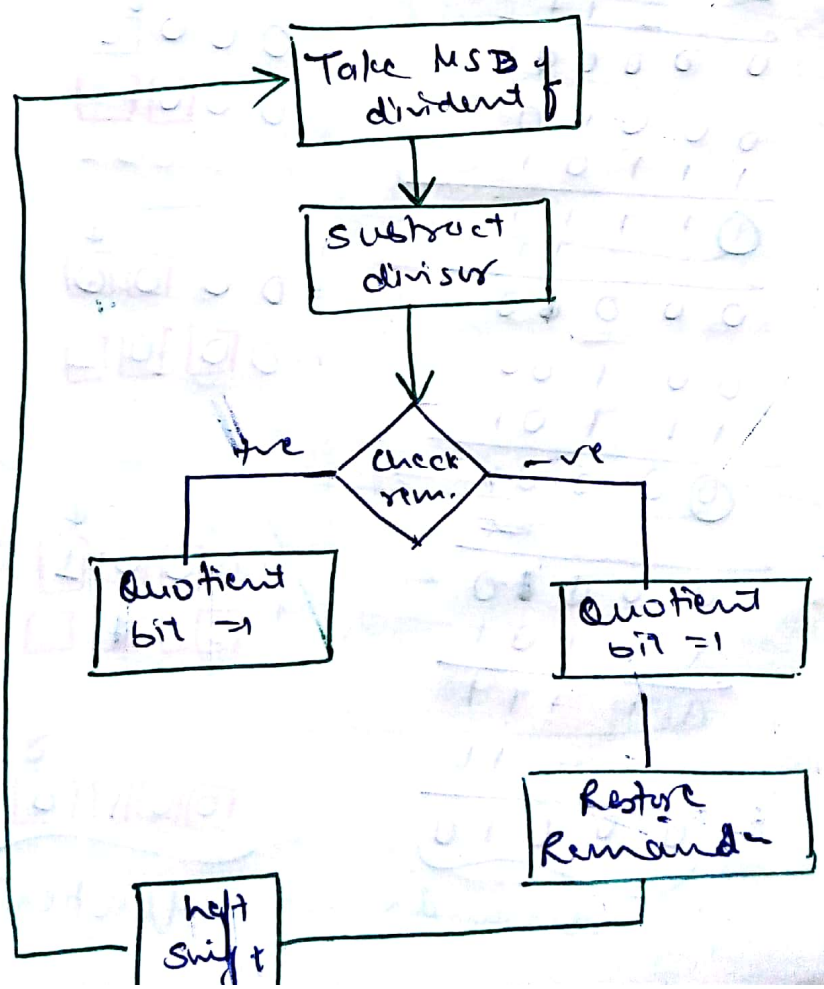
→ (0 1 0 1)$_B$ —2's comp→ 1 0 1 1

-2 (x 0 1 0 1) —2's→ 0 1 1 1

```
  1
 1 0 0 1 1
 1 0 0 1 1
-----------
 1 0 0 1 1 0
```

DIVISION

Subtracting the divisor → adding the 2's complement

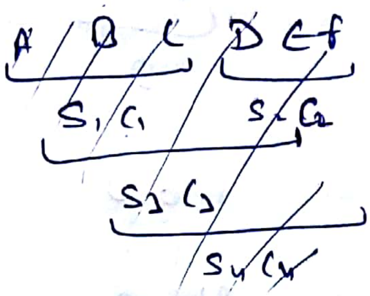


Control Sequencer

will make sure that ~~stem~~ divident is added to restore the remainder

Eg

```
        021
13 ) 274
    -13
    ---
    -11
    +13
    ---
     27
     26
    ---
     14
     13
    ---
      1
```



Eg

Divisor 11, Divident 1000

```
        10
 11 ) 1000
       11
      ---
       10
```

11101 (2's of divisor)

Initially — ignore carry

Step	A (remainder)	Q (quotient)
Initially	0 0 0 0 0 / 0 0 0 1 1	1 0 0 0
Shift	0 0 0 0 1	0 0 0 [ ]
Sub	1 1 1 0 1 → 1 1 1 1 0	
Restore	+ 1 1 → 0 0 0 0 1	0 0 0 0
Shift	0 0 0 1 0	0 0 0 [ ]
Sub	1 1 1 0 1 → 1 1 1 1 1	
Restore	+ 1 1 → 0 0 0 1 0	0 0 0 0
Shift	0 0 1 0 0	0 0 0 [ ]
Sub	1 1 1 0 1 → 0 0 0 0 1	
~~Restore~~		0 0 0 1
Shift	0 0 0 1 0	0 0 1 [ ]
Sub	1 1 1 0 1 → 1 1 1 1 1	
Rest	+ 1 1 → 0 0 0 1 0	0 0 1 0

0 0 0 1 0 — remainder

0 0 1 0 — quotient

Divident 7 = 0111

4 = 0100 $\xrightarrow{2's}$ 1100

$$\begin{array}{r} 0100 \overline{)0111} \\ 0100 \\ \hline 1011 \\ 1 \\ \hline 1100 \end{array}$$

2's =

	A	B	
	0000	0111	
Shift	0000	1110	shift
Sub.	1100		
-ve →	1100		
Restore	0100	1110	
A →	0000		
shift	0001	1100	shift
sub	1100		
-ve	1101	1100	
Restore	0100		
A →	0001		
shift	0011	1000	shift
sub	1100		
-ve	1111	1000	
Restore	0100		
A →	0011		
shift	0111	0000	
sub	1100		
	0011	0001	= q.

