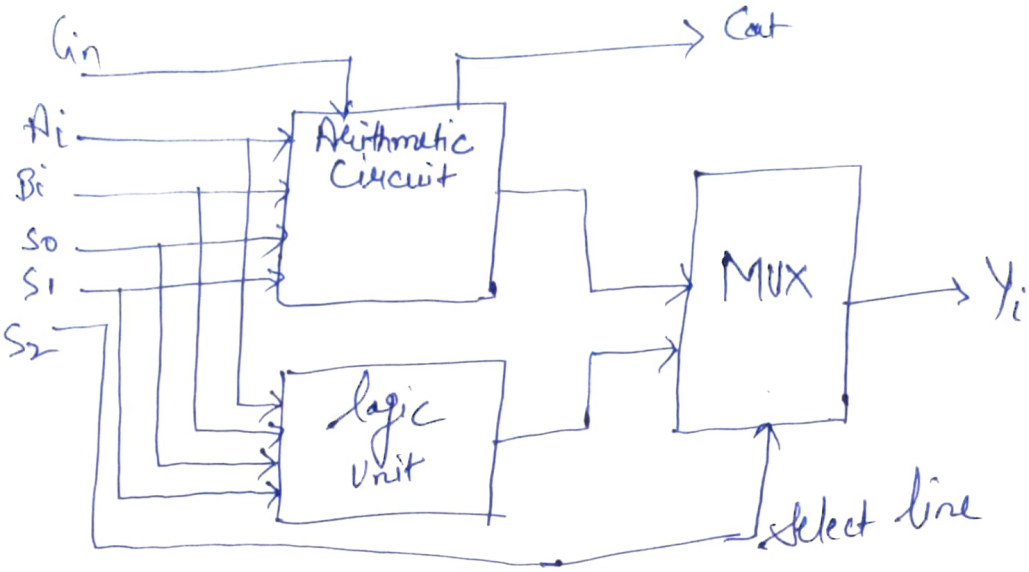
Arithmetic and logic Unit Design.

Describe the sequential Arithmetic & logic Unit (ALU) using suitable diagram.



1. With the aid of Multiplexer, we may Combine arithmetic & logic circuit to Create arithmetic & logic unit
2. When the mode Select line $S_2 = 0$, this ALU acts as an arithmetic circuit, so the output of arithmetic circuit is transferred as final op.
3. otherwise $S_2 = 1$, the output of logic circuit is transferred.
4. Based on the Mode Select S_2 & input Carry we Can increase or decrease the number of arithmetic & logic operations.

when $S_2 = 0$, the ALU performs arithmetic operation & when $S_2 = 1$, with $C_{in} = 0$, the ALU Performs logic operation.

We know that the Carry i/p is not required in logic circuits.

When logic operation is selected ($S_1 = 1$) the Carry input must be Zero.

Thus given us the output sum in full adder circuit as.

$$Y_i = A_i \oplus B_i \oplus C_i \quad \because C_i = 0$$
$$Y_i = A_i \oplus B_i$$

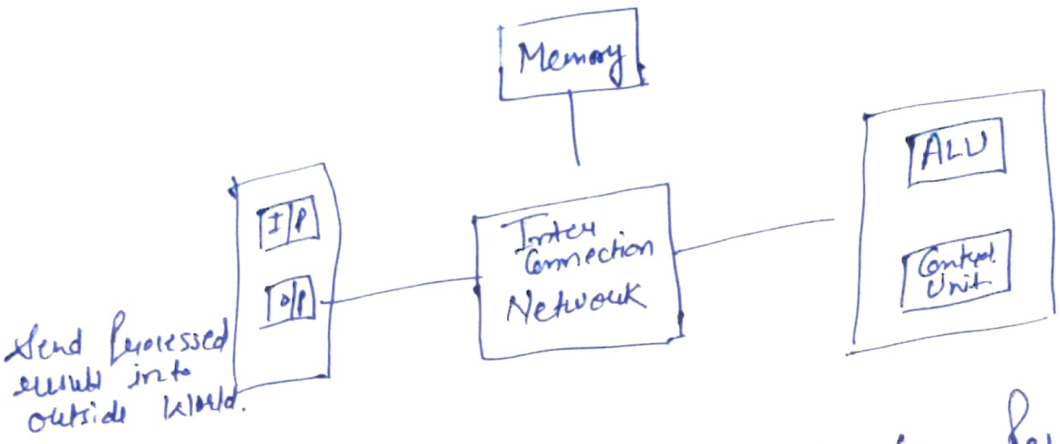
Q Explain Booth's Algorithm in details.

Q Show step by step the multiplication process using Booth's Algorithm when (+15) & (-13) numbers are multiplied.

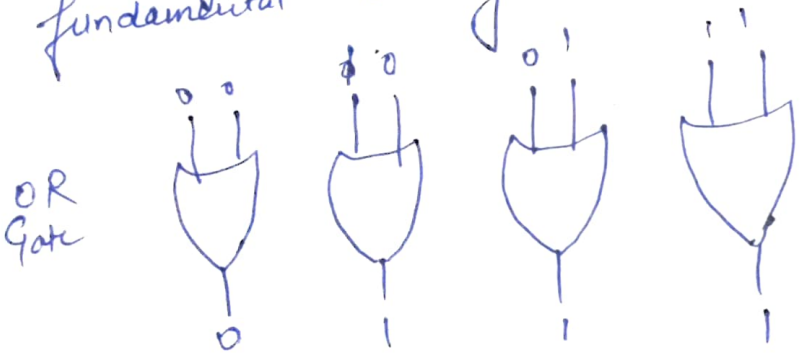
→ Assume 5-bit registers that hold signed No.

