BOOLEAN ALGEBRA

AND	•	$(x.y)$	$\wedge$ $(x \wedge y)$
OR	+	$(x+y)$	$\vee$ $(x \vee y)$
NOT	'	(x')	$\sim$, $-$ $(\bar{x})$
NAND			
NOR			
XOR	$\oplus$	$(x \oplus y)$	

$\Rightarrow$

x	y	$x.y$	$x+y$	x'	y'	$x \oplus y$	$(x.y)'$	$(x+y)'$	$x'.y'$
0	0	0	0	1	1	0	1	1	1
0	1	0	1	1	0	1	1	0	0
1	0	0	1	0	1	1	1	0	0
1	1	1	1	0	0	0	0	0	0

$\equiv$

$\equiv$

$(x'+y')$
1
1
1
0

LAWS

- $x \cdot 0 = 0$; $x + 0 = x$
- $x \cdot 1 = x$; $x + 1 = 1$

COMMUTATIVE LAW
- $x + y = y + x$; $x \cdot y = y \cdot x$

ASSOCIATIVE LAW
- $(x+y)+z = x+(y+z)$; $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

D'MORGAN'S LAW
- $(x \cdot y)' = x' + y'$; $(x+y)' = x' \cdot y'$
- $x + yz = (x+y)(x+z)$

$R.H.S = x \cdot x + x \cdot z + y \cdot x + y \cdot z$

$= x(1 + y + z) + yz$

$= x \cdot 1 + yz$

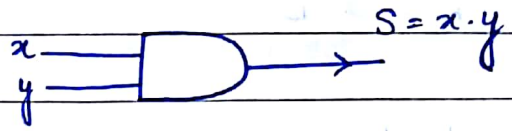
$= x + yz$

- $x(y+z) = x \cdot y + x \cdot z$
- $(x')' = x$
- $x + x' = 1$
- $x \cdot x' = 0$.
- $x \cdot x = x$

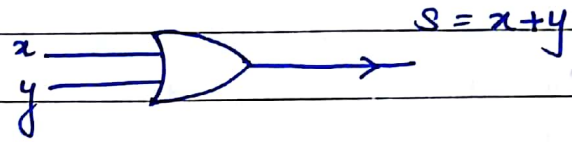
$x + x = x$

⊙ Logic Gates.

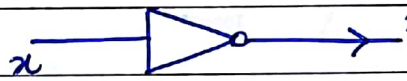
1. AND



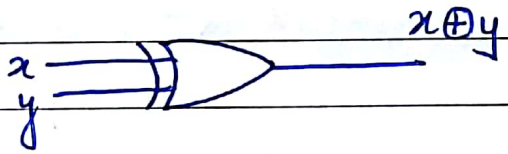
2. OR



3. NOT



4. XOR

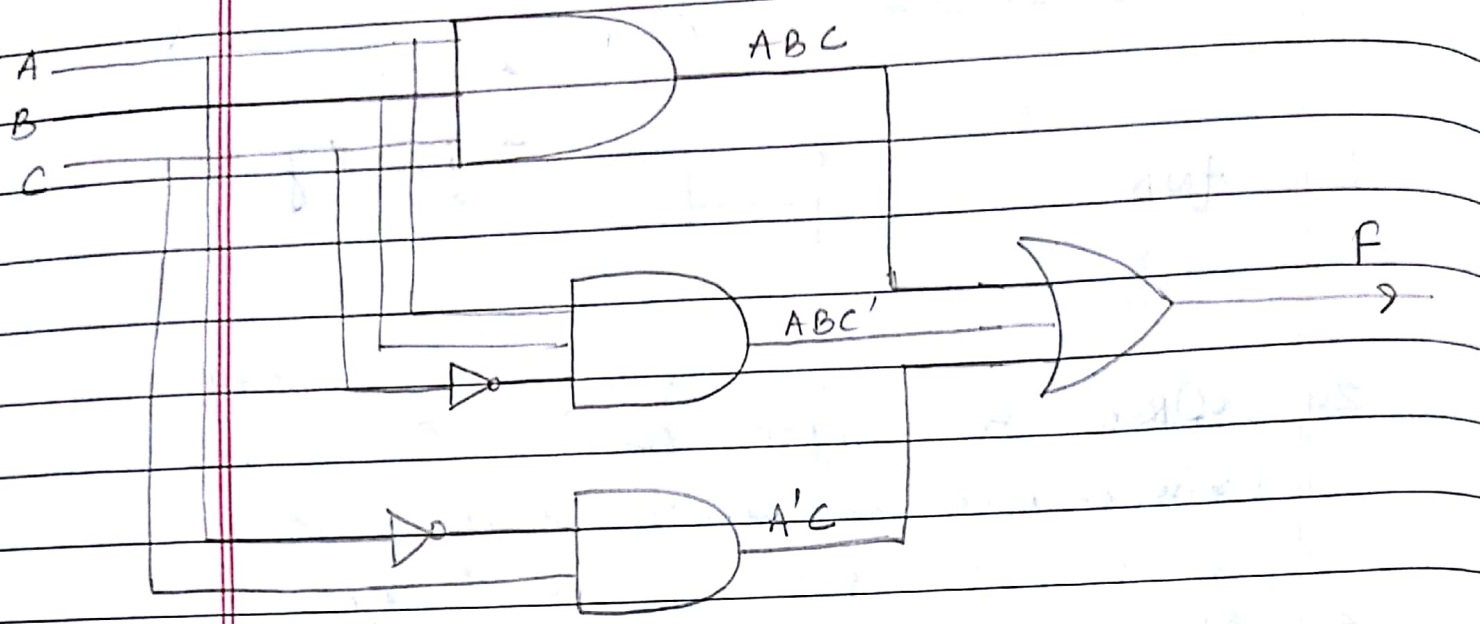


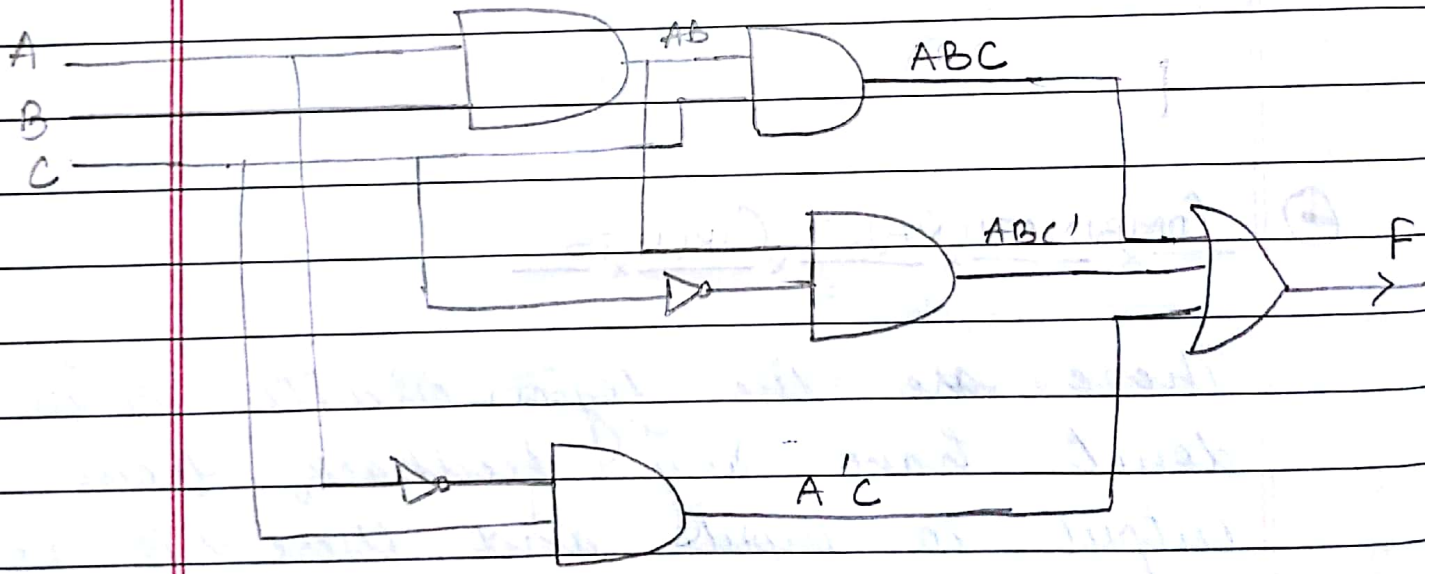
⊙ Combinational Circuits

These are the logic circuits which donot have any feedback from output to input and there is no memory.

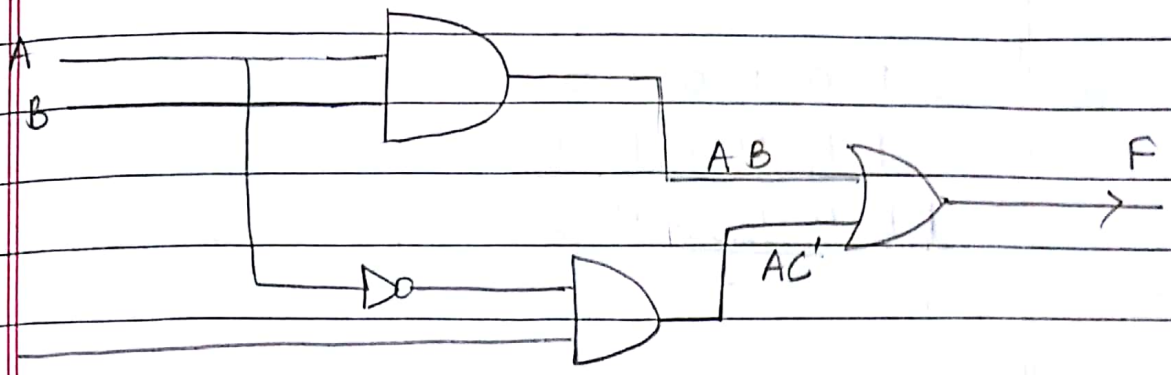
→ $ABC + ABC' + A'C = F$

P.T.O





$$F = ABC + ABC' + A'C$$
$$= AB(C+C') + A'C$$
$$= AB.1 + A'C$$
$$= AB + A'C$$



Before we represent the Boolean Expressions through logic gates, the expressions are simplified by using the basic identities so that the expressions could be achieved by using lesser hardwares.

⊙ HALF ADDER

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 10$ ← Sum

↑ carry

INPUT		OUTPUT	
x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

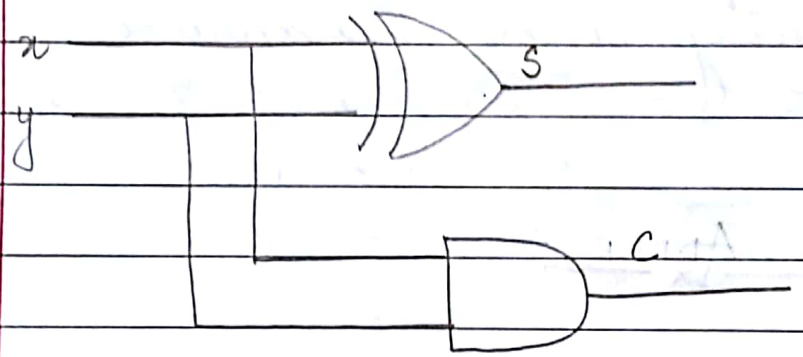
$S = x'y + xy'$

$= x \oplus y$

$C = xy.$

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

It is a combinational circuit which adds 2 bits of data and produces the sum and the carry from the sum.



x	Half Adder Block Diagram.	S
		C
y		

Each row of the truth table forms a minterm. The component of the input variable is taken if the value is zero and the variable is taken as such if the value is 1,

the product of these variables or their components is taken which forms the minterm. The sum of these minterms where the output is 1, forms the expressions.

⊙ Full Adder

It is also a combinational circuit which adds 3-bits of data and produces two outputs, the one is the sum of these bits while the second is the carry from the sum of these bits.

TRUTH TABLE.

I/P			O/P	
x	y	z	S	C
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

Out of three bits, 2 bits represent the significant to be added and the 3rd bit is the carry from the addition of lower significant bits.

$$C = x'yz + xy'z + xyz' + xyz$$

$$= (x'y + xy')z + xy(z + z')$$ (z + z' = 1)

$$= (x \oplus y)z + xy$$

$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$= (x'y' + xy)z + (x'y + xy')z'$$

Now,

$$(x'y' + xy)' = (x'y')' \cdot (xy)'$$

$$= (x+y)(x'+y')$$

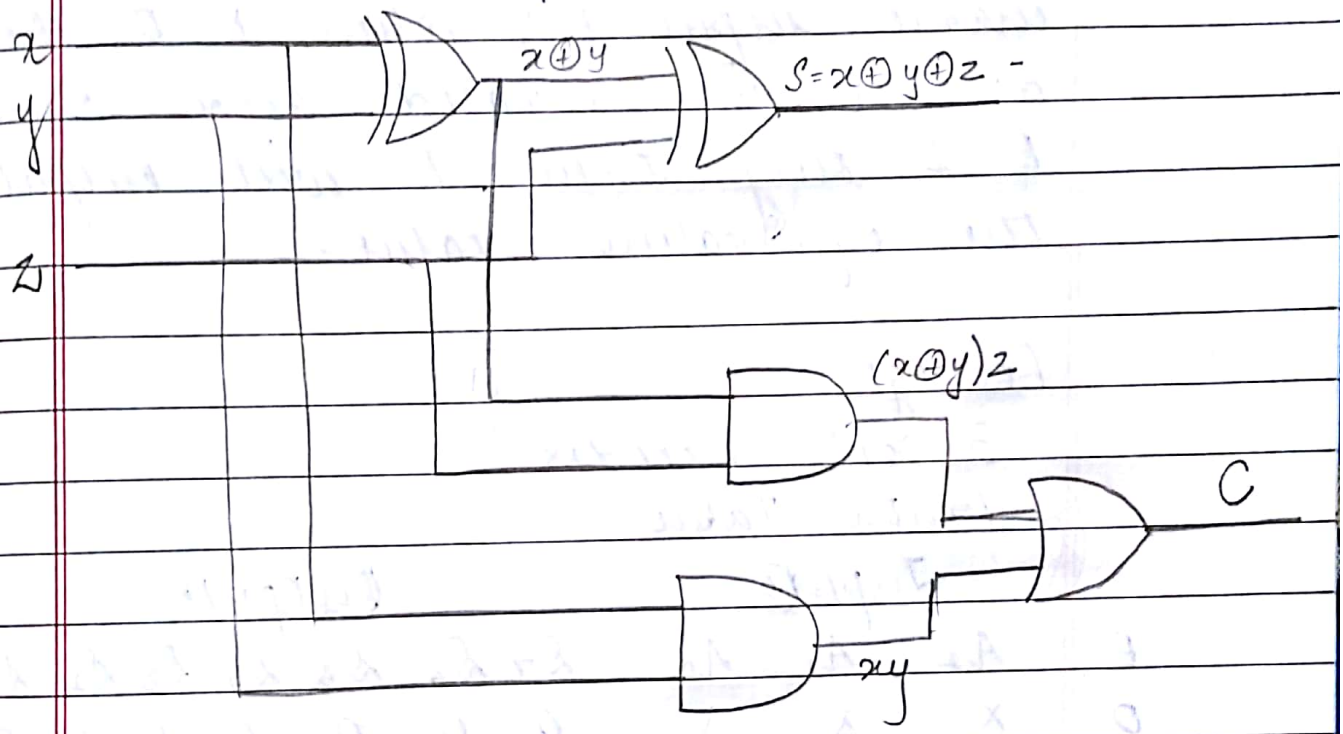
$$= xx' + xy' + x'y + yy'$$ (xx' = 0, yy' = 0)

$$= xy' + x'y$$

$$= x \oplus y$$

$$S = (x \oplus y)'z + (x \oplus y)z'$$

$$= ((x \oplus y) \oplus z)$$



⊙ Decoder

We may represent the information in the coded form. If we have a code of n bits, then it can represent 2^n distinct information (values).