+ve

↓

Remainder

Non-Restoring Division

If A = 0 → shift, sub; 1 → shift, add; 0 → quo = 1; 1 → quo = 0

11101 (divisor)

		A	Q
A →		00000	1000
		00011	
shift		00001	0000
	sub.	11101	
A →		11110	0000 quotient
(shift)		11100	0000
	Add.	00011	
		11111	

Dividend = 1000

Divisor = 00011 —2's→ 11101

	A	Q
Initially A	00000	1000
Shift	00001	000_
Sub	11101	
A →	11110	0000
Shift	11100	000_
Add	00011	
A →	11111	0000
Shift	11110	000_
Add	00011	
A →	00001	0001
Shift	00010	001_
Sub	11101	
A →	11111	0010
Restore	00011	
	00010	

00010 — Remainder

0010 — Quotient

Dividend = 13 → 01101

Divisor = 3 → 00011

2's → 11101

3) 13

Initially

A →	00000	01101
Shift	00000	1101–
Sub	11101	
	11101	1101 0
A →	1 1101	101 0–
Shift	11011	
Add	00011	
	11110	1010 0
A →	1 1110	010 0–
Shift	11101	
Add	00011	
	00000	0100 1
A →	00000	100 1–
Shift	00000	
Sub	11101	
	11101	1001 0
A →	1 1101	001 0–
Shift	11011	
Add	00011	
	11110	00100 quotient
A →	1 1110	
Restore	00011	
	00001	remainder

finally add → divisor since we get the quotient so add without shifting..

FRACTIONS

If we r having a float, how it is stored in memory-

~~fraction~~

$0 \leq V(b) \leq 1 - 2^{-32}$

↳ for 32 bit machine

456

↳ Scientific notation 4.56×10^2

n

↳ S.N $m_1 . m_2 m_3 m_4 \times b^{\pm e}$

mantissa

⟹ HLL -0.75

Binary -0.11

Normalized form -1.1×2^{-1}

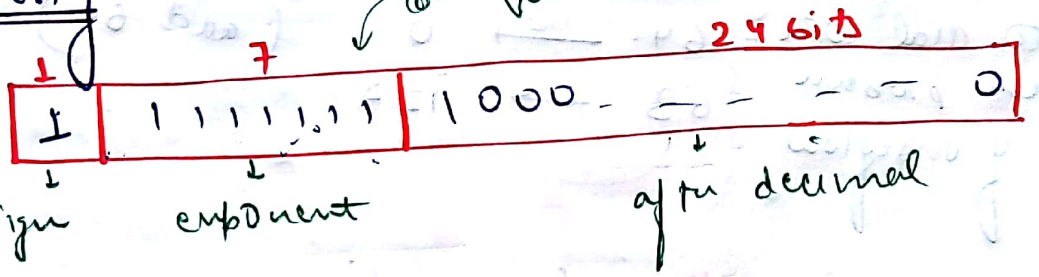
shift left/right till I get 1 ge digit.

we already know that there is 1 @ut in normalized form so we need not to store 1 & Store value after decimal pt

need not to store 2 (since we know there is 2 @ut) need to store exponent

2's complement value

Storing



⟹ Excess 3 coding

to get rid of -ve exponent

∵ it is easy to compare with +ve exponent.

for 7 bit

range $= -2^{7-1} \longrightarrow -(2^{7-1}-1)$

$-64 \longrightarrow 63$

↳ highest +ve

So we will add 63 to exponent

$\therefore 2^{-1} \longrightarrow 2^{-1+63} = 2^{62}$

now we will store this in the 7 bit exponent place

∴ 8 bit then add +127 to exponent

Underflow

Overflow

7-bit

range → -64 to 63

$< 2^{-65}$ ↓ add 63 → 2^{-2} → underflow

$> 2^{63}$ ↓ add 63 → 2^{126} → overflow

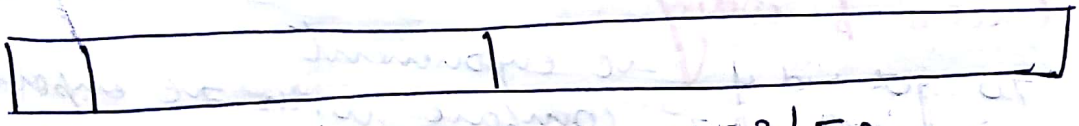
EXCESS NOTATN

to deal with the problem of underflow

-64 → 0 (add 64)

63 → 127

* IEEE Notatn → (all machines use this standard)



Sign | 8 | 11 | 23/52

8 → -128 → 127

↓

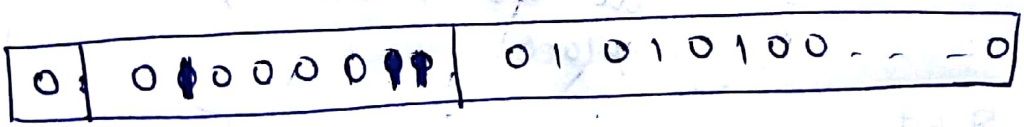
to add 128

eg:

21.25

→ 010101.01

1.010101 × 2^{+4} → $2^{4+127} = 2^{131}$



eg. decimal no. ??!