15 → 01111
13 → 10011

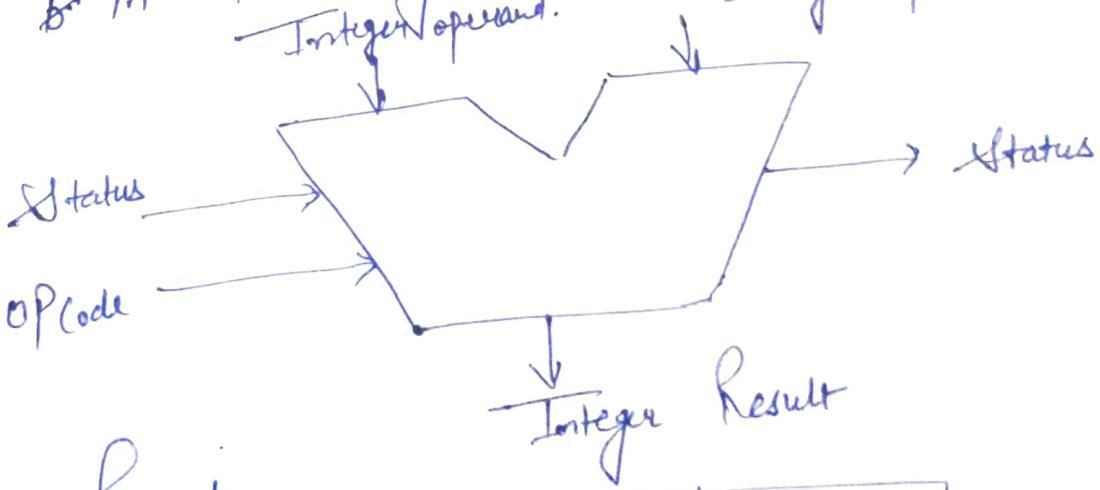
Q Show the Contents of E A Q SC during the process of multiplication of two binary No. 11111 (multiplicand) 10101 (multiplier. The sign are not included.



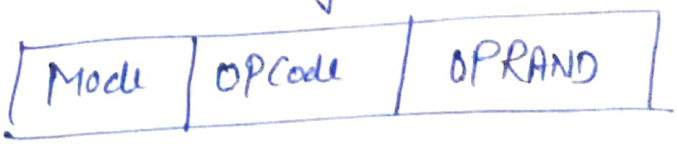
ALU is digital Circuit used to perform arithmetic & logic operations. It represents the fundamental building block of CPU



All information in Computer is stored in & manipulated as in the form of 0 and 1



Instruction format



The opcode part of the instruction format defines the operation to be performed.

Mode → Addressing Mode is

The data can be stored in the m/m of a computer or it can be located in the register of CPU

Carry look Ahead Adder →

The adder produce carry propagation delay while performing other arithmetic operation like multiplication & division as it uses several addition or subtraction steps.

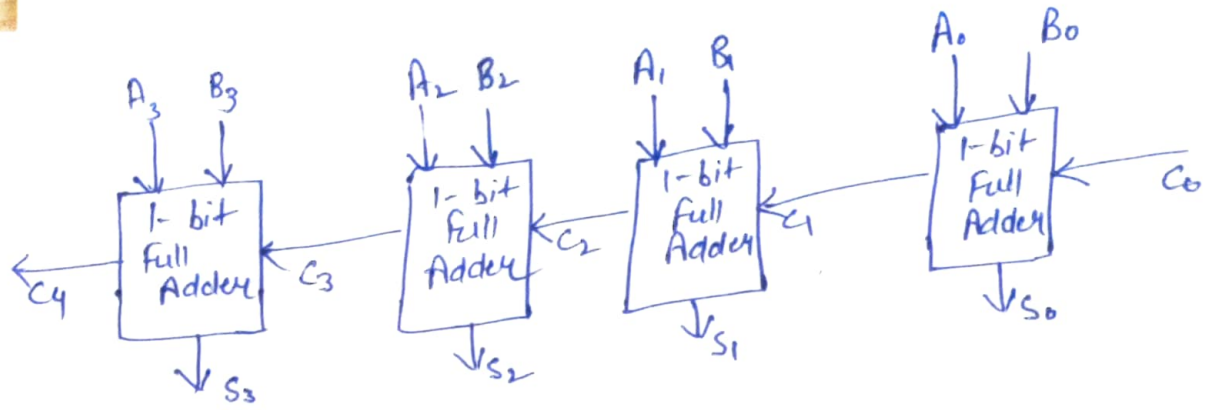
This is major problem for the adder & hence improving the speed of all arithmetic operation.

→ To overcome the carry propagation delay one widely approach is employ a carry-look ahead which solves the problem by Calculating the carry signal in advanced

Here carry signal will be generated in two cases

→ Input bit A & B are 1.

→ when one of the two bits is 1 & the carry-in is 1.



In Ripple Carry adders, for each block, the two bits that are to be added are available instantly

→ However, each block waits to Carry to arrive from its previous block, so it is not possible to generate the Sum & Carry of any block until the input carry is known.

→ ith block waits for the i-1th block to produce its Carry.