A decoder is a combinational circuit which decodes the input code of n bits and outputs

its equivalent value, which is one of the 2^n distinct values. The decoders may have are termed as n - form decoders where $m \leq 2^n$.

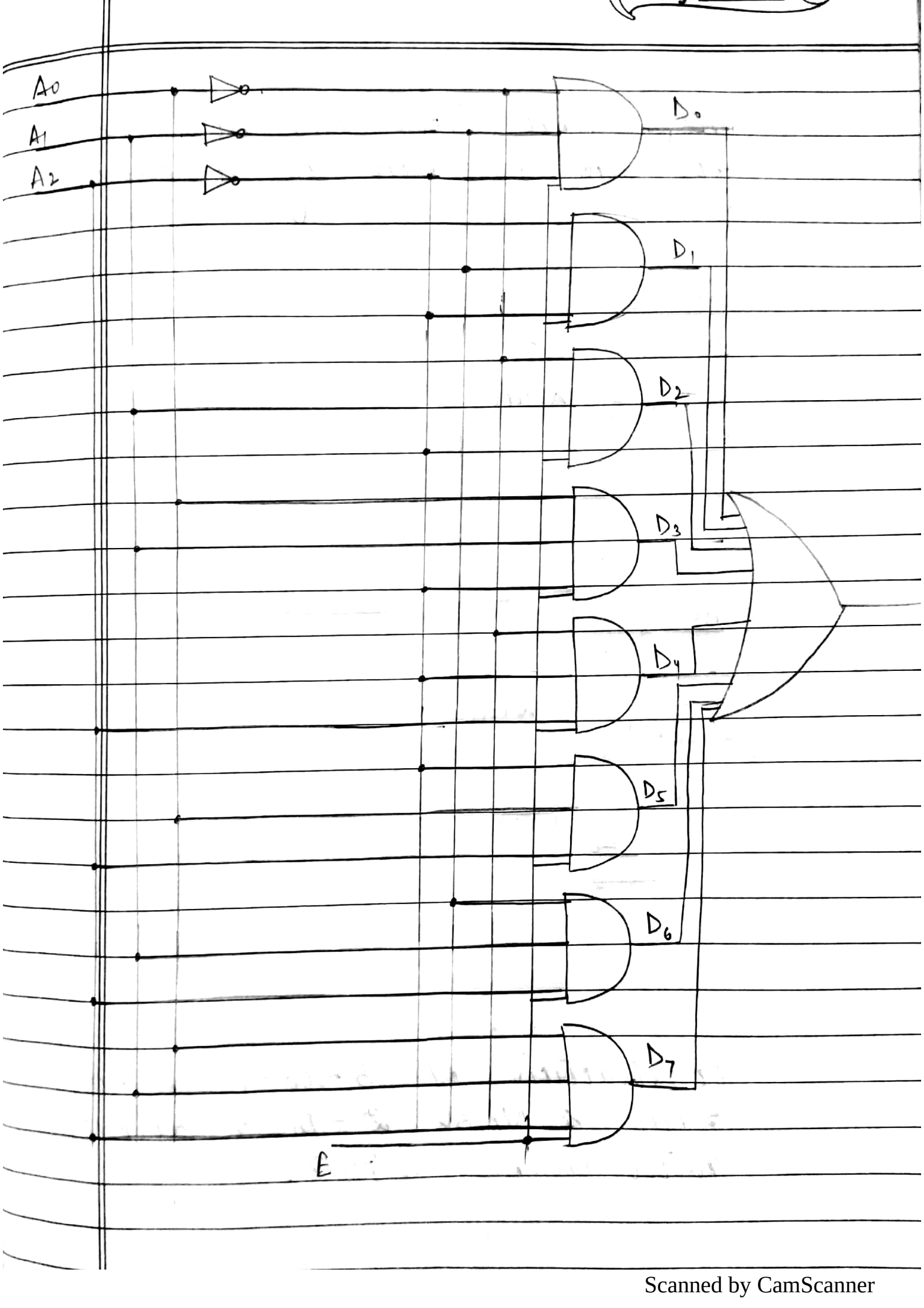
A decoder may have an enable input E, when E = 0, the decoder is disabled and when E = 1 only then it will output the equivalent value.

For eg:

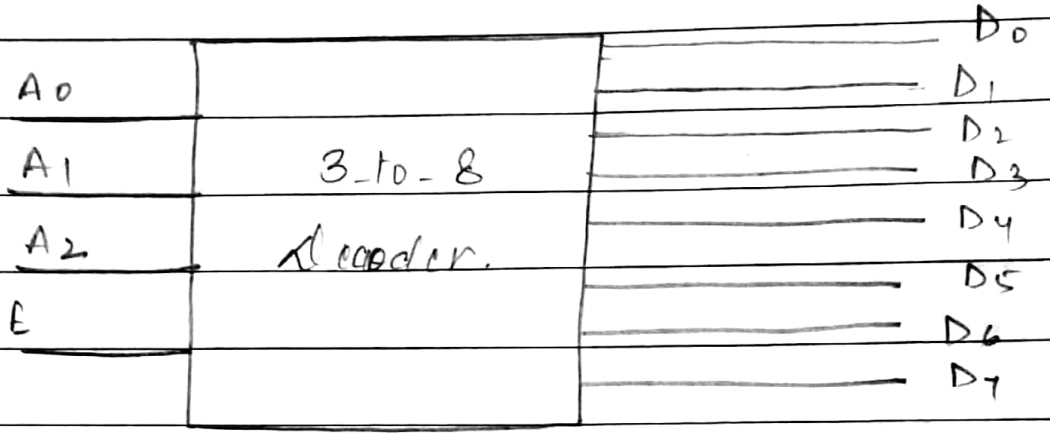
3 bits decoder

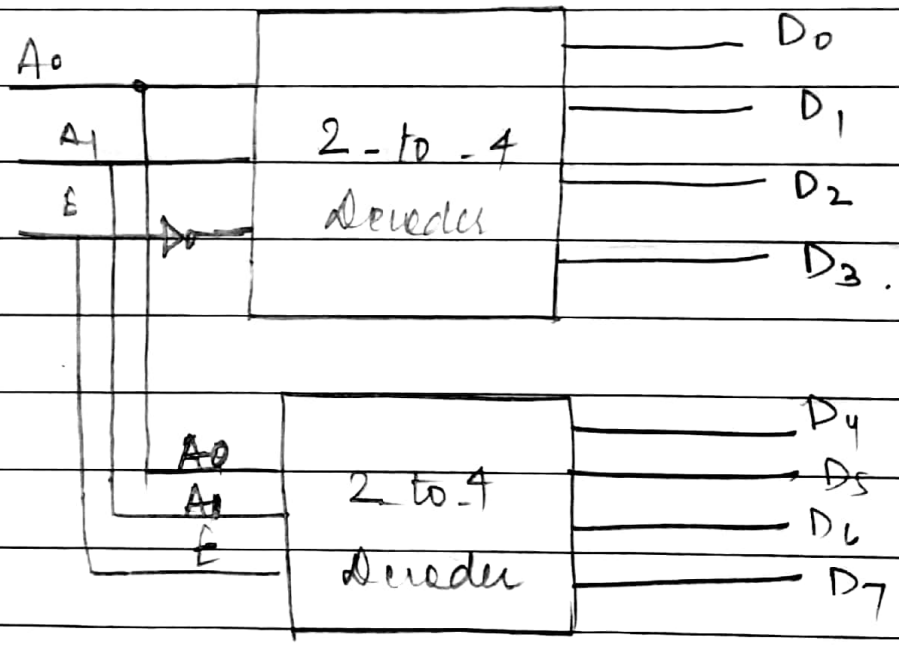
Truth Table

Inputs				Outputs							
E	A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0



The enable input can be used for the extension of decoders.





Construction of 3-to-8 decoder from 2 No's 2-to-4 decoders with enable inputs.

⊙ Multiplexers

These are also combinational circuits. It has 2^n input lines, n selection inputs and 1 output line.

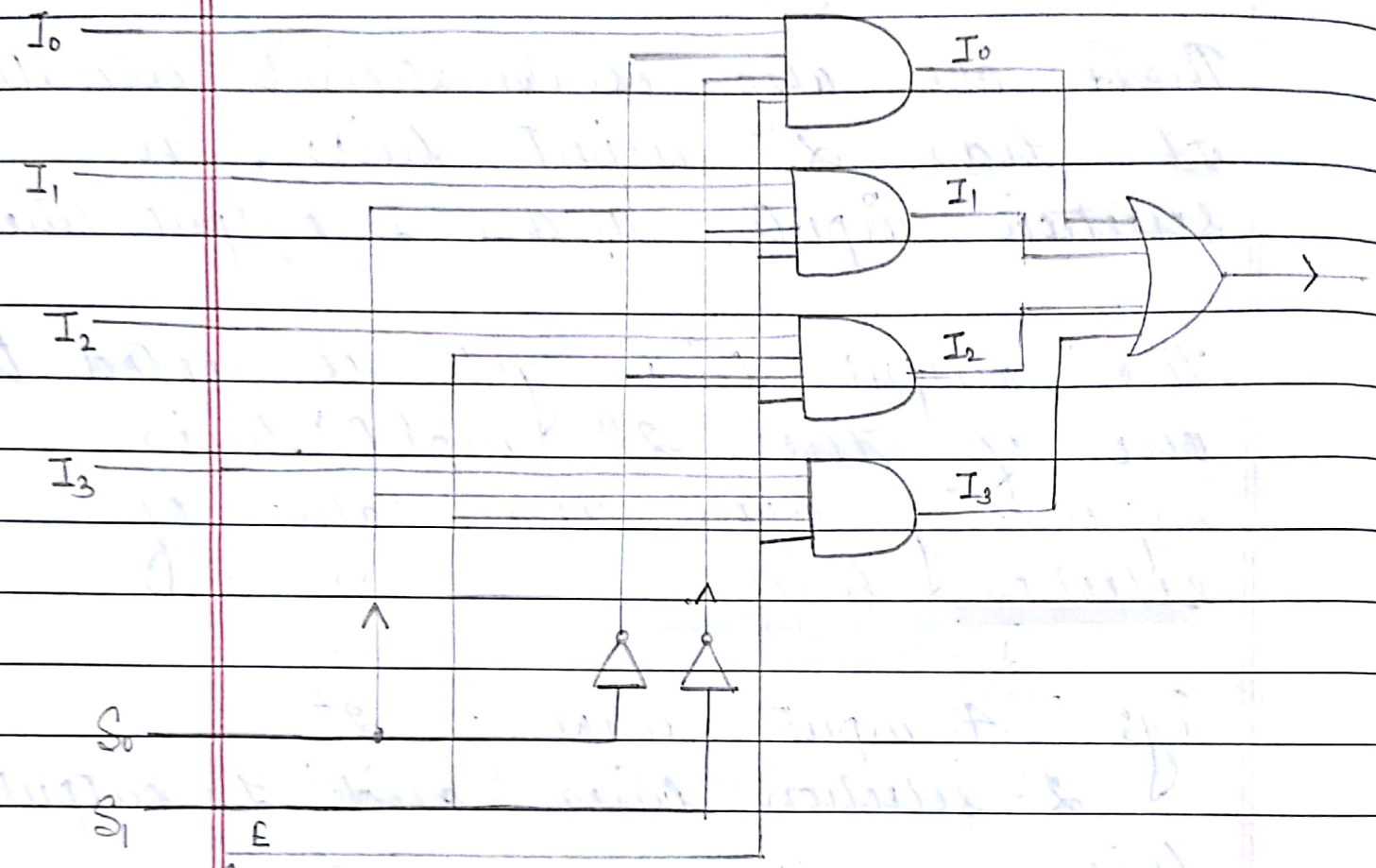
The output line gets connected to one of the 2^n input lines depending upon the value of selection lines.