1 0001 0101 1 010110 — 0

↓ -ve, ↓ 2^{21}, after decimal

Ad

-1.101011×2

Excess 8

$-8 \to 7$
$0 \to 15$

Excess 7

$-8 \to 7$
$(-1) 0 \to 14$
$-7 \to 7$

Excess 6

$-8 \to 7$
$(-2) 0 \to 13$
$-6 \to 7$

for 4 bit range

$-2^{n-1} \to (2^{n-1} - 1)$
$-8 \to 7$

$2^{-128} \to 2^{127}$ ∴ $p = 128$

but acc. to IEEE standards → $p = 127$

add 127

$-128 \to 127$
$(-1) 0 \to 254$
$0 \to 254$
$1 \to 254$

available range → true: $-126 \to 127$

IEEE reserves 0 & 255 for some special purpose
└ will read later

How to represent 0 & ∞

Sign	Exponent	Mantissa	
0	00000000 (reserved)	00 - - - 0	+ve zero
1	00000000	00 - - - 0	-ve zero
0	11111111 (reserved)	00 - - - 0	+ve infinity
1	11111111	00 - - - 0	-ve infinity
	11111111	Non-zero	NaN

$a.10^x * b.10^y = (a.b)\,10^{x+y}$

acc. to IEEE

$a.10^{(x+b)} * b.10^{(y+b)} = (a.b)\,10^{(x+y+2b)}$

instead it will be $x+y+b$

Q adding 2 no.s

$2.3516 \times 10^3 + 4.258 \times 10^2$

$2351.6 + 425.8$

↓ binary form

↓ normalized

$1.\_\_\_ \times 2^{-}$

↓ excess b

S E+P M

check for (E+P) of both the no.s then shift it

⇒

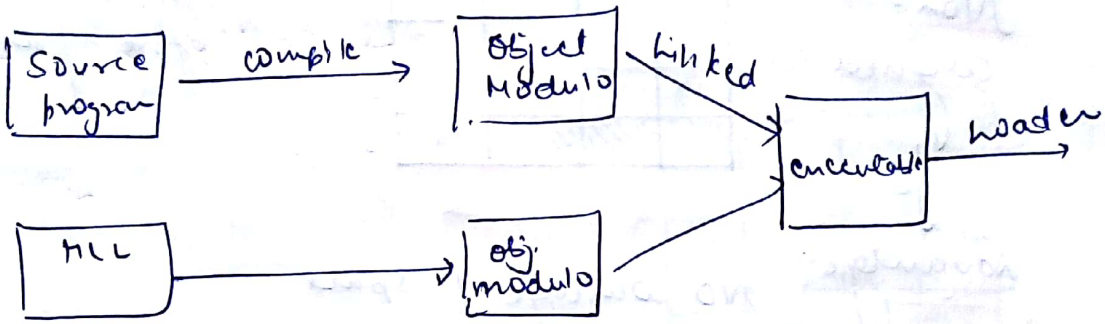
1.011
1.011

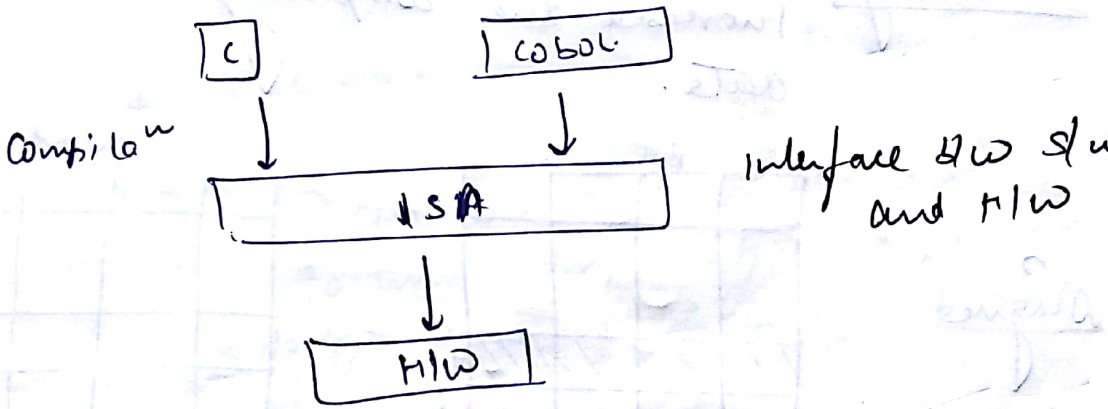
Guard bits

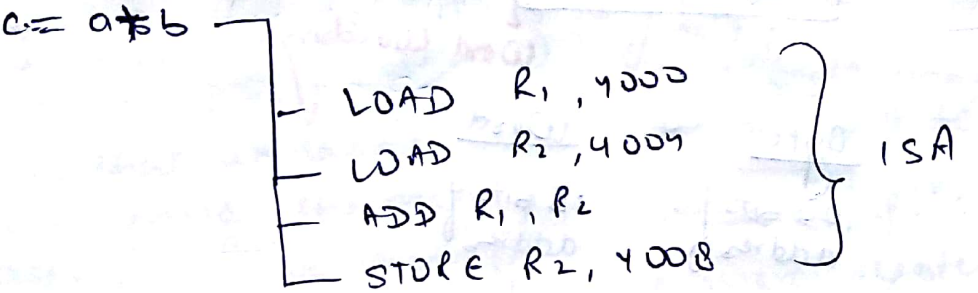
if any of GB = 1 then LSB = 1

Instruction Set Architecture (ISA)