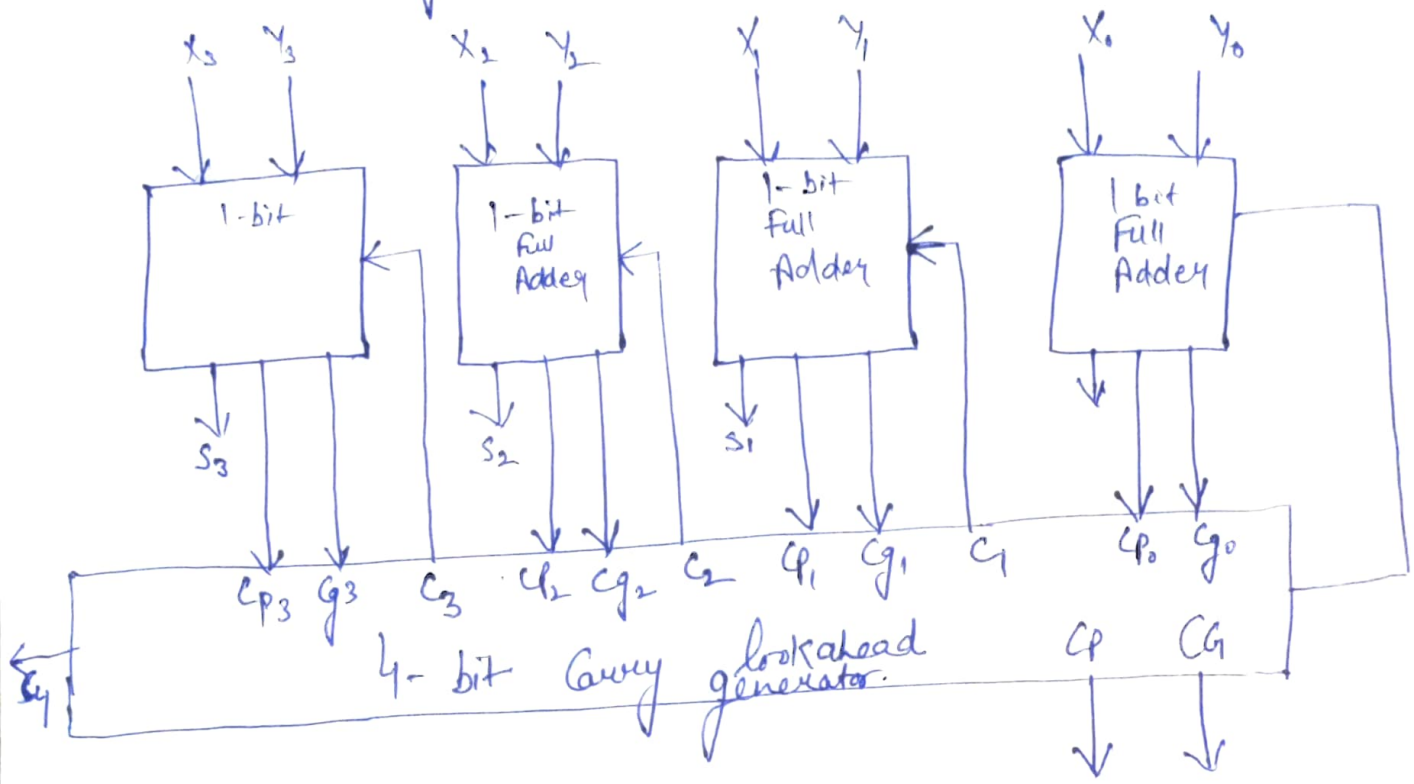
Propagation time is equal to the propagation delay of each adder block, multiplied by the no. of adder blocks in the circuit

Propagation time = Propagation delay X No. of adder block.

e.g 20 X 3 · 60 nanoseconds.

→ Carry look added reduces the propagation delay by introducing more Complex h/w

Block Diagram →



The Carry output Boolean Function of each stage in 4 stage Carry look - ahead adder

$C_1 = G_0 + P_0 C_{in}$

$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$

$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$

$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$

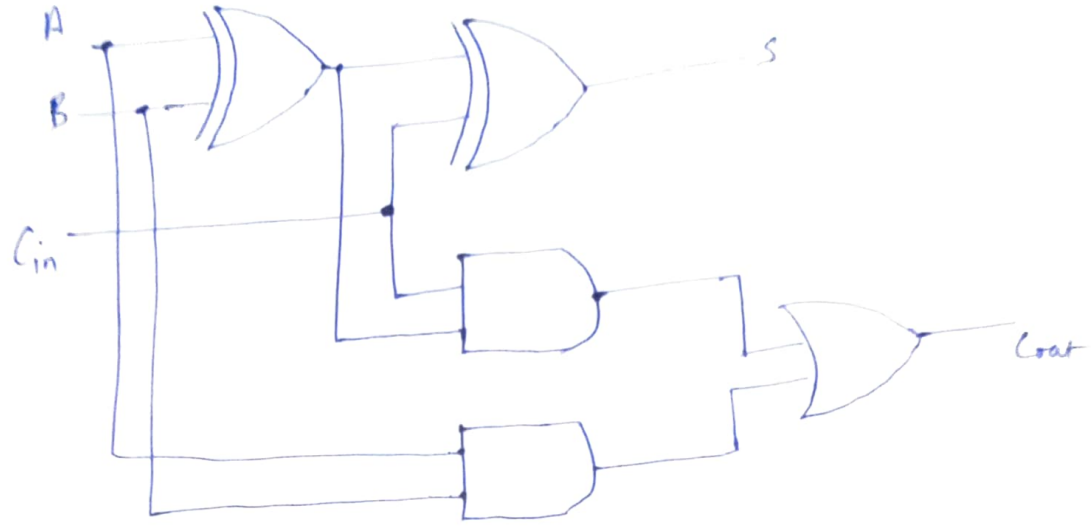
Propagation delay is reduced.

Provide fastest addition logic

→ gets Complicated as No. of Variable increase

→ Costlies as it involves more hardware.

Implementation Algorithm in Signed Magnitude



A	B	C	C+1	Condition
0	0	0	0	No Carry generate
0	0	1	0	
0	1	0	0	
0	1	1	1	No Carry Propagate
1	0	0	0	
1	0	1	1	
1	1	0	1	Carry generate
1	1	1	1	

We define two Variable Carry generate G_i & Carry Propagate P_i

$P_i = A_i \oplus B_i$

$G_i = A_i B_i$

$S_i = P_i \oplus C_i$

$C_{i+1} = G_i + P_i C_i$

The Sum output and Carry output can be expressed in terms of Carry generate G_i & P_i

Multiplication Algorithm in Signed Magnitude Representation.

Multiplication of two fixed point binary number in signed magnitude representation is done with process of successive shift & add operation

In multiplication process, we are considering successive bits of the multiplier, least significant bit first.

→ If the multiplier bit is 1, the multiplicand is copied down else 0's are copied down.