Eg: 4 input lines = 2^2
2 selection lines and 1 output line.
Let the selection lines are S_1, S_0.

function table

Selection lines		Output
S_1	S_0	S
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

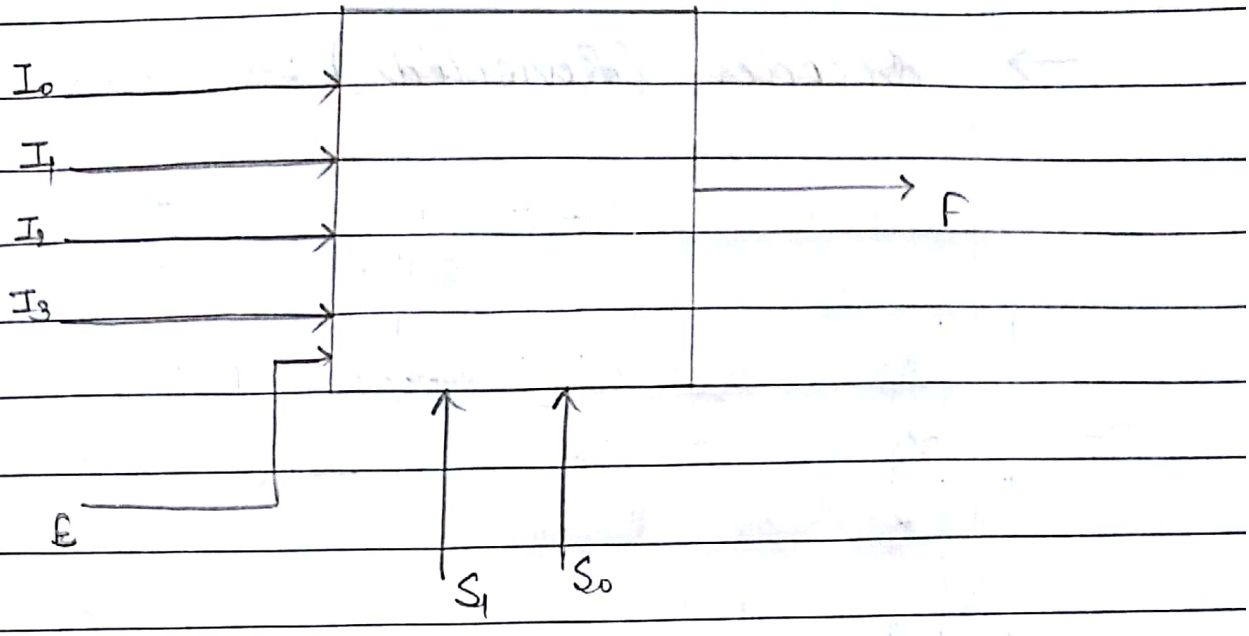
→ 4_to_1 Multiplexer :-

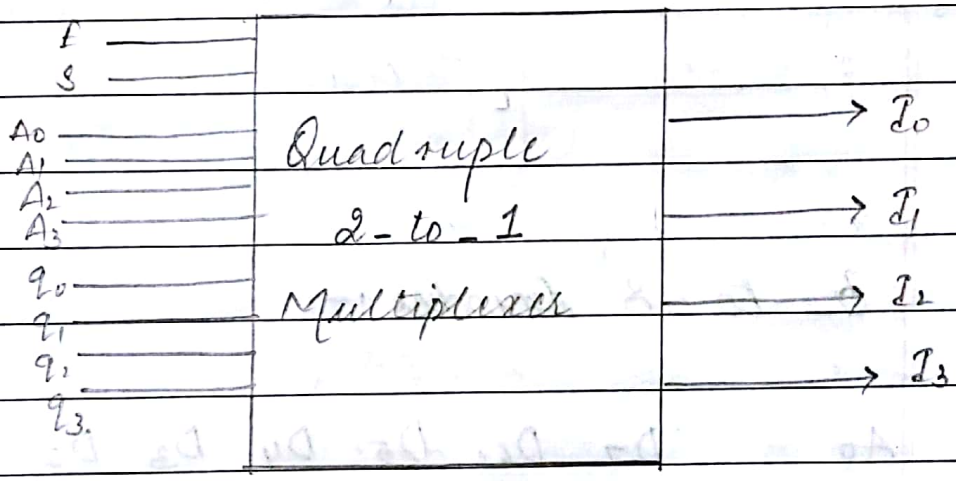


Function Table :-

S_1	S_0	Output
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

$$\begin{Bmatrix} S_0 = 1 \\ S_1 = 0 \end{Bmatrix}$$

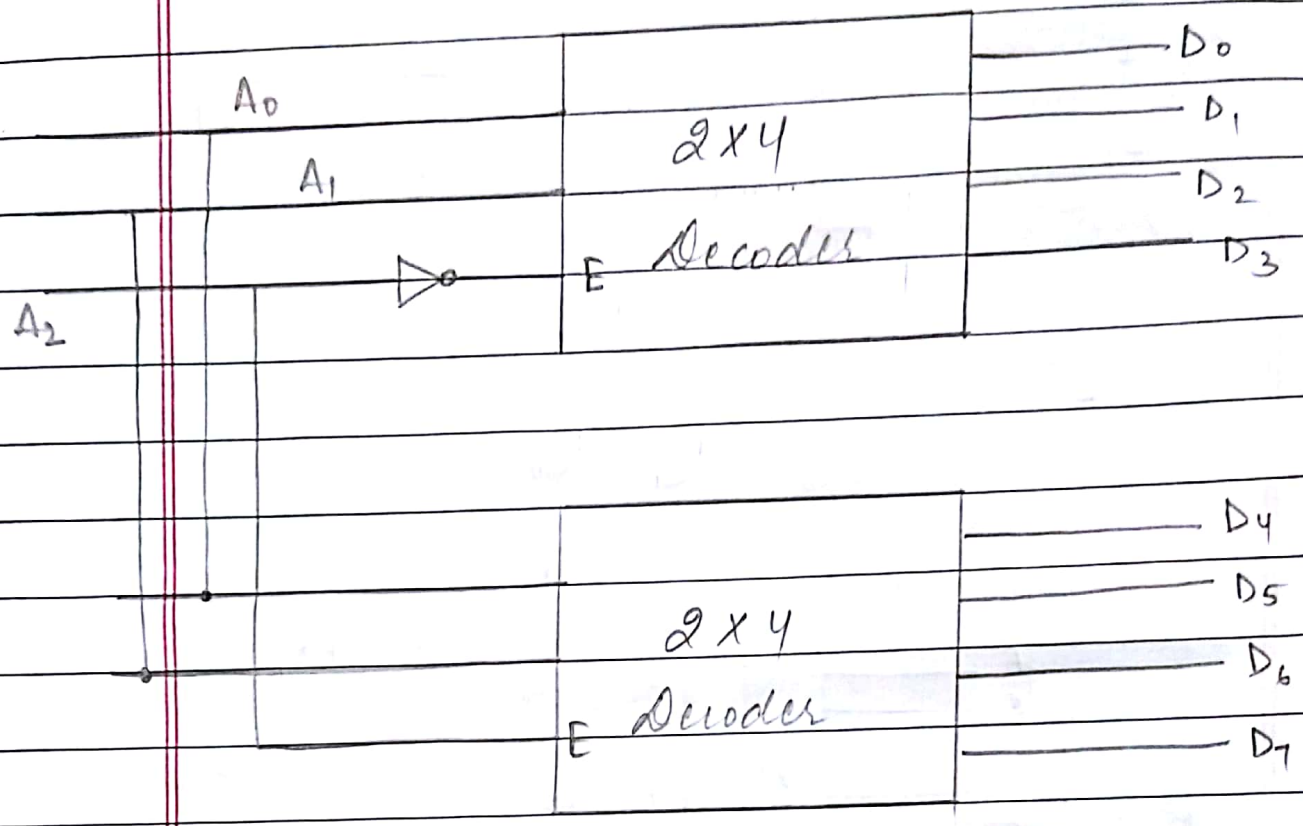




Function Table:-

Inputs		Outputs
E	S	F
0	X	0
1	0	A
1	1	B.

→ Decoder (Revisited) :-



3_to_8 Decoder :-

A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

→ AND GATE:-

x	y	F = x.y
0	0	0
0	1	0
1	0	0
1	1	1

In terms of maxterms :-
(Product of sums)

$F = (x+y)(x+y')(x'+y)$

$= (x+y)(xx' + xy + x'y' + yy')$

$= (x+y)(xy + x'y')$

$= xxy + xxy' + xyy + x'yy$

$= xy + xy$

$= xy$

In terms of minterms :-
(Sum of products)

⊙ Encoders:-

These work opposite to the decoders. The information required to be transmitted may be encoded and after it reaches the destination it is decoded. This way we can save the hardware.

FUNCTION TABLE:

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

$$A_0 = D_1 + D_4 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$

⊙ Register Transfer Language :-

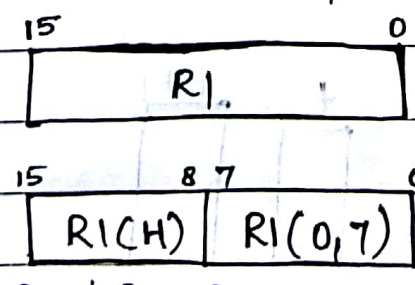
The register hardware of a computer is better understood by the following:

1. The set of registers and their func^{n}s.
2. The sequence of microoperations that can be performed on the registers.
3. The controls that initiates all these microoperations.

The symbolic notations used to describe the microoperation transfers amongst registers is called register transfer language.

⊙ Register transfer

The registers are denoted by capital letters followed by some numerals such as R1, R2, MAR, PC, IR, etc.
The registers are represented as:-



Register Representation

The transfer is represented as

$$R2 \leftarrow R1$$

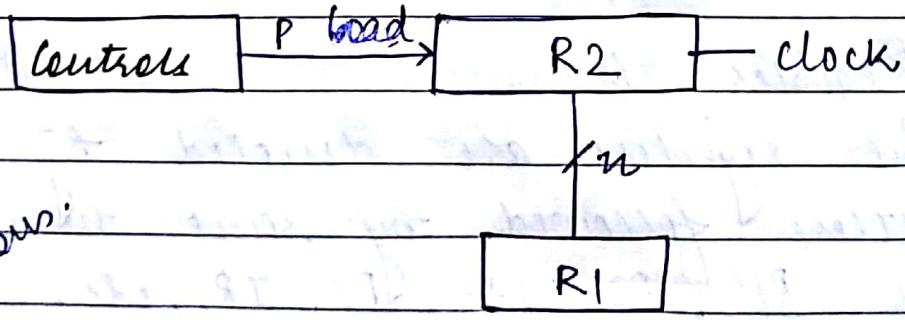
This operation transfers the contents of R1 to R2 but the contents of R1 does not change.

If we want to associate a control with this transfer then we may write,

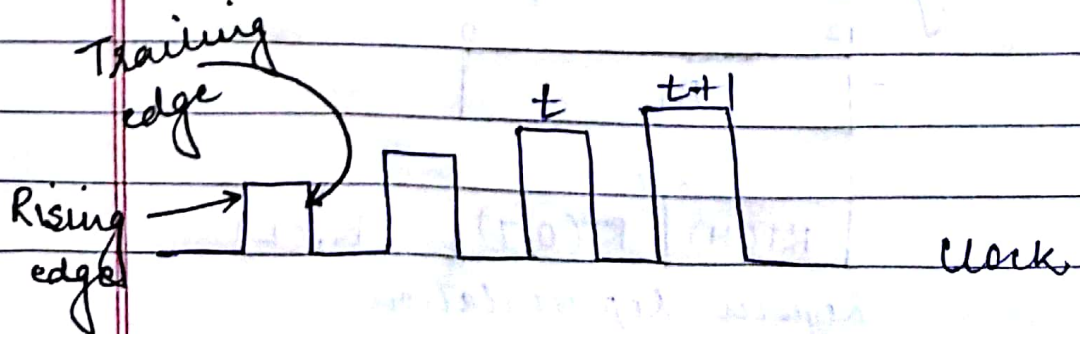
$$If\ (P=1)\ then\ (R2 \leftarrow R1)$$

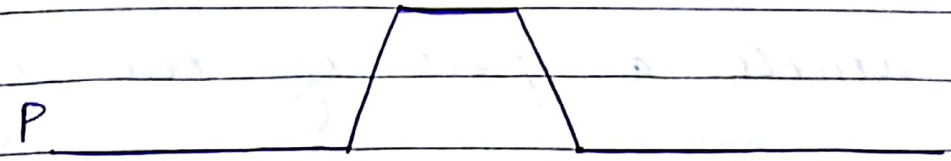
Further in order to dissociate the control signal from the micro-operation then we may write,

$$P:\ R2 \leftarrow R1$$



n denotes no. of lines in the bus.





The clock generates pulses at regular intervals of time. The control signal P becomes high (active) at time t but the transfer shall take place at the next rising edge of the clock. that is at t+1. After the transfer has taken place, the control signal is deactivated, i.e., path set to 0. If we do not do it then the transfer shall take place at every rising edge of the clock thereafter.

If we want to have more than one microoperation transfers at the same time, then we may write:

T: R2 ← R1, R4 ← R1

The comma (,) shall separate the two microoperations.

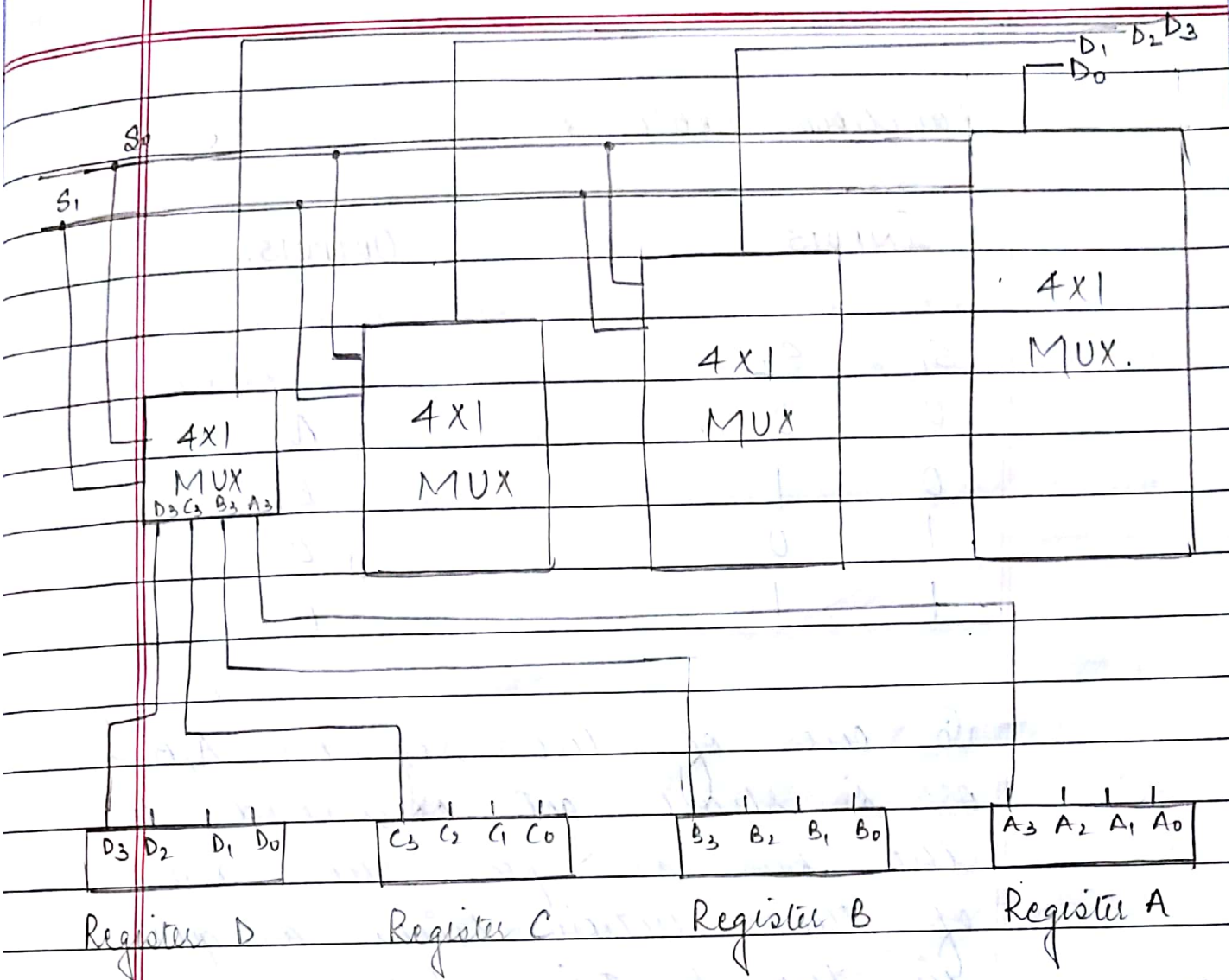
So we use the symbols as under:-
, for separating the two operations,
← denotes data transfer, () used to

denote a part of the register.