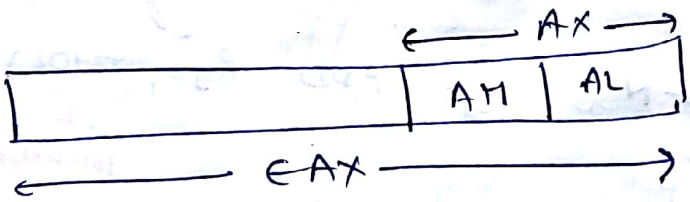
Object Module → when we compile source program







⇒ 32 bit



→ GPR (General Purpose registers)

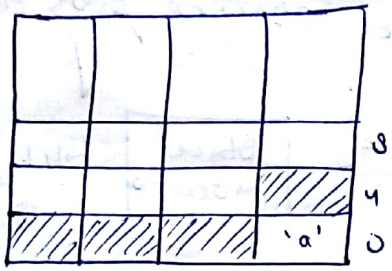
If 8 bit then stored in AL
16 bit " " " AX
32 bit " " " EAX

① Memory Model

~~Intel~~ Intel doesn't share its ISA with anyone

Spark share its ISA

Non-aligned

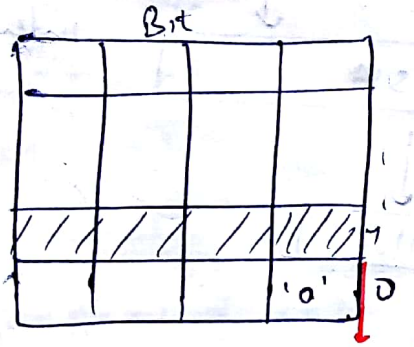


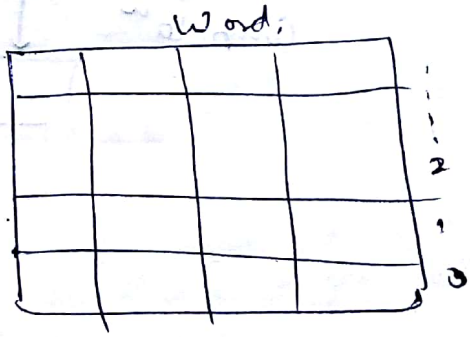
to store an int, store from 1 → 4

Advantage → NO wastage of space

Disadvantage → Increased the complexity to extract the data.

Aligned





	Byte	Word
1MB	20 bits of address	18 bits of address

↓

Dis: Can't access every byte

Adv: reduce no. of address bits

ADD Reg, #N

ie ADD R1, #102

↓ Immediate value

Little Endian / Big Endian

Start storing from lowest address

integer is always at least significant byte

0	0	0	8 20
			4 'L'
'U'	'M'	'A'	'R'

] ← int ⟶ [

Start storing from highest address.

int is always at least significant byte

0	0	0	8 20
'L'			4
'R'	'A'	'M'	'U'

Little Endian ⟶ Big Endian

flip data

character flip & don't flip int.

90-time complexity yes ∴ in

today's machines LE is available

I need more no. of registers

Adv

so that we can minimize the memory access (if we can store more things in register then no need to access memory).

Disadv

- we will be needing space for registers
- more registers so more space in my instruction

~~SR status~~

- **SR** (status register) / flag register / PSW
  - Program Status Word
- ESI (extended source index)
  - in transferring data [used by memory transfer instructions]

Bit map
- 1 bit per pixel
- mapping from some domain → bits
- when domain = rectangle → bit map stores → binary image [each pixel is either black/white (any two colors)]

Pixmap
- multiple bits per pixel.

⇒ Addressing mode (AM)
- → where my operands are kept

(1) ADD R1, #5
- R1: Register
- #5: available in the instruction itself

AM → Register
→ Immediate

(2) ADD R1, R3

AM → Register
→ Register

(3) LOAD R1, 4000

AM → Register
→ Direct

loading the instruction at 4000 memory address to the register R1.