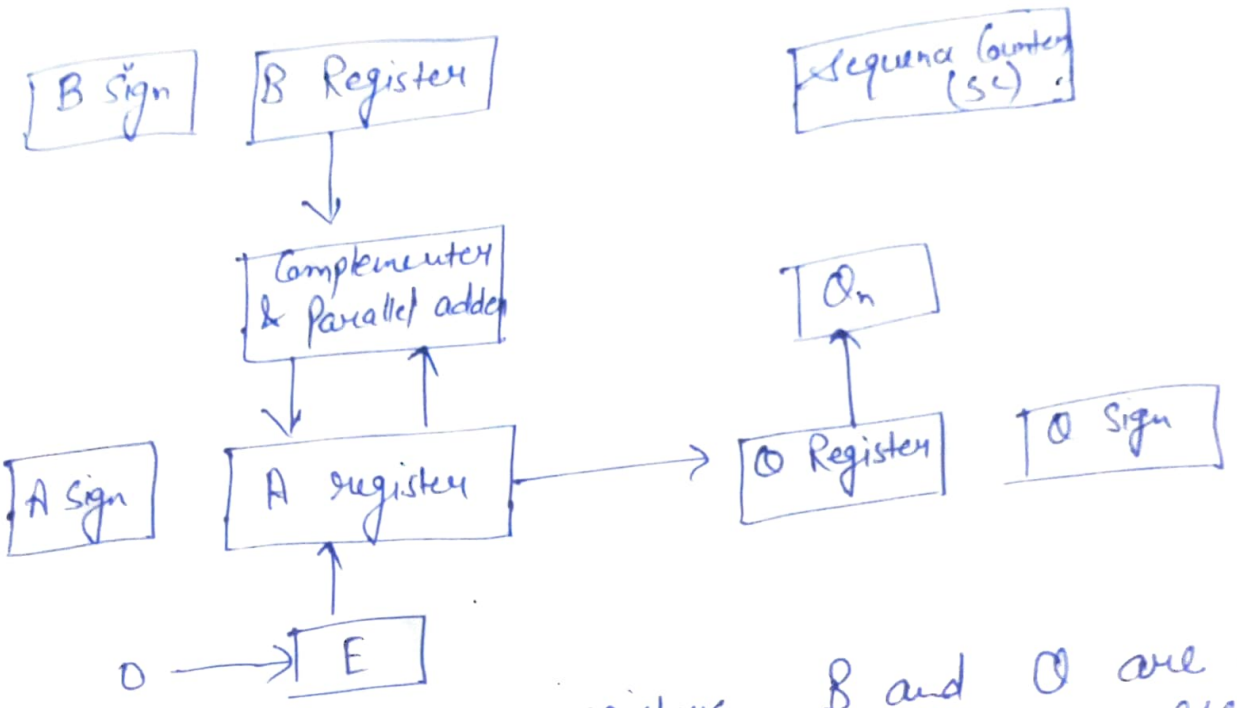
→ The no. copies down in successive lines are shifted one position to the left from the previous no.

→ Finall No. are added & their sum form the product.

→ The sign of the product is determined from the sign of multiplicand & multiplier if they are alike, sign of the product is positive else Negative.

```
  10111   (Multiplicand)
  10011   Multiplier
  -----
  10111
 10111
00000
10111
--------
01101110  (Product)
```

Hardware Implementation →



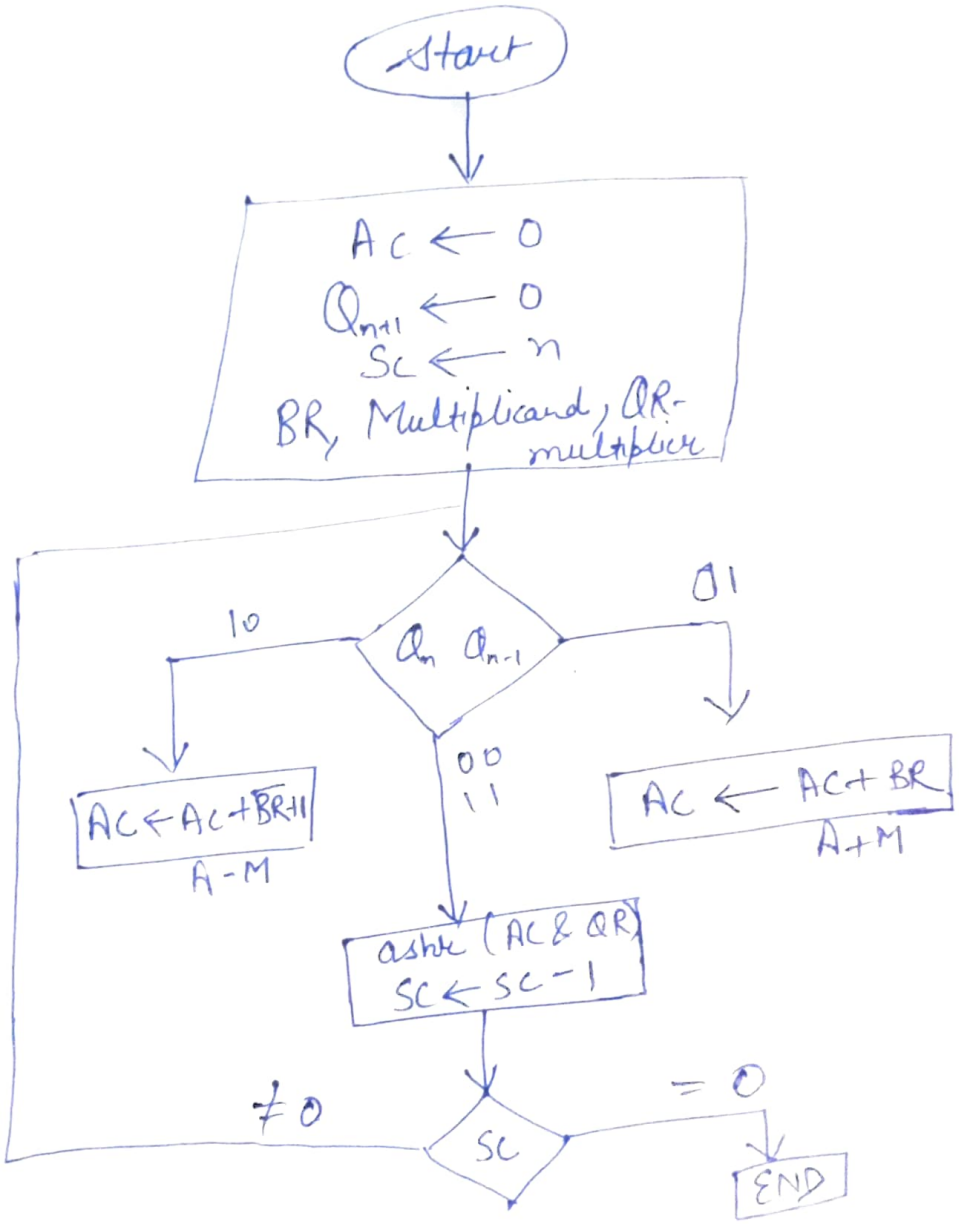
1. Registers: Two registers B and Q are used to store multiplicand & Multiplier respectively. Register A is used to store partial product during multiplication.
SC is used to store no. of bits in the multiplier.