⊙ Bus of Memory Transfers.
There are many registers in the CPU and auxiliary locations in the RAM. If we connect all the pairs of registers and the registers and memory locations then the circuitary shall be very complex and cumbersome. In order to make it simple we use a common bus and all the registers and memory locations are connected to the bus. All data transfers shall take place by using the bus.

For example, 4 registers with 4 bits, each may be constructed as under:

→



Function Table:

Next Page →

Inputs		Output
S_1	S_0	
0	0	A
0	1	B
1	0	C
1	1	D

So one of the registers A, B, C or D shall get connected to the Bus, as per the

Function Table:

INPUTS		OUTPUTS.
S_1	S_2	
0	0	A
0	1	B
1	0	C
1	1	D

So one of the registers A, B, C or D shall get connected to the bus as per the values of the selection lines as given in the function table.

$K = x^P$

$\log_x K = P$

If there are K registers with n bits each, then we need to have n multiplexers of the size K to 1. The number of selection lines shall be $\log_2 K$.

Once the data is available on the bus from one of the registers selected by the selection lines, the data can be transferred from the bus to any one of the registers.

For example: Transferring the data from register C to register R by using the bus can be written as,

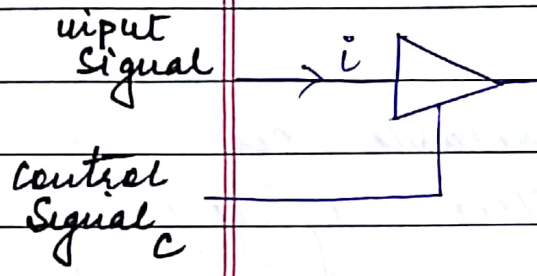
bus ← C, R ← bus.

If it is implied that we are accessing the common bus then we may write:

R ← C;

⊙ Tristate Bus Buffers

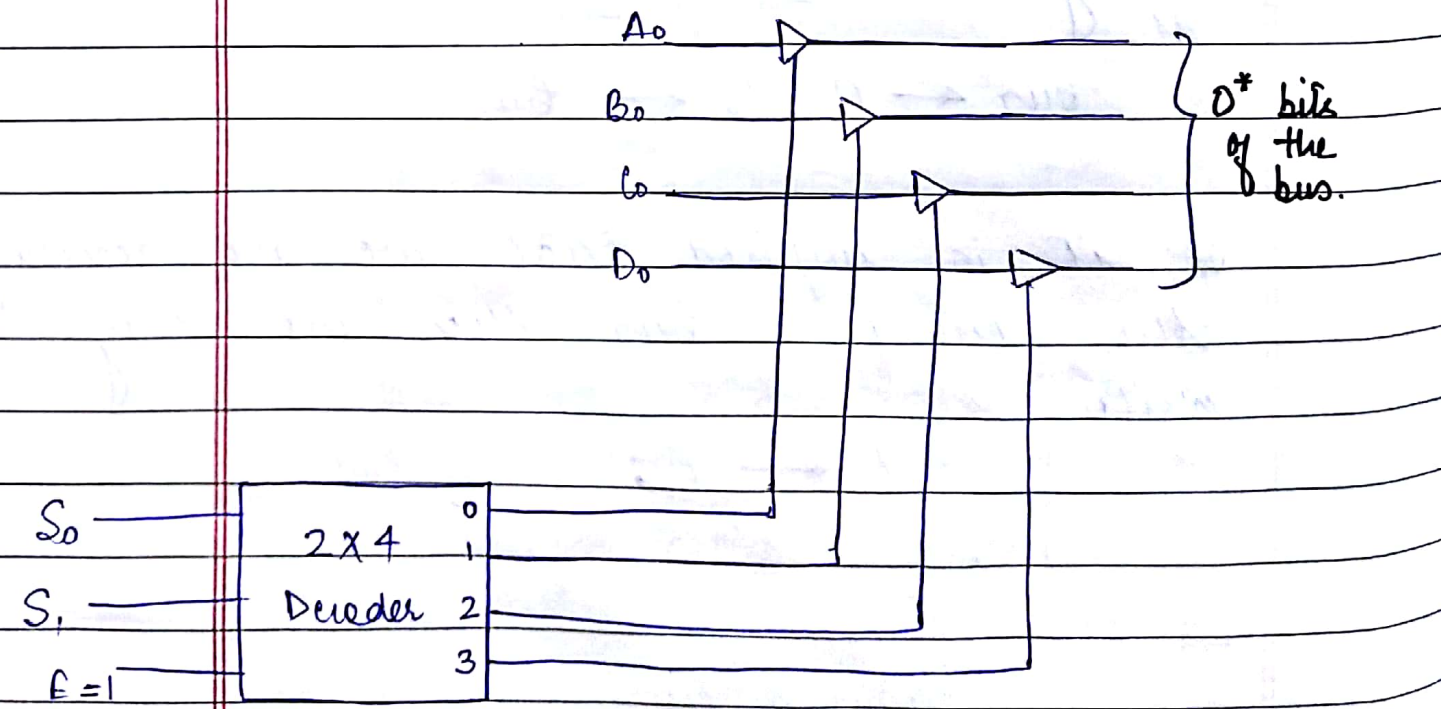
Instead of using the multiplexers we may use tristate buffers for using the common bus.



output = i if C=1

if C=0, no O/P
high impedance state

In the high impedance state when the control signal is 0, the tristate buffer behaves like an open circuit (i.e; high impedance).



Decoder function Table:

I/P			O/P
E	S_1	S_0	
0	x	x	x
1	0	0	0
1	0	1	1
1	1	0	2
1	1	1	3.

For example, when selection inputs S_0, S_1 are 0, 0 then as per the function table, the line 0 of the decoder shall be high and the tristate buffer connected to this ~~state~~ shall behave like an ordinary buffer and all other buffers having the control signal low shall be in the high impedance state. So A0 gets selected. Likewise if each register is having n bits then there shall be n such arrangements of tri-state buffer but the decoder shall only be one.

If there are 8 registers, then we need to have 3 to 8 decoders.

⊙ Memory Transfers

In this case, either the source or the destination is the memory, i.e; we transfer the data either from memory to a register, or from a register to a memory location. The memory is specified by the capital letter M. The address of the location is specified by the Address Register (AR). So M[AR] specifies the memory with address AR. Taking out the data from memory is called a READ operation and putting in the data to the memory is called WRITE operation.

READ: $R1 \leftarrow M[AR]$

is transferring the data from memory to R1.

and

WRITE: $M[AR] \leftarrow R1$

transfers the data from the register R1 to the memory specified by AR when WRITE control signal goes high.

① Arithmetic Microoperations

Incrementing, decrementing the contents of a register and addition and subtraction of the contents of two registers and taking the 1's and 2's complements fall under this category.

Incrementing :-

$$R1 \leftarrow R1 + 1$$

Decrementing :-

$$R1 \leftarrow R1 - 1$$

Addition :-

$$R1 \leftarrow R1 + R2$$

Subtraction :-

$$R1 \leftarrow R1 + \overline{R2} + 1$$

1's complement :-

$$R1 \leftarrow \overline{R1}$$

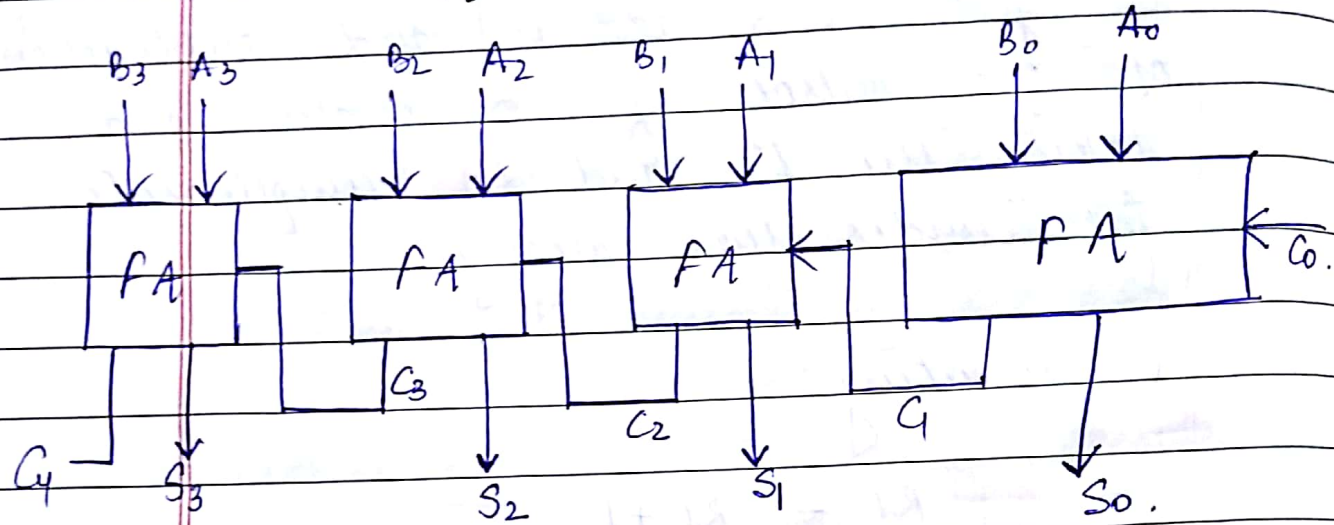
2's complement :-

$$R1 \leftarrow \overline{R1} + 1$$

8.

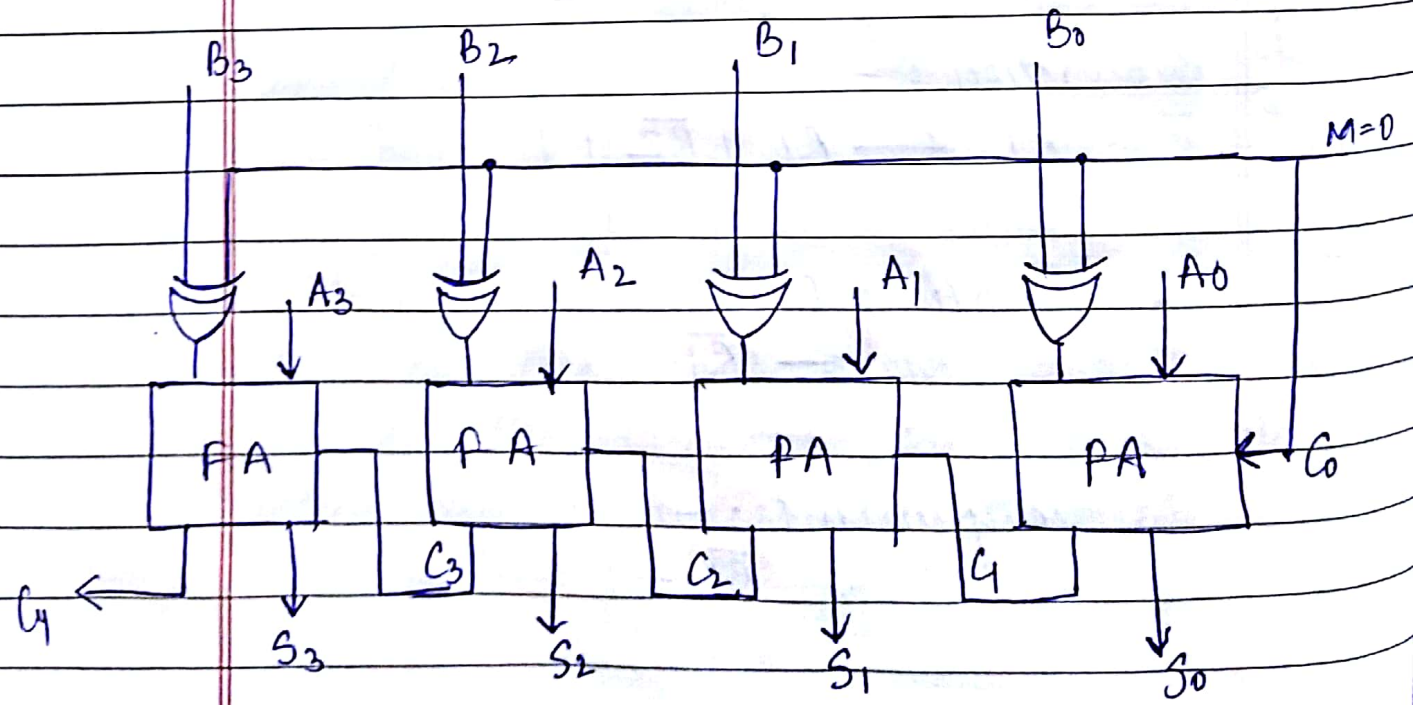
⊙ 4-bit Adder

BLOCK DIAGRAM:



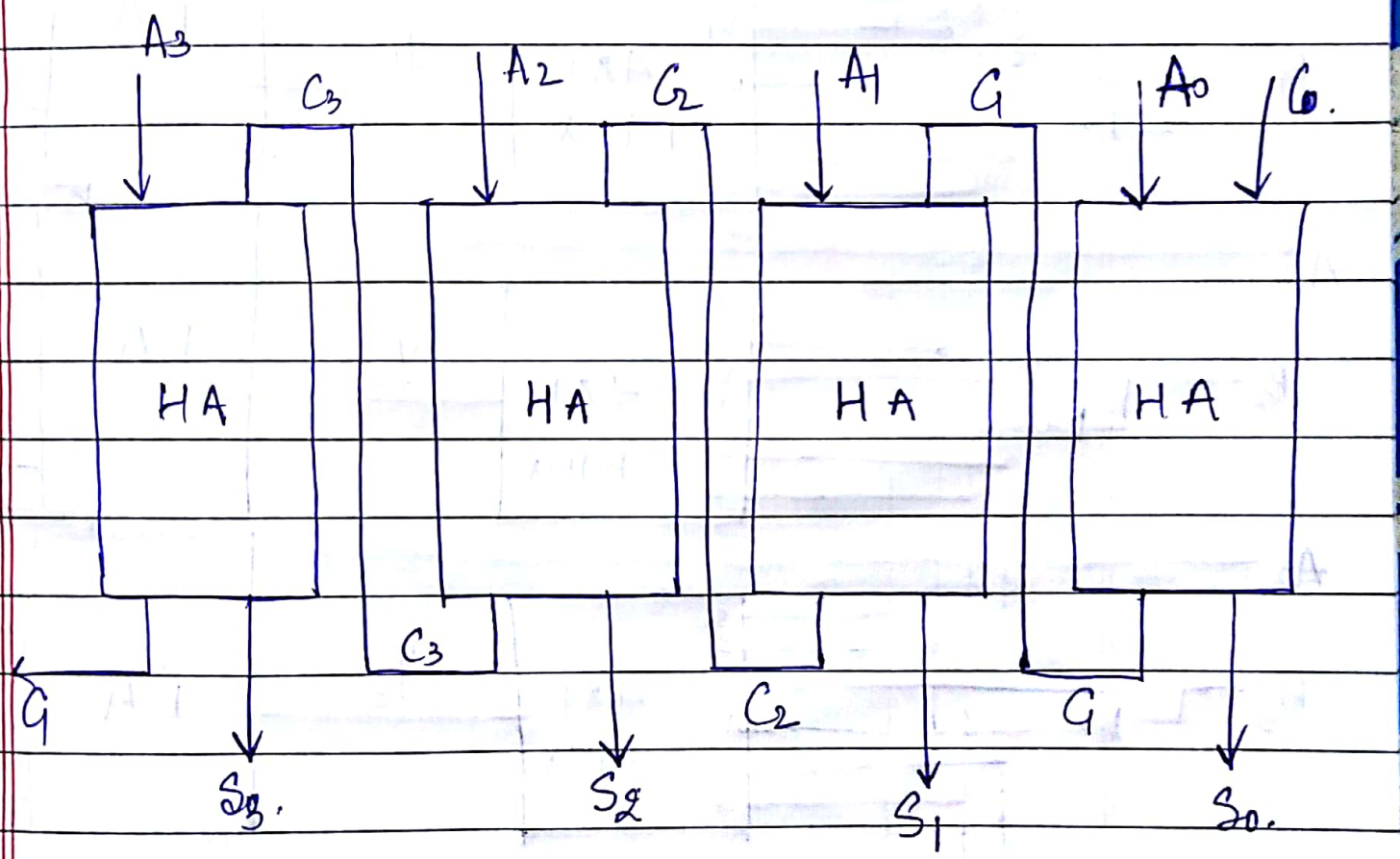
The sum shall be given by $S_3 S_2 S_1 S_0$
The out carry shall be given by C_4. C_1 is the input carry which shall be zero.

⊙ 4-bit adder / Subtractor



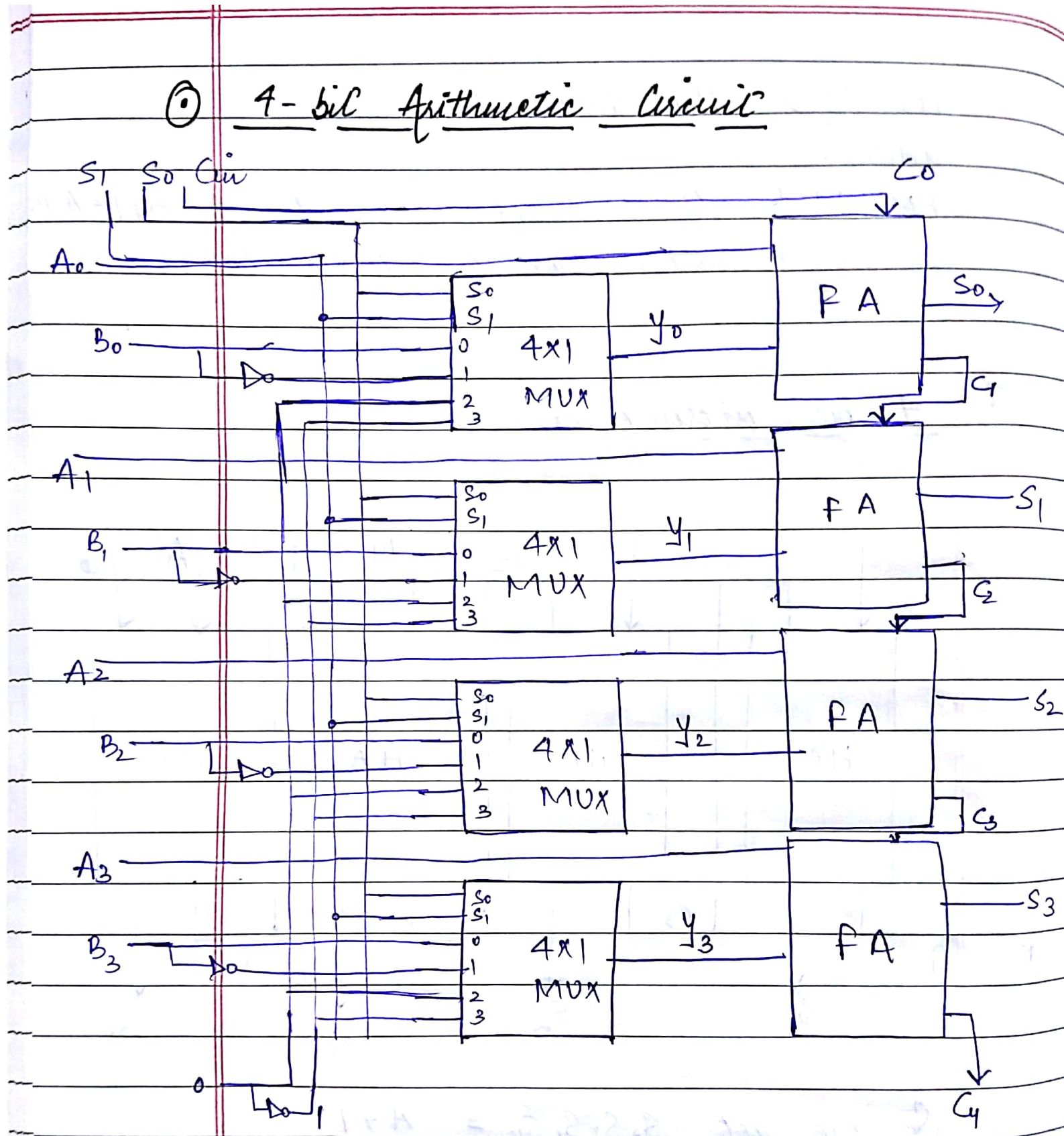
for M=0, the block diagram works as an adder.
For M=1, the result shall be $A+\bar{B}+1=A-B$ so it is working as a substractor.

⊙ 4-bit incrementor



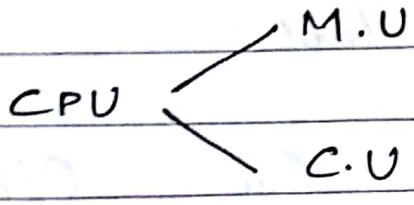
So we get $S_3S_2S_1S_0 = A+1$

⊙ 4-bit Arithmetic Circuit

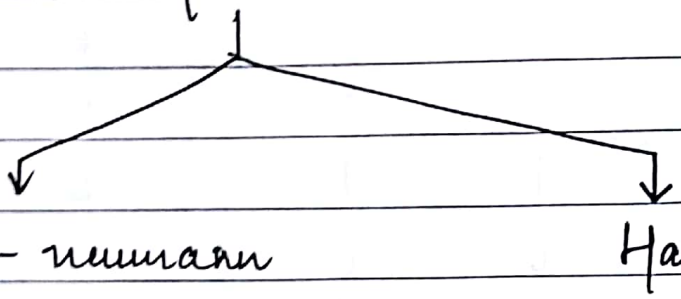


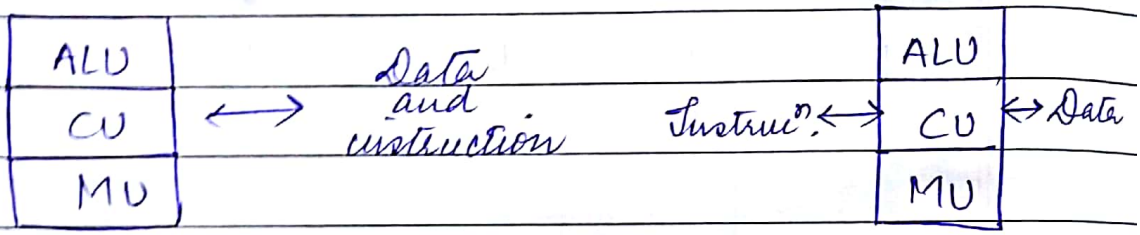
Function Table:

S_1	S_0	C_{in}	O/P Y for MUX	Function
0	0	0	B	A+B (Add)
0	0	1	B	A+B+1 (Add with carry)
0	1	0	$\overline{B}$	A+$\overline{B}$ (Subtraction with borrow)
0	1	1	$\overline{B}$	A+$\overline{B}$+1 (Subtraction)
1	0	0	0	A (Buffer)
1	0	1	0	A+1 (Incrementing)
1	1	0	1	A-1 (Decrementing)
1	1	1	1	A (Buffer)

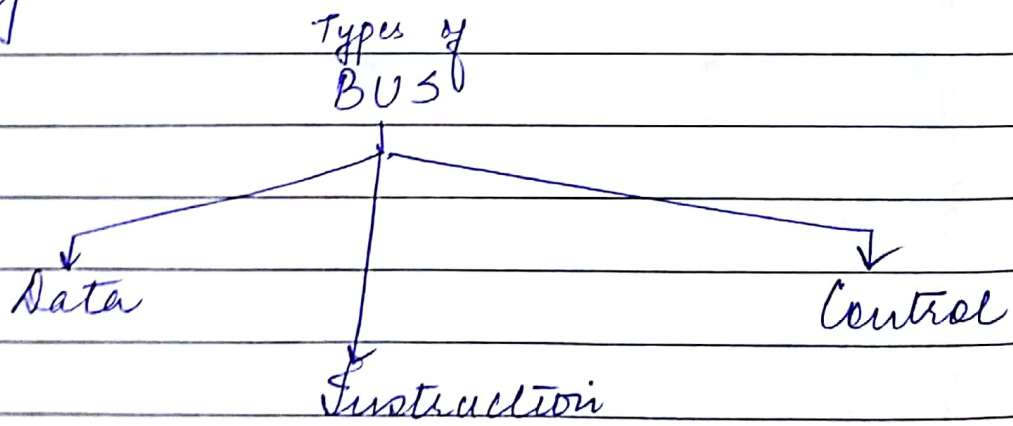


Computer Architecture

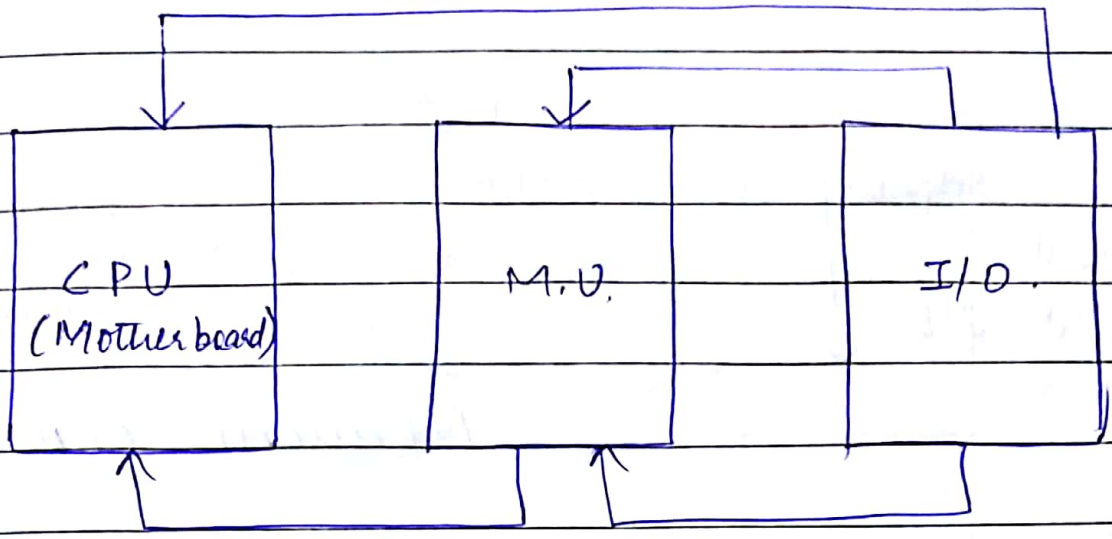




→ Bus is transfers data from keyboard to CPU.



→ Program is a set of instructions while process is the execution of program. Eg: C program.