(4) MOV R2, 4000

AM → Register
→ Immediate

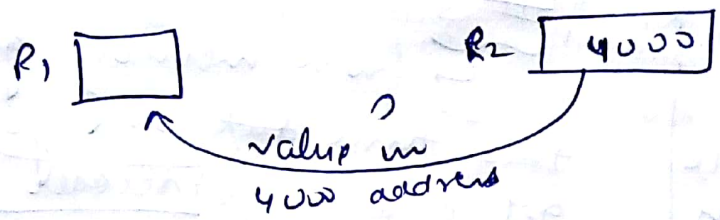
storing 4000 value to R2

(5) LOAD R_1, @R_2

AM → Register
→ Indirect

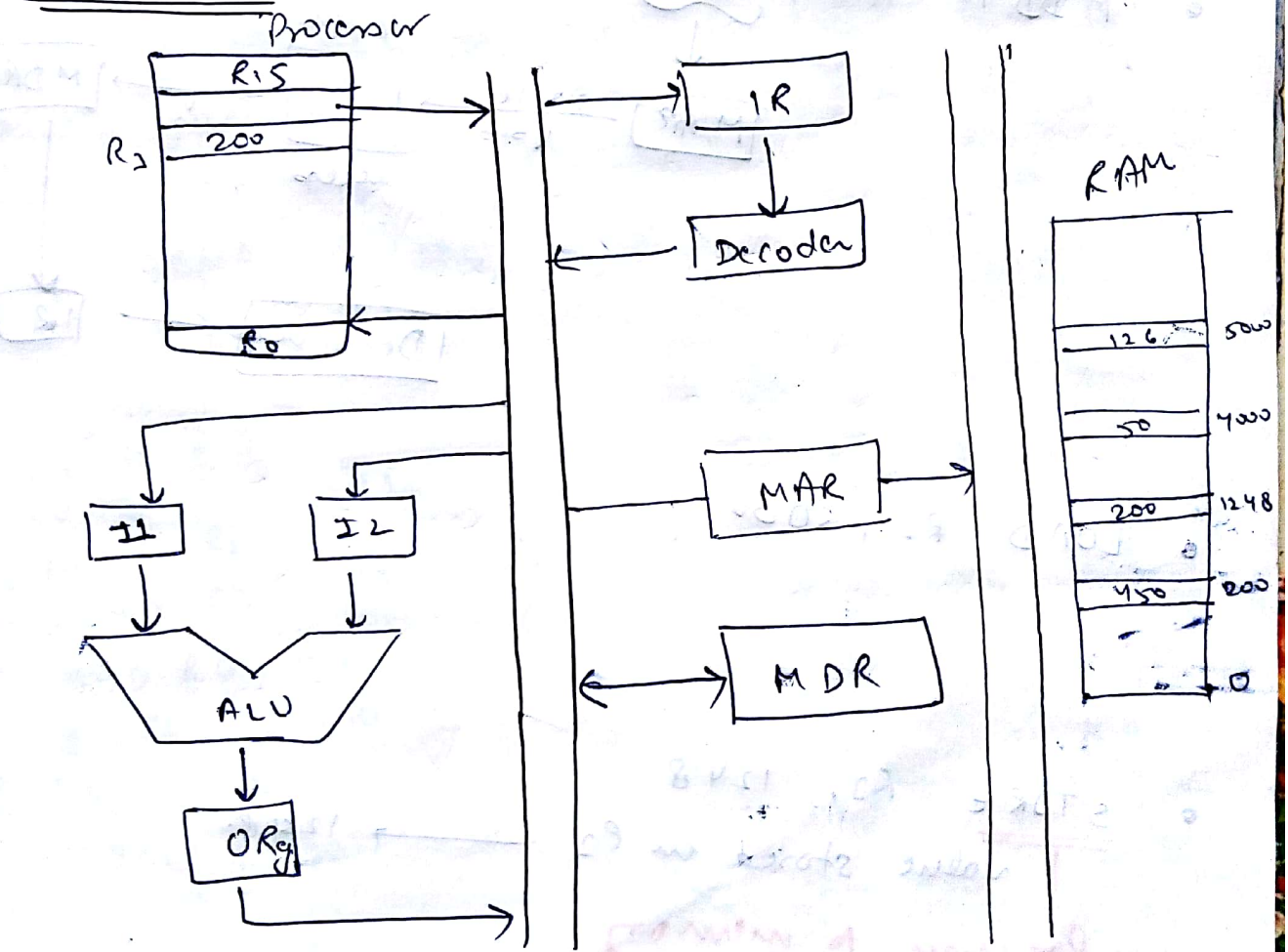
@R_2 → access the instruction stored in the value present in R_2

load that value to R_1



- REGISTER ADDRESSING MODE :-
Operand is kept in the register.

Data Path



- ADD R_1, R_2

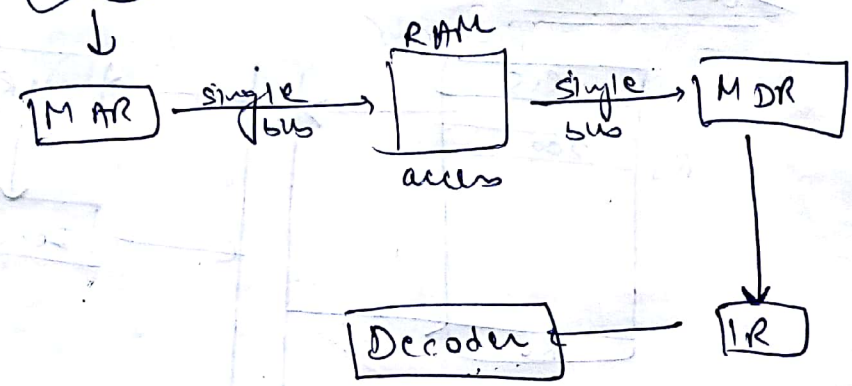
  from data bus will go to the I, R 12
  then ALU and get added → O Reg.
  then data bus get stored in any
  R_1 or R_2.

- ADD R_1, #5

  Single bus → Input register ALU

  from memory go to MDR
  then from ~~data~~ bus go to IR then
  get decoded in decoder
  ↓ single bus
  Input register ALU

- ADD M R_1, 4000



- LOAD R_2, 5000

- STORE R_1, 1248

  Value stored in R_1 → 1248.