2. Flip-Flop → To store sign bit of registers we required three flip-flops (A sign, B sign & Q sign)
flip-flop E is store Carry bit generated during Partial Product addition.

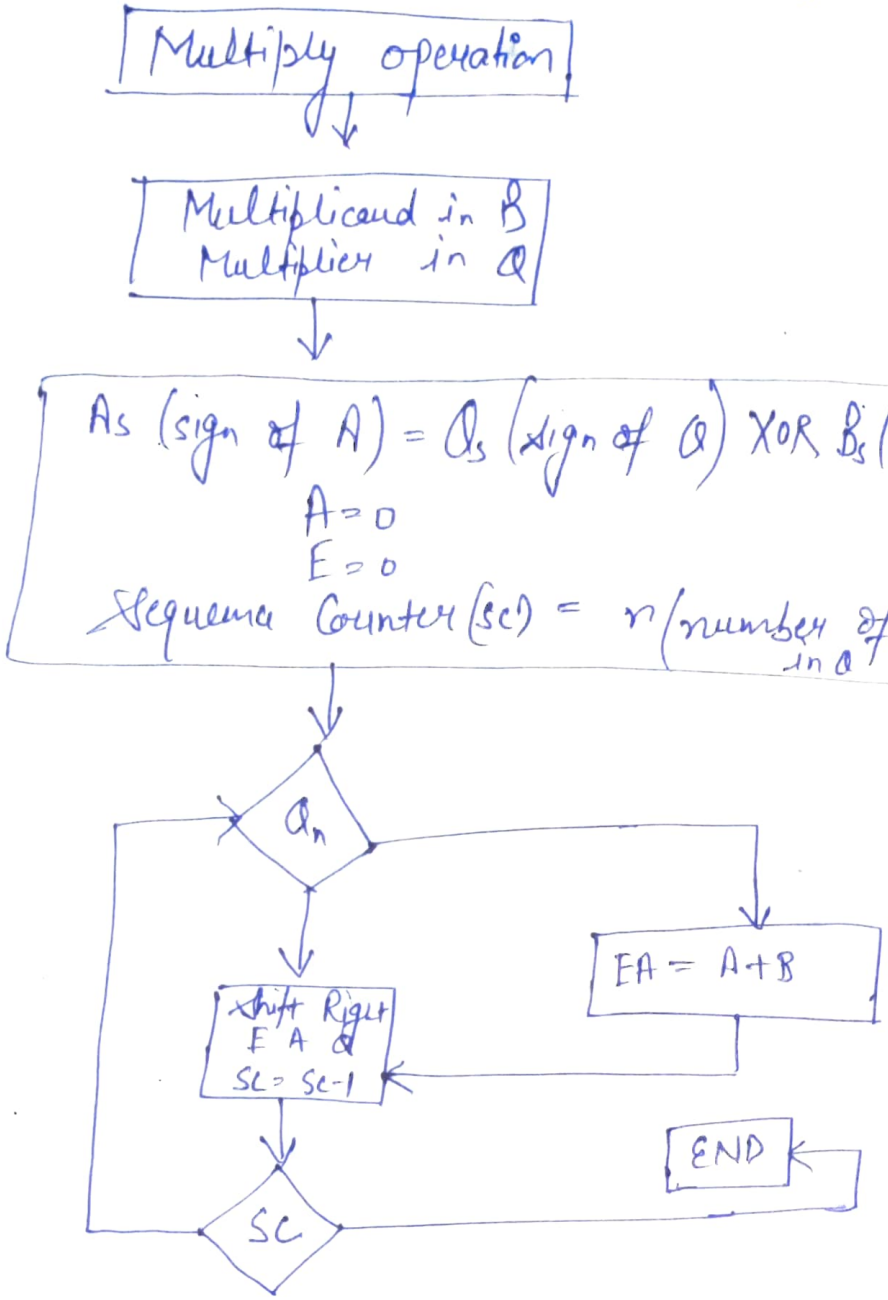
3. Compliment & Parallel adder:-
This h/w unit is used in Calculating Partial Product i.e perform addition required.