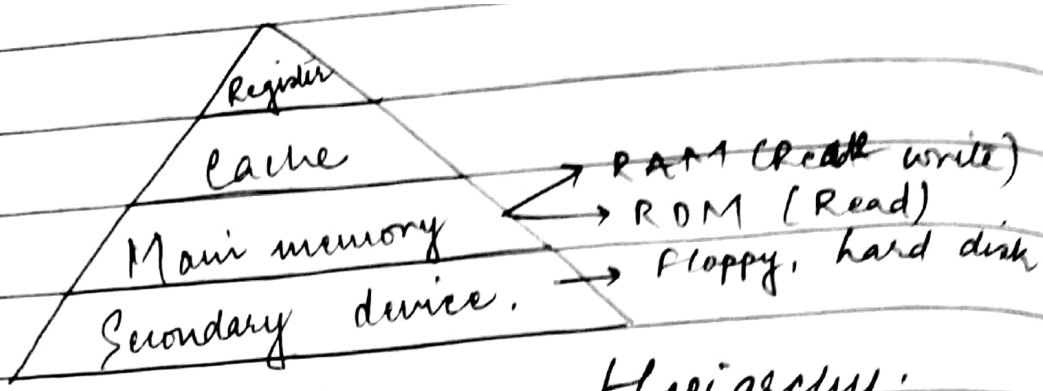


Computer Organisation

* 0/1 → Logic Gate → IC → Motherboard.

There are IC's in motherboard to convert from 1 number system to another.

Whenever an instruction is passed to CPU, an instruc^{n} cycle is made and is transferred to MU. In MU address is passed to instruc^{n} cycle & again info is received back to CPU.



Size.

8-16GB

500GB - 2TB.

Memory Hierarchy.

Size ↓ Speed ↑

→ The smallest storage device is register.

↓

microoperation

→ Flip flop is a storage device to store 1 bit. (i.e, 0/1)

(Min - 1 bit
Max - ?)

→ Register consists of flip-flop.

Register

↓

Flip flop

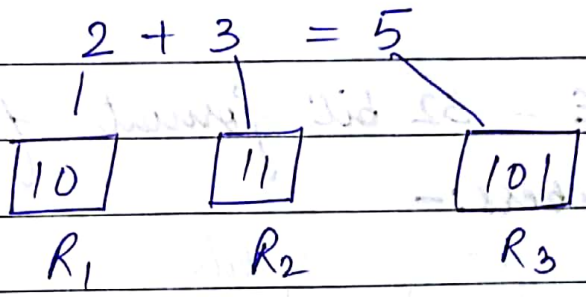
↓

0/1.

Register Transfer Lang. (RTL)

Lang. in the form of register.
or
Info that is passed in the lang. of registers.

eg:



$R_1 + R_2 \longrightarrow R_3$.

a = b.

$R_2 \leftarrow R_1$

* Register does not store data, it just processes data

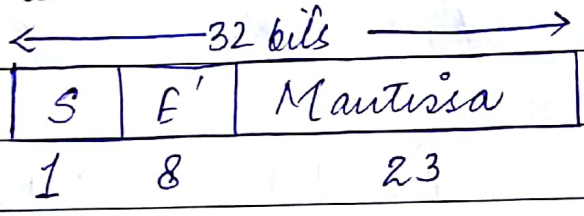
Cache consists useful & wanted data

⊙ Arithmetic

IEEE - 32 bit format - Single precision

IEEE - 64 bit format - Double precision

→ IEEE - 32 bit format for floating point numbers:-



Sign bit is for the sign of the number

It is 1 if the number is -ve.
It is 0 if the number is +ve.

E' is biased exponent = E + 127
↳ Exponent

Mantissa → It is the fractional part of the binary number.

Why biased?
The bias of +127 is added in order to represent the negative exponent.

$\{7.85 = 7.85 \times 10^{27}\}$

The maximum -ve exponent that can be represented is -127.
When we add +127 to it, it shall become zero.

Example:-

+124.75

2	124	0
2	62	0
2	31	1
2	15	1
2	7	1
2	3	1
2	1	
	0	

$\Rightarrow 01111100$

$.75 \times 2 = 1.50$

$.50 \times 2 = 1.00$

Not in normalised form.

$\Rightarrow (01111100.11)_2$

$1.1111001_1 \times 2^6$

8 bits

it is called normalised form

Exponent = 6.

Biased exponent = 6 + 127

= 133.

2	133	1
2	66	0
2	33	1
2	16	0
2	8	0
2	4	0
2	2	0
2	1	
	0	

$\Rightarrow 01000101$

Sign = 0.

S → sign of number, not sign of exponent

E' → exponent

S	E'	
0	10000101	11110011000000000000000

we added 15 zeros as it was 8 bits and mantissa requires 23 bits!

Zero is a CONVENTION that is used throughout the world as 32 bit

32 zeros will also represent a zero.

$BE = 133$

$Actual\ E = 133 - 127$

$= 6.$

$\rightarrow (0.75)_{10} = (0.11)_2$

$= (1.1 \times 2^{-1})$

$B.E = -1 + 127 = 126$

$= (01111110)_2.$

① Booth's Algorithm (Used for multiplying signed numbers in their 2's complement)

0 0 0 1 1 1 0

Multiplicand.

→ 1 0 1 0 1 0 ←

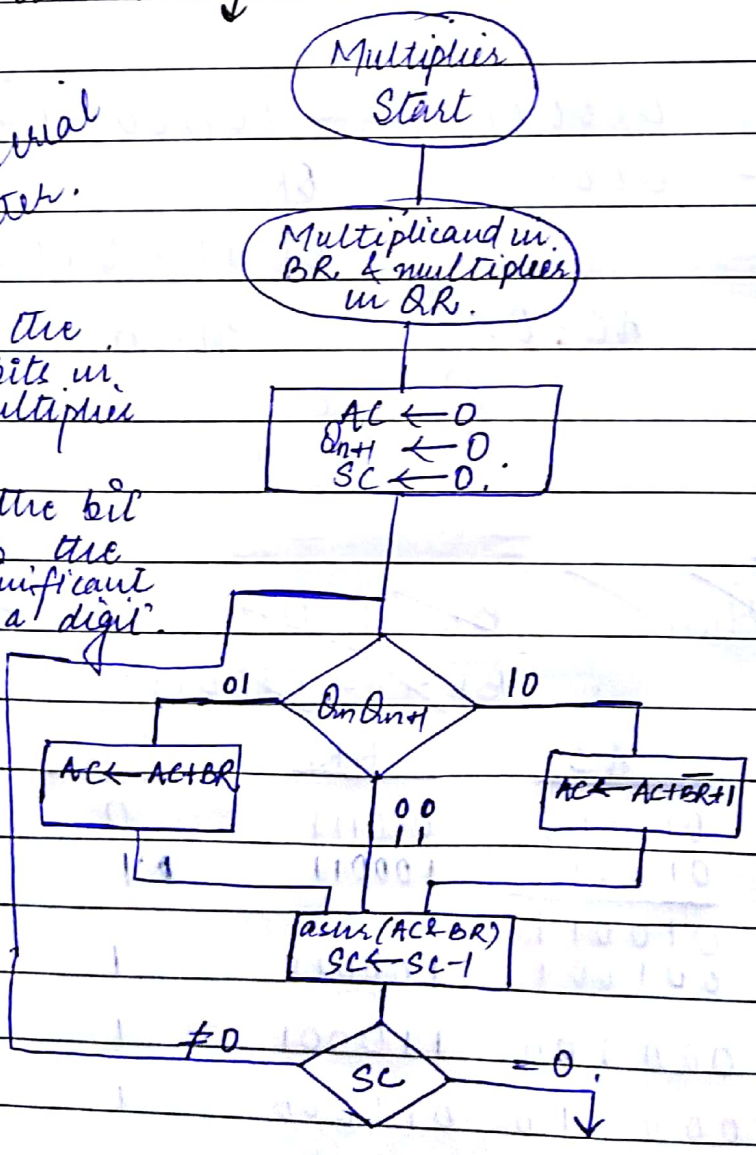
0 0 1 1 ← Multiplier

1 0 1 0 1 0

1 0 1 0 1 0 x

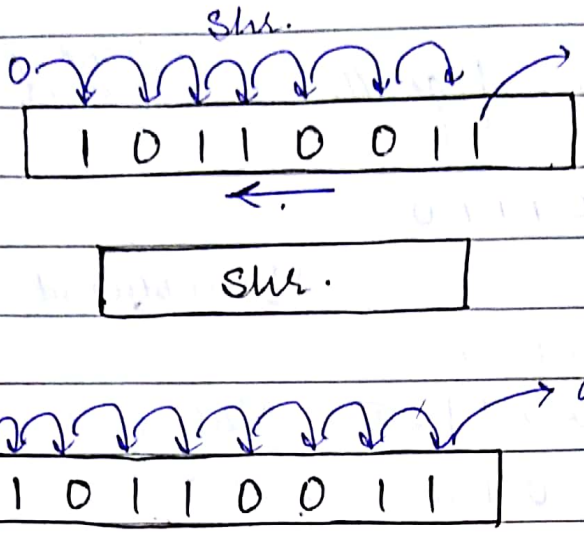
1 1 1 1 1 0

Flowchart ↴



SC is usual counter.

- n is the no. of bits in the multiplier
- Qn+1 is the bit next to the least significant bit in a digit.



Shr = Shift right

ashr = arithmetic shift right

⇒ −19 multiplicand
+7 multiplier
−133

$-19 = 010011 \xrightarrow{1's} 101100 + 1 \xrightarrow{2's} 101101$ = BR

$7 = 000111$ ← QR

$\overline{BR} + 1 = 010011$

$AC = 0$. $n = 6$.

$Q_{n+1} = 0$.

~~$Q_n Q_{n+1}$~~ ~~BR = 101101~~ ~~AC~~

~~$\overline{BR} + 1 = 010011$~~

$Q_n Q_{n+1}$	BR → 101101 $\overline{BR}+1$ → 010011	AC	QR	Q_{n+1}	SC
1 0		000000	000111	0.	0110
	AC = AC + $\overline{BR}+1$	010011			
		010011			
1 1	ashr	001001	100011	1	0101
1 1	ashr	000100	110001	1	0100
0 1	ashr	000010	011000	1	0011
	Add.	101101			
		0 1 1 1			

~~AC = AC + BR + 1~~

00	ashr	111011	101100	0	0010.
00	ashr	111011	110110	0	0001
10	ashr	111101	111011	0	0000

VERIFICATION:

$\Rightarrow$ 1111011111011

Take 2's complement to know the magnitude

1's comp. → 0000100001100

+1

00001000101

$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

M

$\Rightarrow 1 \times 2^7 + 1 \times 2^2 + 1 \times 2^0$

$= 128 + 4 + 1$

$= 133.$

① Single Precision IEEE Representation

S	Biased Exponent	Mantissa M

If the biased exponent is 0 or 255 then the following special provisions are made:-

(i) $E' = 0$; $M = 0$; The no. representation is $0 . \pm 0$.

(ii) $E' = 255$; $M = 0$; The no. represented is $\infty . \pm\infty$

(iii) $E' = 255$; $M \neq 0$; Then NaN Not-a-Number.

(iv) $E' = 0$; $M \neq 0$; then the no. represented as $\pm\ 0.M * 2^{-126}$.

⊙ Booth's Algorithm (Revised)

$33 = 0100001$

$-(11) = 0001011 \rightarrow 1110100$

$+1$

1110101

$\therefore (-11) = 1110101$

$BR = 0100001$

$\overline{BR}+1 = 1011111$

BR → 0100001
BR+1 → 1011111

Q_n Q_{n+1}		AC	QR	Q_{n+1}	SC.
1 0		0000000	1110101	0	0111
		+1011111			
		1011111			
	ashr	1101111	1111010	1	0110
0 1		+0100001			
		0010000			
1 0	ashr	0001000	0111101	0	0101
		+1011111			
		1100111			
	ashr	1110011	1011110	1	0100
0 1		+0100001			
		0010100			
	ashr	0001010	0101111	0	0011
1 0		+1011111			
		1101001			
		+1011111			
		1101001			
1 1	ashr	1110100	1010111	1	0010
	ashr	1111010	0101011	1	0001
	ashr	1111101	0010101	1	0000

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

↑

Since the result is -ve, we take 2's complement.

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

$2^8\ 2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$\Rightarrow 1 \times 2^8 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^3 + 1 \times 2 + 1$

$= 363.$

TUTORIAL 2

* To perform multiplication we add & shift.

* To perform division we subtract & shift.

Q. 1100

Left shift

⇒ 1000

Right shift

⇒ 0110.

In left & right shifts we lose data.

Circular left shift

⇒ 1001

Circular right shift

⇒ 0110.

In shifts, MSB doesn't change.

⇒ 0011

we check, if it is same then easily shifted.

∴ ⇒ 0110.

⇒ If it is not the same, then it is the cond^n of overflow.
Then we use XOR.

→ 2: Increment
3: decrement
0: Add
1: Subtract.
} Arithmetic Operation

* 00 & 11
One bit shift right

* SC: Sequence Counter (To count)
AC: Accumulator (To perform)
↓
initial value = 0.

Registers

* QR → Multiplier
BR → Multiplicand

AC & SC size depends on the size of BR. (bits)

Initially, $Q_{n+1} = 0$.
QR ki LSB $= Q_n$.