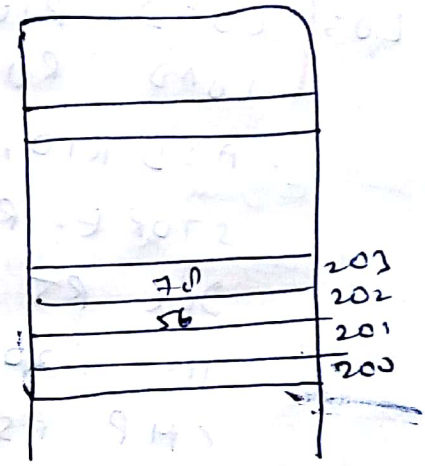
  Process to memory (MDR)

  Content of R_1 → MDR → IR → Decoder

- LOAD R_2, @R3

~~#~~

```
    MOV  R5, #4
    MOV  R0, 200
LO: LOAD  R10, @R0
    ADD   R10, #1
    STORE R10, @R0
    DEC   R5
    INC   R0
    CMP   R5, #0
    JNE   LO
```

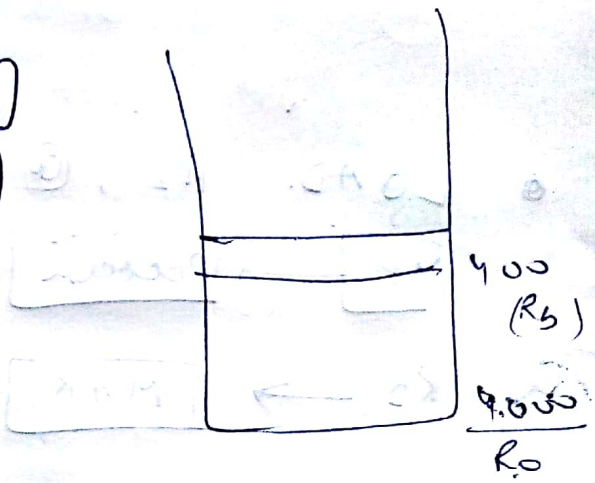



```
LOAD R0, 200
LOAD R1, 201
LOAD R2, 202
LOAD R3, 203
ADD  R0, #1
STORE R0, 200
ADD  R1, #1
STORE R1, 201
```

Indexed addressing

A = [_ _ _ _]

B = [_ _ _ _]




```
      MOV   R5 , #4
      MOV   R1 , 300
      MOV   R0 , 200
      MOV   R20 , #0
LO:   LOAD  R10 , R0(R20)
      LOAD  R11 , R1(R20)
      ADD R10, #1
      STORE R10, R0(R20)
      DEC R5
      INC R20
      CMP R5, #0
      JNE LO
```

if base addrs is added to both then (R20 + R5)

↓

Base-Indexed Addressing

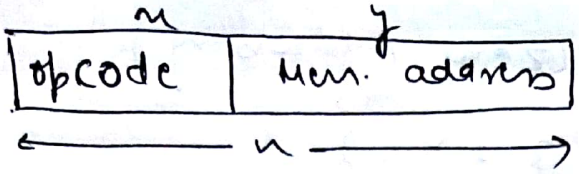
Stack Addressing (zero Addressing)

$(4 * 7 + 6) / (3 - 2 \times 5)$

Z = 4 7 * 6 + 3 2 5 * - /

```
PUSH  4
PUSH  4,7
MUL   28
PUSH  28,6
ADD   34
PUSH  34,3
PUSH  34,3,2
PUSH  34,3,2,5
MUL   34,3,10
SUB   34,-7
DIV   -4
POP   Z      → to store in Z.
```

#



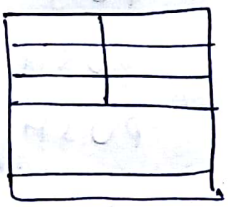
no. of opcodes $= 2^x$

Mem. address $= 2^y$

$x = 2$ → no of opcodes = 4

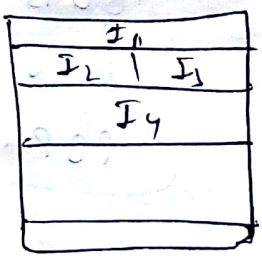
not in english

ADD	00
ADDI	01
LOAD	10
STORE	11



Equal length instruc^n

- instruc^{n}s can be easily decoded



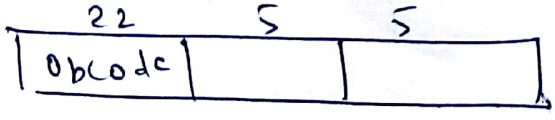
Unequal length instruc^n

- May have diff. format (flexibility ↑s)
- Decoding complexity ↑s.

Different types of instruc^{n}s

1. Control transfer
2. I/O
3. Arithmetic
4. Logical
5. Data transfer
   - ① MOV Rg Rg

I have 32 registers in my machine. How many bits I need for registers $(2^5) = 32$



MOV R1 R5

000 _ _ _ 0 0001100101

22 bits

② MVI Reg, #value

Depends on the range of values I want to have

17	5	10
Opcode	Reg.	Immedi

Eg: if I want to store till 1000 as value then max bits = 10 $2^{10} = 1024$

0010 _ _ 0 00101 0000001010

MVI Reg, #10

If address bits in our PC = 2^{30}

Ie: 1×2^{30}

↳ 1 GB RAM

If our PC is having 500 GB RAM then you can extend it to 1 GB but not above that.

DYADIC

having 2 operands

Masking → Monadic

Eg:

10000001	10001111	00000000	00100000
00 _ _ _ 0	1111111	00 _ _ _ _ _	_ 0 MASK

Rotate | 1011 | Shift

Left 1011

0111

Right 1101

Left 0110

Right 1101

due to sign

eg. 15 3 - add instructions (in remembrance)

14 2 add instructions

31 1 - add instructions

16 0 - add instructions

If shorter instruction then I can bring more instructions in 1 cycle

No of bits transferred per second - **Bandwidth**

i.e. ↑ing the bandwidth,

↑ing the efficiency

reduce the space occupied by instruction.