Booth's Multiplication Algo

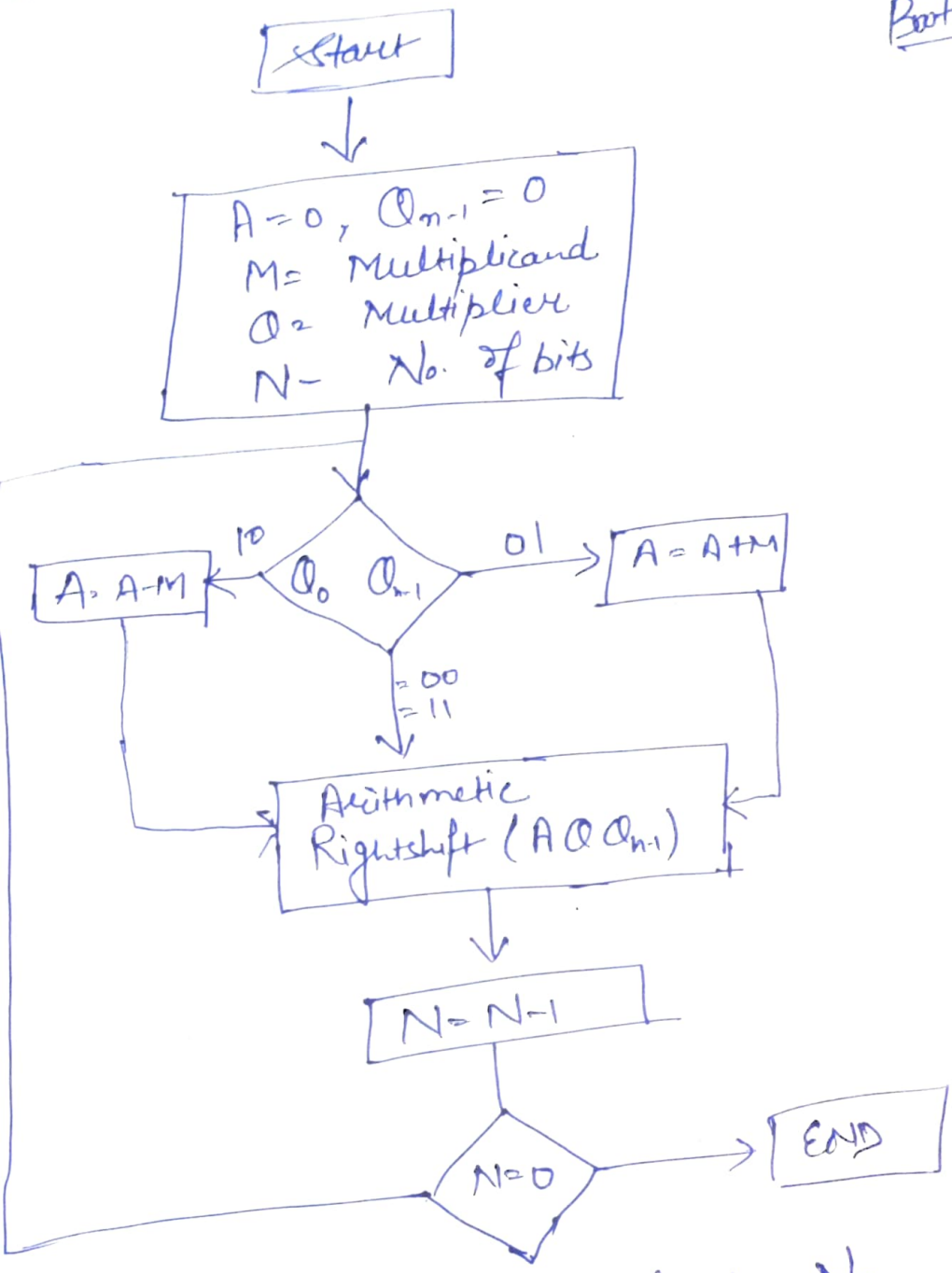
- that allows us to multiply the two signed binary integers in 2's Compliment, uses.
- → Also used the speed up performance of the multiplication process.



Flowchart of Multiplication Algorithm



Booth's Algorithm



Multiplication of two signed No.

Ex $7 \times 3 = 21$

$0111 \times 0011 =$

M Q

16 8 4 2 1 = 21

1 0 1 0 1

Q_3 Q_2 Q_1 Q_0	Q_{-1}	Q_{n-1}	Operation
A	Q		
1) 0000	0011	1	
1001	0011	0	A - M = A + 2's M
2) 1100	1001	1	0000 + 1001 = 1001
			RS

0111
1000
1001

	1100	1001	1
(3)	1110	0100	1
	0101	0100	1
(4)	0010	1010	0
	0001	0101	0

168421
10100

Shift operation

A + M = A → 1110 + 0111 = 0101

RS

RS

= 21

N = 0

-5 = 1011 = B

0000
1011
0100
1
0101

Ex 2

-5 X 4 = 20

	A	Q	Q_{-1}
	0000	0100	0
(1)	0000	0010	0
(2)	0000	0001	0
(3)	0101	0001	0
	1010	1000	1
	0101	0	

operation.

Initial

RS

RS

A = A-M

RS

A = A+M

1010
1011
10101

0000
0001
0010
0011
0100

Multiplicand (B) = 1011

-10 X -4

-5

168421
5 0101
10100
10100
0100

1010
1
1011
1011
1110

0111
168421
1010
0101
1
0110
0100
1011
1
1100

Division (Restoring and Non-Restoring)

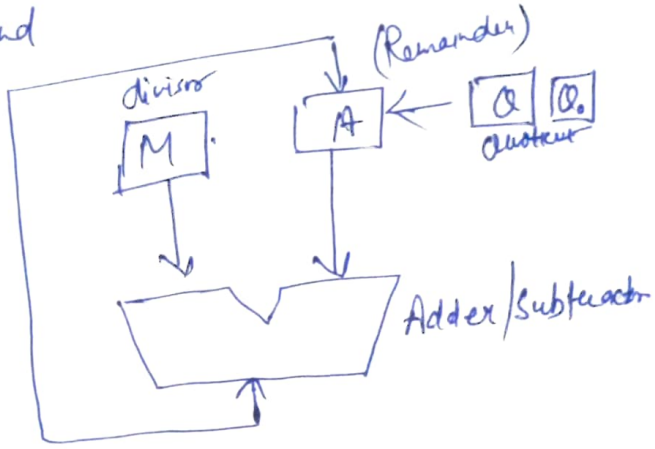
```
          14   ← Quotient
Divisor → 12 | 169  ← dividend
           -12
           ----
            49
           -48
           ----
             1   ← Remainder
```

Division is More Complicated than Multiplication

12 = 1100

169 = 10101001

```
          00001110  → Quotient
1100   | 10101001  ← Dividend
       - 1100
         -----
         01001
          1100
          ----
           1100
           1100
           ----
           00001
         - 0000
           -----
           0001
```



Restoring Division (Hardware Implementation)