00 - ashr
01 - BR add
10 - 2's comp. (BR+1) add
11 - ashr.

classmate
Date
Page

⇒ +9 = 1001 (BR)

↓

0110

+1

-9 : 1011

13 → 1101 (QR)

↓

0010

+1

0011

-13 → 10011

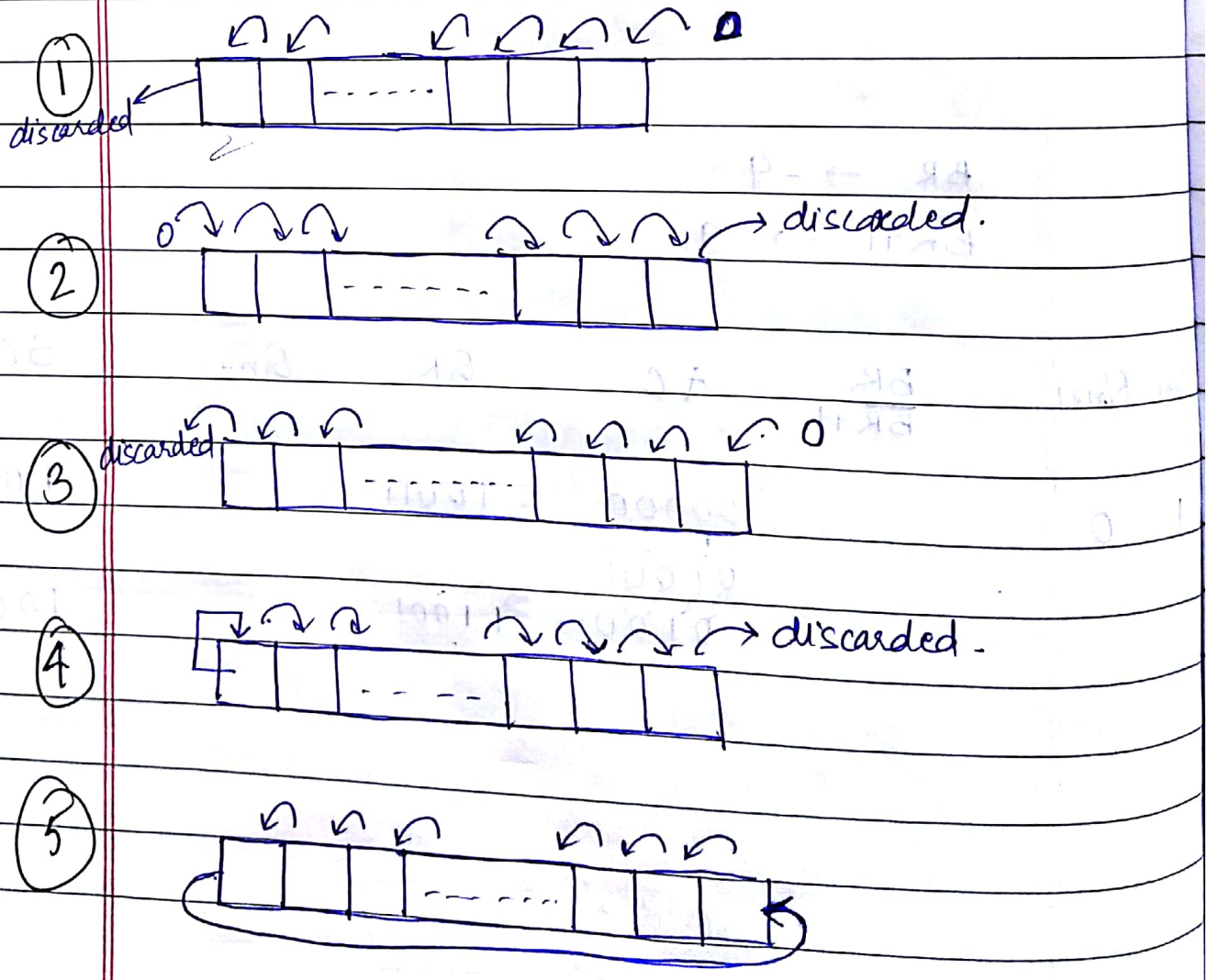
BR → -9

$\overline{BR+1}$ → 9.

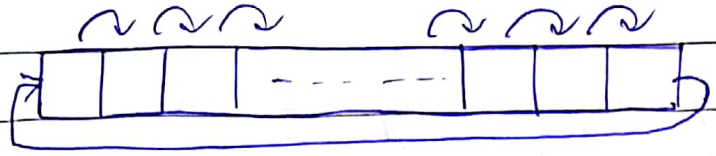
Qn Qn+1	BR / $\overline{BR+1}$	AC	QR	Qn+1	SC
					5
1 0		00000.	10011		101 0.
		+1001			
		01001			
		0100 →	11001		
					100

① Shift Microoperations

1. Logical shift left — $R \leftarrow shl\ R$
2. Logical shift Right — $R \leftarrow shr\ R$
3. Arithmetic shift left — $R \leftarrow ashl\ R$
4. Arithmetic shift right — $R \leftarrow ashr\ R$
5. Circular shift left — $R \leftarrow cil\ R$
6. Circular shift right — $R \leftarrow cir\ R$



6

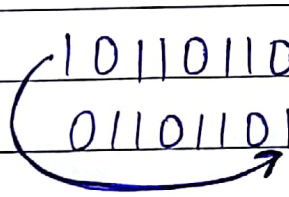


1. 11100011
   11000110
2. 10101101
   01010110
3. 10110010
   01100100

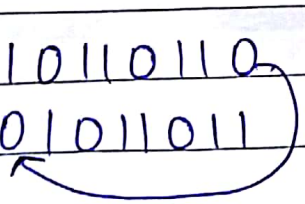
for arithmetic shift: * The sign of the number has changed. Therefore there is an overflow.

4. 10110010
   11011001

5. 10110110
   01101101



6. 10110110
   01011011

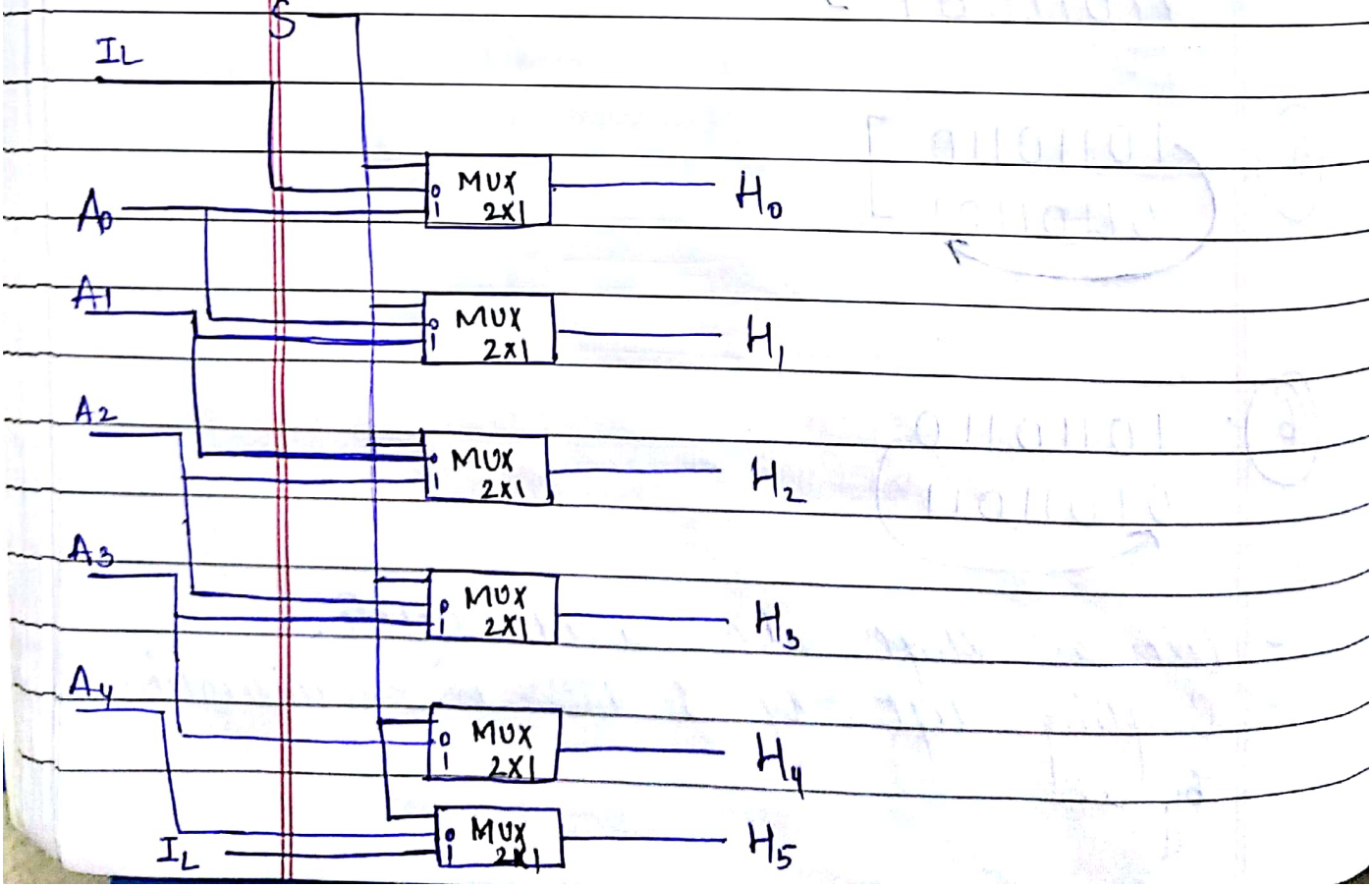


- Circular shift also called Rotate.
- Shifting left by 1 bit is multiplication by 2.

- Shifting right by 1 bit is division by 2.

Q. Draw the function table and combinational circuit for shifting left and shifting right by 1 bit.

	Selection Input	Output					
	S	H_0	H_1	H_2	H_3	H_4	H_5
Shift right	0	I_L	A_0	A_1	A_2	A_3	A_4
Shift left	1	A_0	A_1	A_2	A_3	A_4	I_L



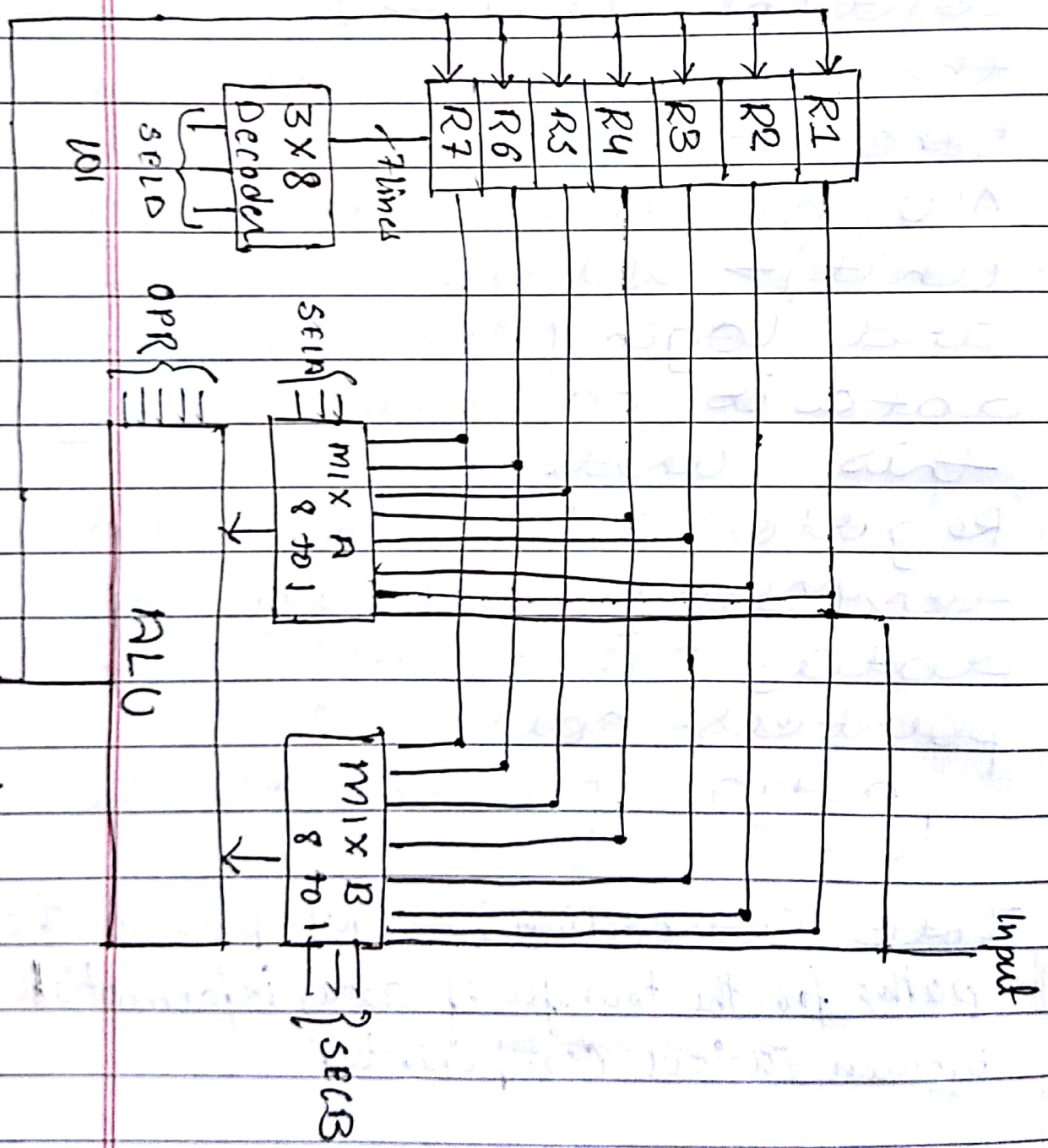
⊙ CPU

This is also called the brain of the computer. It consists of the following components :-

1. Control unit → It issues the control signals for all other components of the OS.
2. ALU (Arithmetic and logic unit) → All the arithmetic and logic processing of data is carried out by this unit.
3. Registers → These provide a temporary storage for data, partial results, pointers, counters, partial products etc.
4. Inter connection :— These provide the paths for the transfer of data / information between various components.

COMPUTER REGISTER ORGANIZATION:

Let us assume that there are 7 register and 32 operation that can be carried out.



Control word

3	3	3	5	= 14 bits
SELA	SELB	SELD	OPR	
001	011	101	00001	

$R_5 = R_1 + R_3$

Encoding of Registers:-

Code	SELA	SELB	SELD
000	Input	Input	Output
001	R_1	R_1	R_1
010	R_2	R_2	R_2
011	R_3	R_3	R_3
100	R_4	R_4	R_4
101	R_5	R_5	R_5
110	R_6	R_6	R_6
111	R_7	R_7	R_7.

Encoding of operation codes :-

Code	Meaning	Operation.
0000	TS PR	$R_1 \leftarrow R_1$
0001	INC	$R_1 \leftarrow R_1 + 1$
0011	ADD	$R_3 \leftarrow R_1 + R_3$
00100	SOB	$R_3 \leftarrow R_1 - R_3$
00101	AND	$R_3 \leftarrow R_1 \wedge R_2$
00110	OR	$R_2 \leftarrow R_1 \vee R_2$
00111	NOT	$R_2 \leftarrow \bar{R}_2$
		$R_3 \leftarrow 0$

Code:	Control Word
0000 →	001 000 111 00000
0001 →	001 000 001 00001
00110 →	001 111 010 00110
01000 →	011 011 011 01000

⊙ Stack Configuration

Stack is a part of memory or a set of registers of the CPU which is used to enter the items and to take out the item. For entry of items we use PUSH and for taking out items we use pop operation. It works on the principle of Last-in-first-out (LIFO) which means that the item (information) pushed out in the last is taken out (popped) first.