Assuming that instruction length = 16 bits

" " having 16 Registers. → 4 bits of address

opcode	Reg	Reg	Reg
4	4	4	4

16 = 4 bits.

ADD R1, R2, R3

0000 0001 0010 0011

15 3-add instrns

0000

Add 0000 xxxx yyyy zzzz

sub 0001 " " " "

1110

14 2-add instrns

Add 00000000 xxxx yyyy

escape code

1111 0000 xxxx yyyy

1111 0001 " "

1111 1101 " "

get confused with

ADD R0 R1 R2

1111

31 1-add instrns

1111 1110

1111 1111

these two of available now

16 { 1111 1110 0000 xxxx
1111 1110 0001 "
1111 1110 1111
1111 1111 0000 }

15 { 1111 1111 1110 }

if 1111 then no 0-add instr

16 0-add instrns

1111 1111 1111 0000

1111 1111 1111 1111

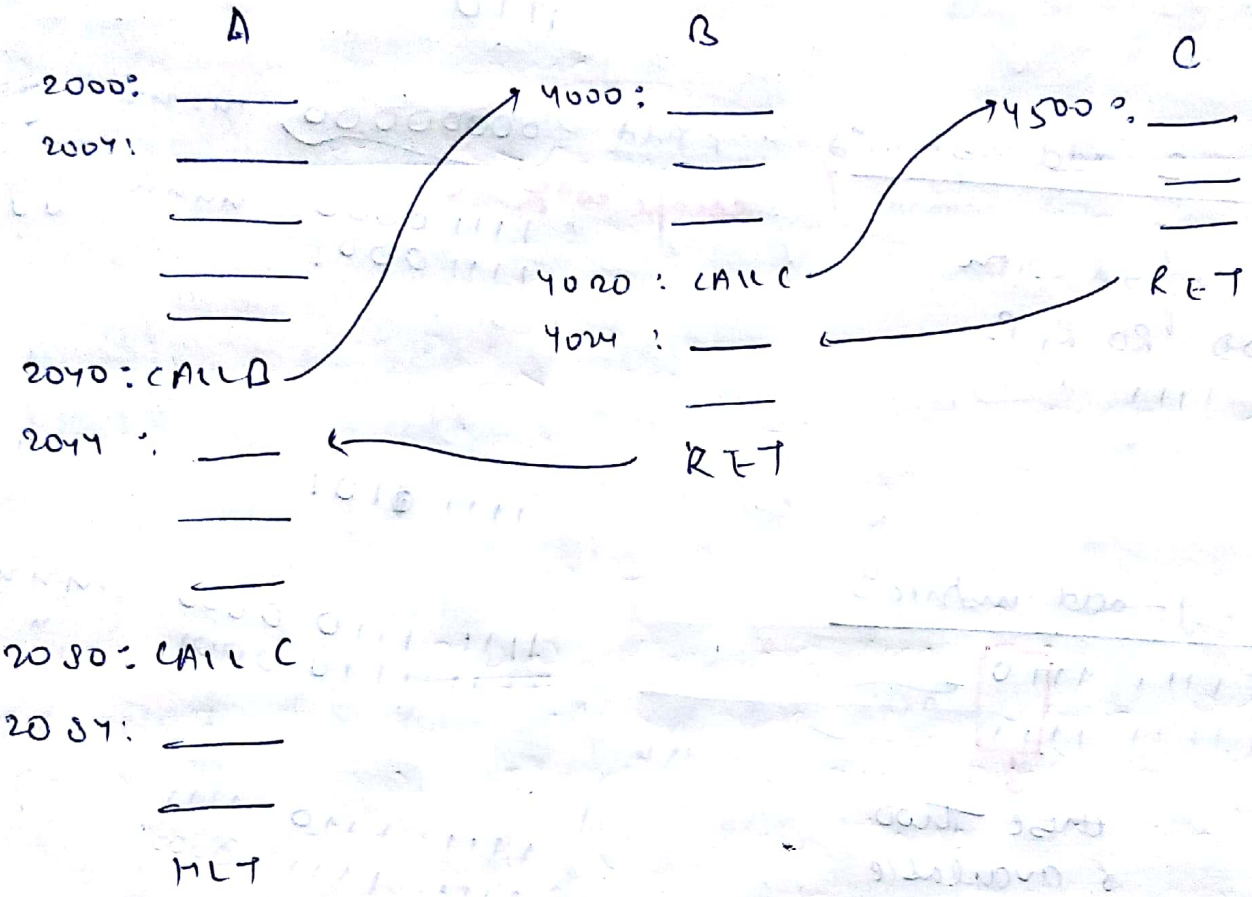
0-addressing Machine

[Push, POP, NULL,

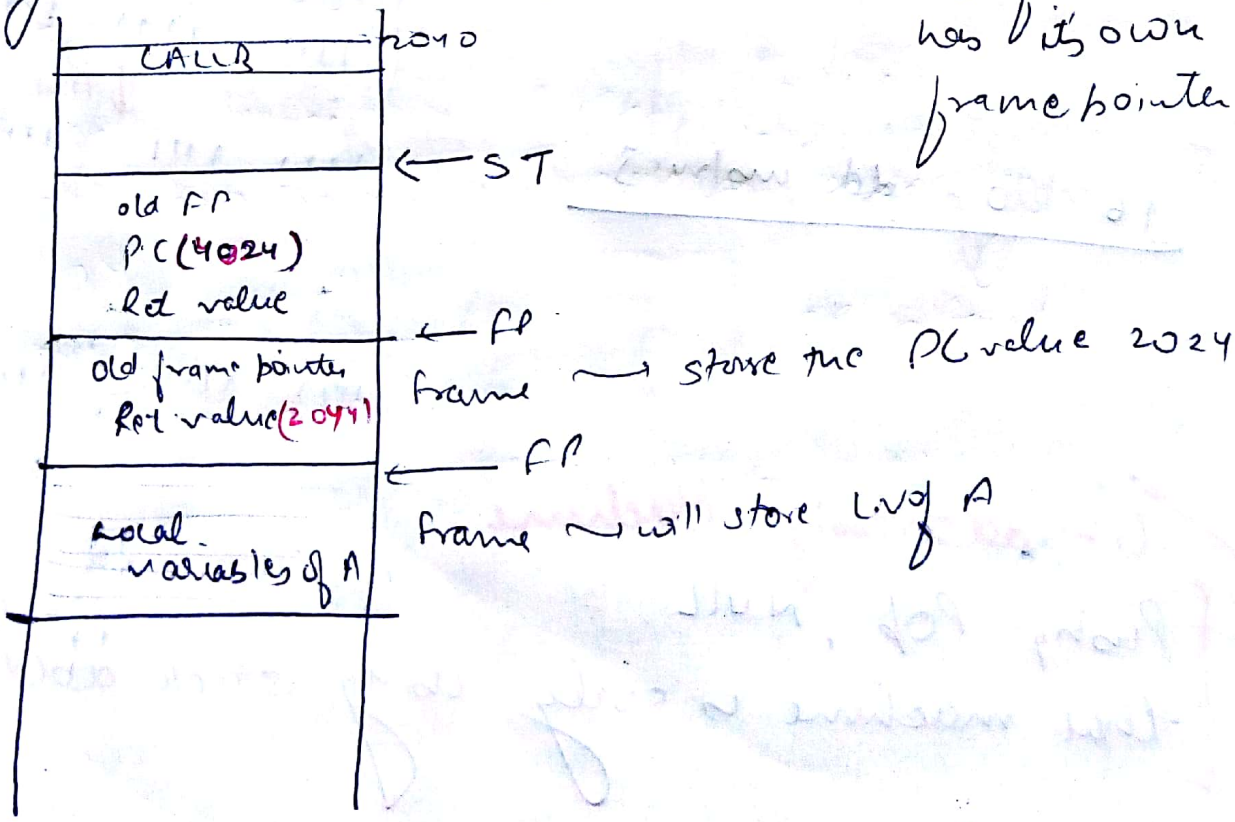
that machine is only using stack addressing

- Conditional + Branch

like JLT, JGT - through the flag register

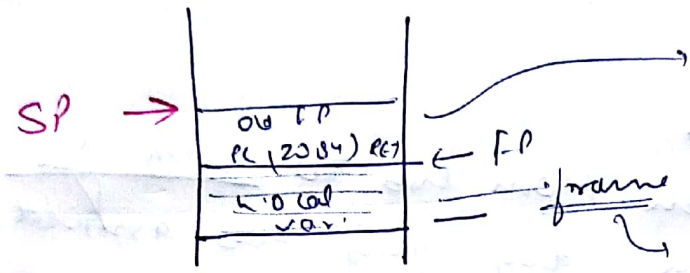


During function call, frame will be created



As soon as function call finishes the frame of that function get destroyed / removed

As call to C is done & RET frame of C is removed



after call to C is done in A it gets removed

after A is completed it also gets removed.

- Peripheral devices

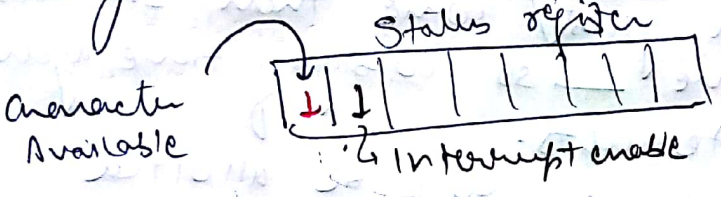