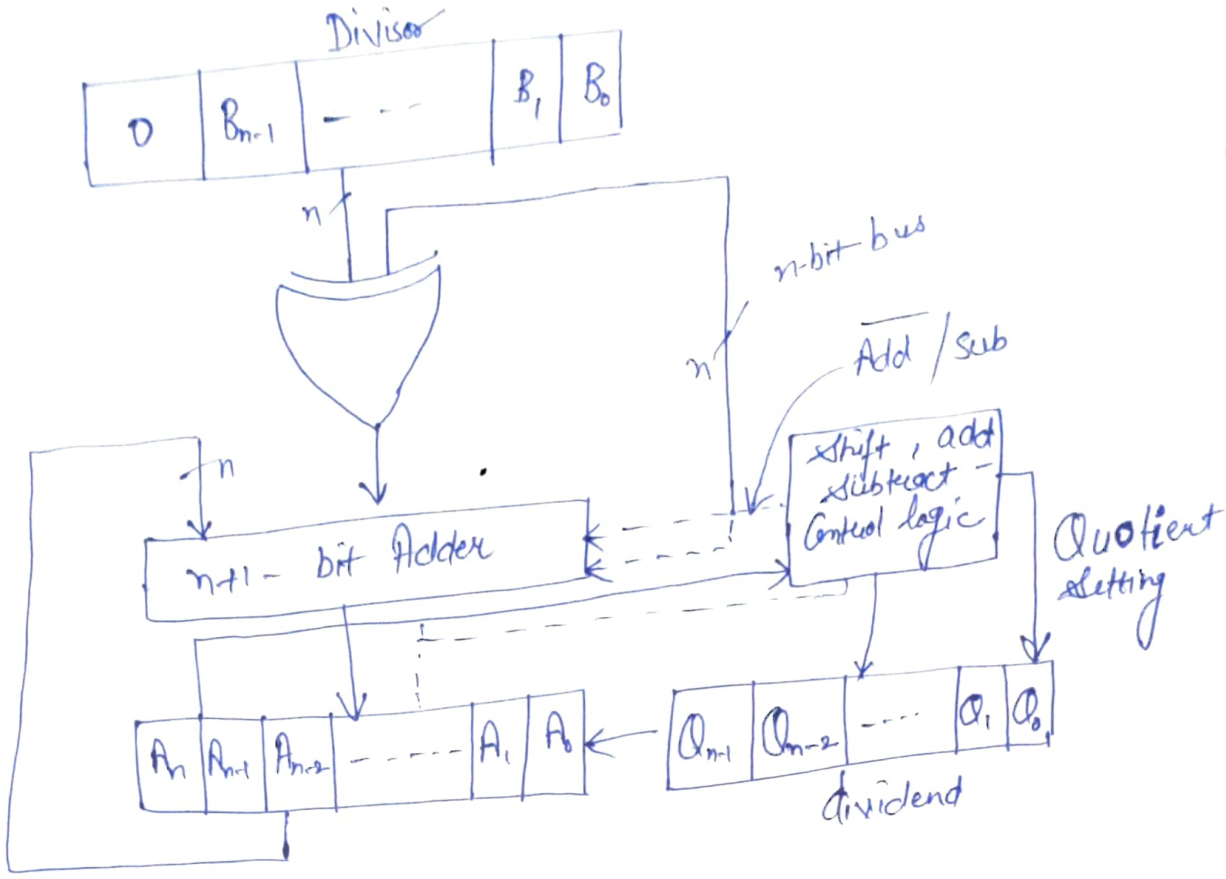
There are three Register A, B & Q

In Q Register, we are used to store dividend

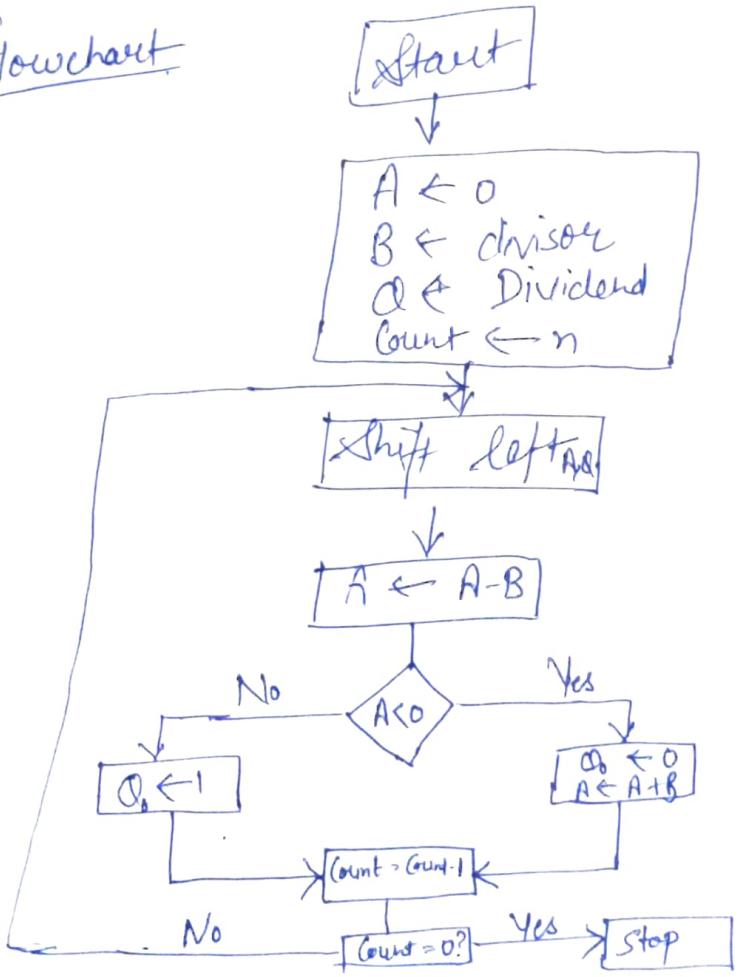
In B Register, ——— ——— divisor

In A Register, ——— ——— Remainder

Remainder is in A register & Quotient i.



Flowchart



00000

Remainder

Quotient.