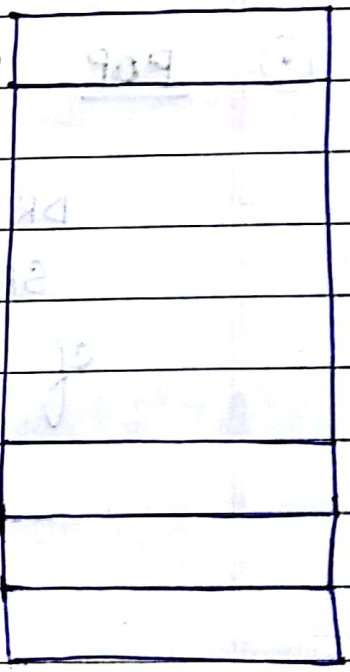
SP → Stack pointer (points the top of the stack)

DR → Data register (whatever item is to be pushed/popped is done through DR)

FULL

EMPTY

SP



DR.

Full and empty are two binary pointers.
If the stack is full then FULL = 1
If the stack is empty then EMPTY = 1.
To start with SP = 0.

For pushing (adding) the new item, we use PUSH which consists of the following microoperations.

SP = SP+1

M[SP] ← DR

If (SP=0) then (FULL ← 1)

EMPTY ← 0;

For taking out one item, we use POP which consists of the following microoperations:-

⊙ **POP**

DR ← M[SP];

SP ← SP-1;

If (SP=0) then (EMPTY ← 1);

FULL ← 0;

↓ operation

A+B

1.9.18

SC = n (No. of bits in QR)

Q_n = last bit of QR

Q_{n+1} = Sign bit of BR in the end

(-9)x(-13)

BR = 10111

$\overline{BR}+1$ = 01001

SC = 5 = 101

Q_n	Q_{n+1}	AC	QR	Q_{n+1}	SC
1	0	00000	10011	0	~~101~~
		01001			
		01001	10011	0	
		00100	11001	1	100
1	1	00010	01100	1	011
		10111			
0	1	11001			
		11100	10110	0	010
0	0	11110	01011	0	001
		01001			
1	0	*00111			
		00011	10101	1	000

⇒ 0001110101

1110001010 ⇒ $2^6+2^5+2^4+2^2+2^0$

+1

1110001011 = 117.

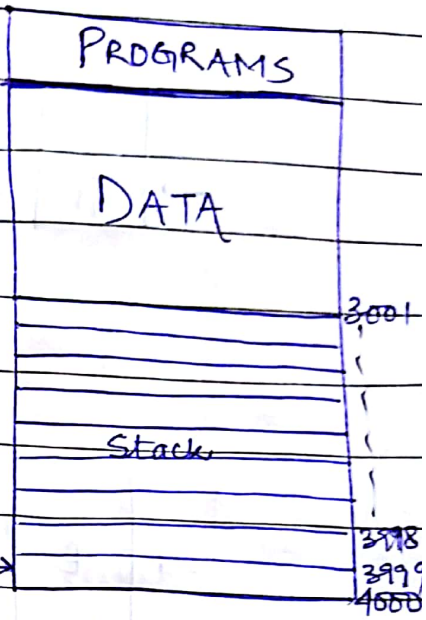
⇒ ~~XX~~

⑥ Stacks (Contd...)

SP may be pointing to the top of the stack which may be a place to push the next item.

OR

The SP may be pointing to the last item in the stack placed at its top.



Reverse Polish Notation

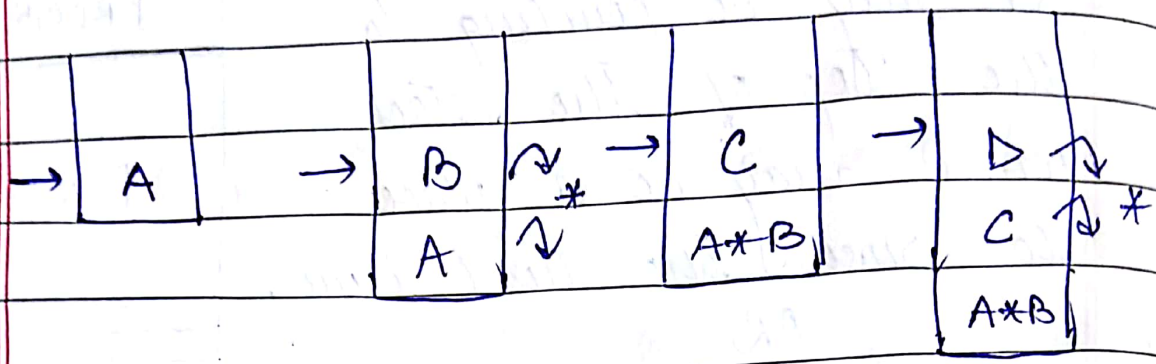
In mathematics, we generally write the expressions cannot be evaluated in one pass. If on the other hand, we write it in reverse polish (postfix) form, then it can be evaluated by traversing it only once.

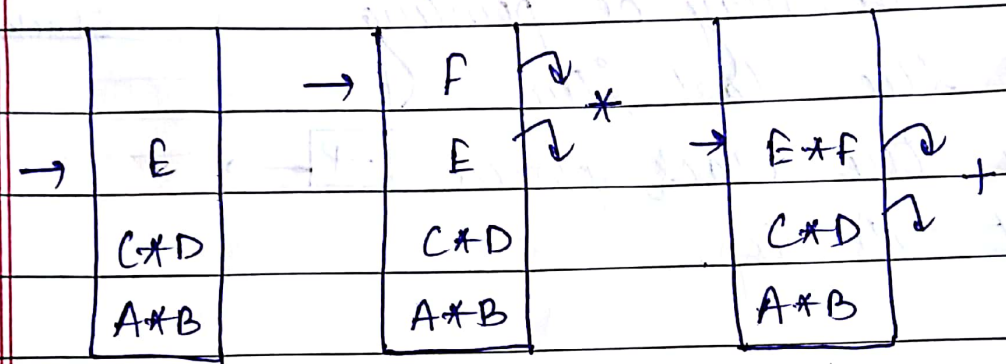
Q. Convert the following infix expressions into postfix:

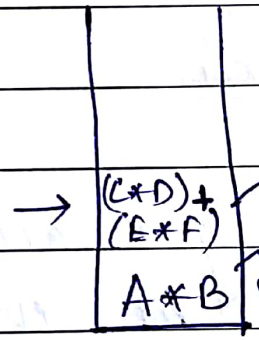
A*B+C*D+E*F

AB*CD*EF*++

→ AB*CD*EF*++







⇒ (A*B)+(C*D)+(E*F)

Q. Convert to Reverse Polish Notation

(i) A*B + A*(B*D+ C*E)

AB*ABD*CE*+*+

(ii) $A+B*[C*D+E*(F+G)]$

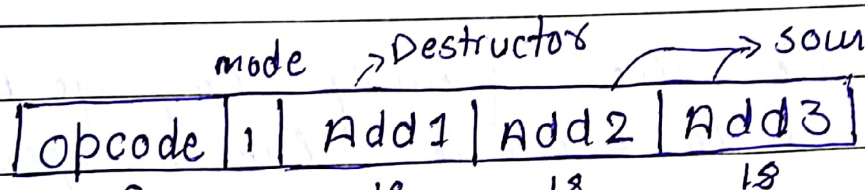
A B C D * E F G + * + * +

(iii) $A*[B+C*(D+E)]/(F*(G+H))$

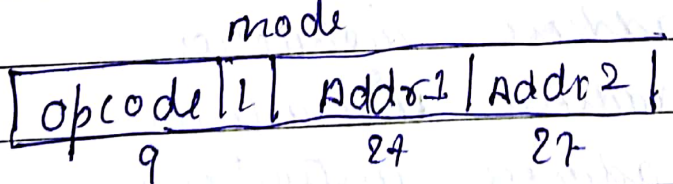
A B C D E + * + * F G H + * /

5/9/18

⊙ Instruction Formats



With 18 bits,
the memory that can be accessed $= 2^{18}$
$= 256$ KB.



The comparators can be categorised into the following 3 categories :-

1. Single accumulator organisation.
2. General register organisation.

3. Stack Register Organisation

Example:

(1) ADD X ACC ← ACC + M[X]

(2) ADD R1 ← R2 + R3

(3) Zero address instruction like ADD, AND.

*

Q. Write the Assembly language instruc-ions to evaluate the Expression:

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

By using,

(i) 3 - address instructions
(ii) 2 - address instructions
(iii) 1 - address instructions
(iv) 0 - address instructions

Where,
A, B, C, D, X are memory addresses.

lion.

ADD R1 A B ; R1 ← M[A] + M[B]

ADD R2 C D ; R2 ← M[C] + M[D]

MUL X R1 R2 ; M[X] ← R1 * R2

2- Address

e.g: ADD R1, R2 ; R1 ← R1 + R2

MOV R1 A ; R1 ← M[A]

ADD R1 B ; R1 ← R1 + M[B]

MOV R2 C ; R2 ← M[C]

ADD R2 D ; R2 ← R2 + M[C]

MUL R1 R2 ; R1 ← R1 * R2

MOV X R1 ; M[X] ← R1

11.9.18

① 1 - Address Instructions

In this case, one address is supplied for one operand within the instruction. The other operand is supplied and is given by the accumulator (most frequently used register inside CPU). The accumulator also provides the destination for the result.

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

where,
X, A, B, C, D are address of memory locations.

LOAD	A ;	ACC ← M[A]
ADD	B ;	ACC ← ACC + M[B]
STORE	T ;	M[T] ← ACC
LOAD	C ;	ACC ← M[C]
ADD	D ;	ACC ← ACC + M[D]
MUL	T ;	ACC ← ACC * M[T]
STORE	X ;	M[X] ← ACC.

* As the no of addresses are decreasing, the no. of instructions are increasing.

① 0-Address Instructions

In this case, we use stack. The data is pushed onto the top of stack (TOS) and is taken out from the top of the stack. For any arithmetic or logic operations, the top two locations of the stack are used. After the operation, the result is stored on the top of stack.

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

PUSH A	TOS ← M[A]
PUSH B	TOS ← M[B]
ADD	TOS ← M[A] + M[B]
PUSH C	TOS ← M[C]
PUSH D	TOS ← M[D]
ADD	TOS ← M[C] + M[D]
MUL	TOS ← (M[A]+M[B]) * (M[C] + M[D])
POP X	M[X] ← TOS.

We can see that as we decrease the number of addresses in the instructions, the number of instruct-

PROCEDURE

1. Write the two numbers in the normalized form.
2. Multiply the floating point parts (components).
3. Add the exponents.
4. Adjust the floating to convert the result to normalized form.

Eg: $(1.0110 * 2^{-2}) * (1.0010 * 2^{2})$

$(1.0110 * 1.0010) * (2^{-2} * 2^{2})$

```
  1.0110
  1.0010
  0 0000
 10110X
1011 0XXXX
-----------
1 1000 1100
```

$\Rightarrow 1.100011 * 2^{-2+2}$

$\Rightarrow 1.100011$

=

12.9.18

Program counter contains the address of the next instruction to be executed.

⊙ Addressing Modes

It is a technique which specifies the method to compute the address of the operand. There can be different methods to specify the addresses. The addresses may be memory addresses or could be register addresses. The following modes are generally used:-

1. Implied mode
The address is not specified explicitly. The address is implied. Like in case of stack operations, we need not specify the address. The operands are in the top two locations. Similarly in case of complementary, the contents of the accumulator, the address may not be specified.

2. Immediate mode
In this case also, the address is not given and in its place, the value of the operand (data) is directly provided inside the instruction. eg: ADI 32H

3. Register mode

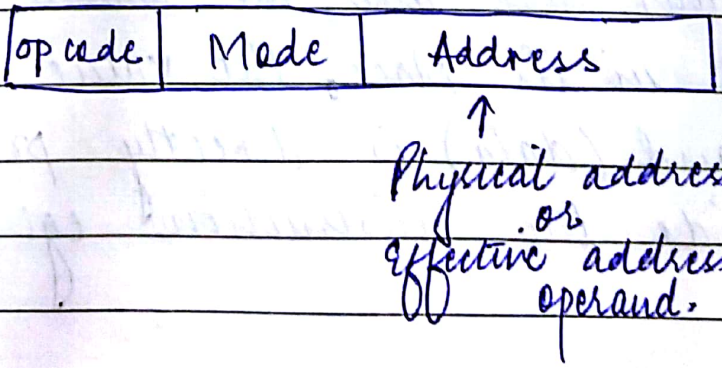
In this case, the address of the register is specified instead of the address of the memory. If we use k bits to specify the register, then 2^k registers can be addressed.

4. Register Indirect Mode.

In this case, the register address is specified inside the instruction but the contents of the registers are not the data but the memory address where the data is lying. This is used to shorten the length of the instruction

5. Direct Addressing Mode

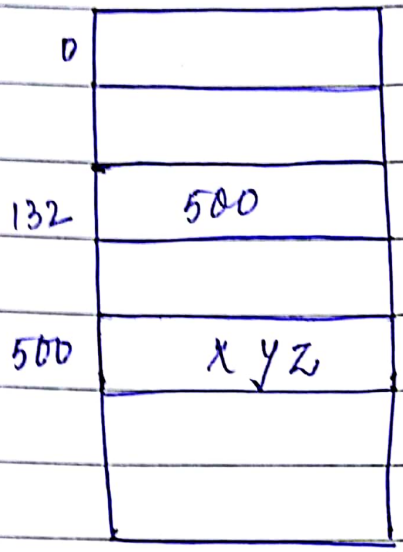
In this case, the address of the operand which is inside the memory is directly given inside the instruction.



The operand is fetched from the memory for the execution of the instruction.

→ Address of the address.

6. Indirect addressing



In this case, the address of the operand is not directly supplied inside the instruction. The address given inside the instruction contains the address of the operand. So we can say that we specify the address of the address of the operand.

For eg:

If 132 is specified in the instruction and the address 132 contains 500 as its data. Here 500 is the address of the operand. XYZ is the data (operand) in this case.