Those devices which connected to the processor

eg. I/O devices

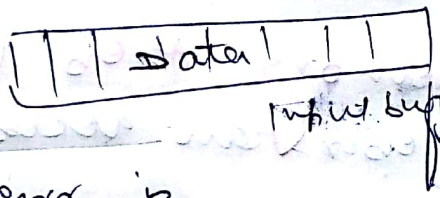
All I/O devices have a controller

like Hardisk have controller

Memory also has a controller



as soon as we input data, becomes 1.



now Microprocessor is going to check SR whether I is there or not, if yes then it will transfer the data into the new processor.

① processor checks the device whether the data is available / not. So every minute it will go to the SR to check.

(1) Programmed I/O with busy waiting

↓ so to stop this

(2) Interrupt driven
Here SR is going to tell to the processor that he has come.

eg Marriage at home,
if no doorbell → then Mom has to check again & again if any guest has arrived/not. Programmed I/O with busy waiting

if doorbell ring → then she need not to check again & again. Guest themselves are going to inform her that we have arrived.

Now the problem is, if 100 no. of guest arrive one by one like 1 guest come, Mom go welcome him and come back, then other guest come then she has to again come → so wastage of time again and lot of interrupts.
So one person will be allotted

(3) Direct Memory Access (DMA)
only processor can access memory
So
DMA
↳ a controller
↳ will get access to memory
↑
SR is Ring