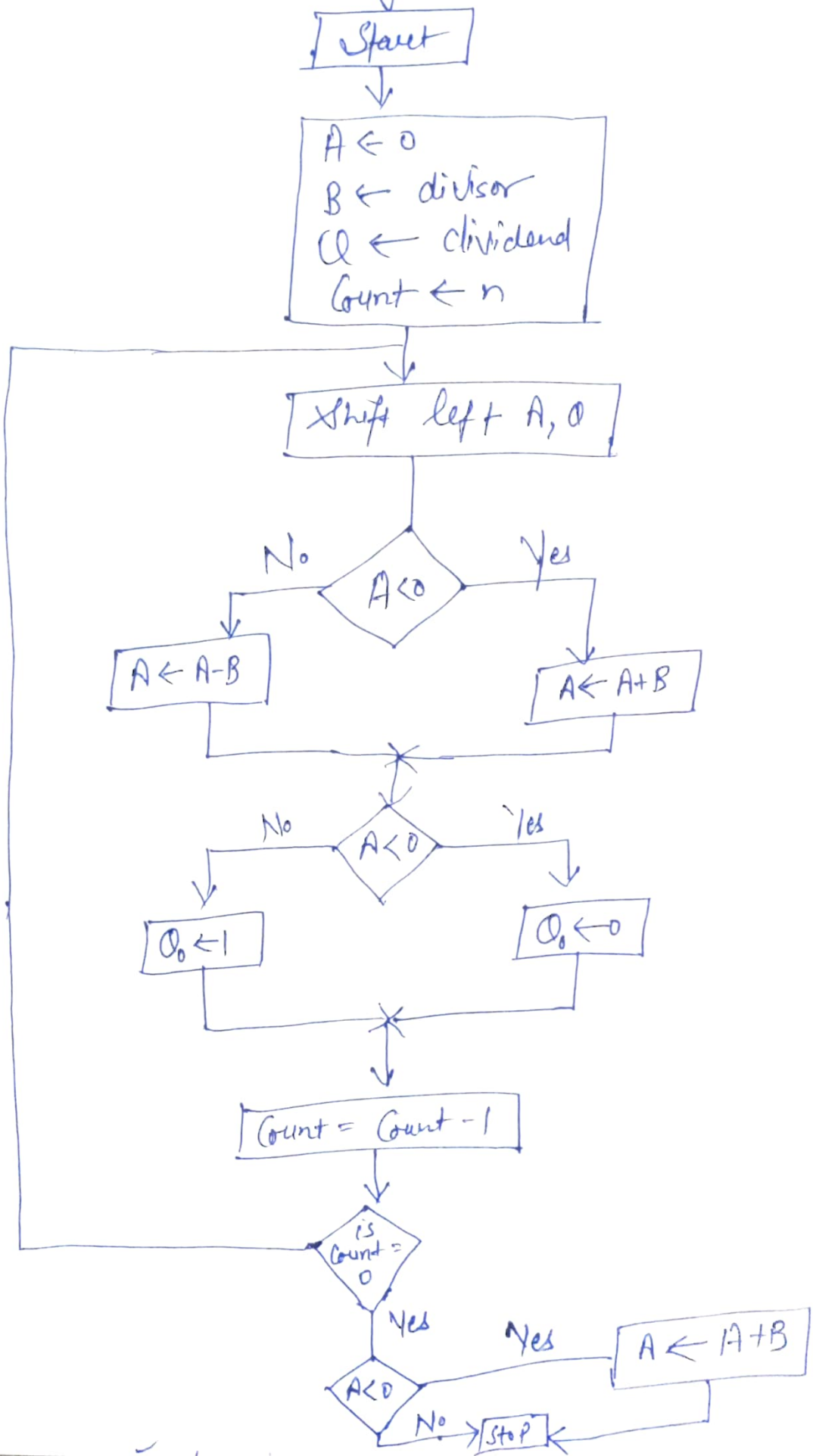
Dividend = 1010 = Q

Divisor = 0011 = B

$-B = \bar{B}+1 = 11101$

operation	A	Q	
Initialize	00000	1010	
Shift left	00001	010□	
A − B	00001 + 11101 = 11110		
	i.e negative Value. so	$Q_0 = 0$	1st
Restore A+B	00001 + 00001 = 00010	0100	
Shift left	00100	100□	
A←A−B	00100 + 11101 = 00001		
		1001	2nd
Restore A+B	00001 + 00011 = 00100		
Shift left	01000	000□	
A−B	01000 + 11101 = 00101		
		0001	3rd
Shift left	01010	001□	
A−B	01010 + 11101 = 00111		
		0011	4th
	Remainder	Quotient.	

Non-Restoring Division Method



Example (Non-Restoring)

Dividend = 1011 → Q

Divisor = 0101 = B − 00101

B + 1 = 11011

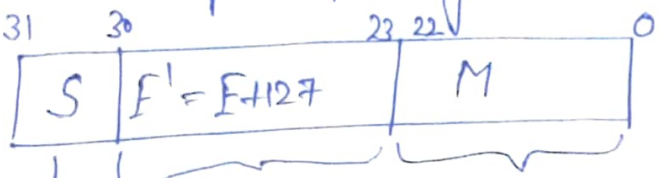
Operation	A	Q	
Initialize	00000	1011	1st
Shift	00001	011□	
A − B	+ 00001		
	11011		
	11100	011[0]	
Shift	11000	110□	2nd
A + B	+ 01000		
	00101		
	11101	110[0]	
Shift	11011	100□	3rd
A + B	11011		
	00101		
	⊗ 00000	100[1]	
Shift	00001	001□	4th
A − B	+ 00001		
	11011		
	11100	001[0]	
	A	Q	

Now Count is 0 & A<0 A is negative.

A + B	11100	
	00101	
	⊗ 00001	0010
	Remainder	Quotient

IEEE Standard for floating-point Numbers

a) Single Precision



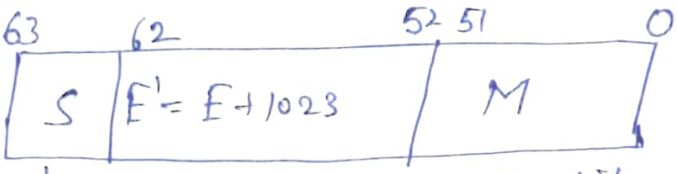
32-bit

Sign of Number
0 signifies +
1 signifies −

8 bit signed exponent in excess -127 representation

23 bit Mantissa fraction

b) Double Precision



64-bit

Sign

11-bit excess 1023 - exponent

52-bit Mantissa fraction.

e.g $(28.125)_{10}$

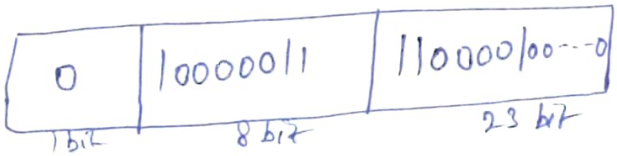
11100.001

1.1100001×2^4

M

$E' = E + 127 = 4 + 127 = 131 = 10000011$

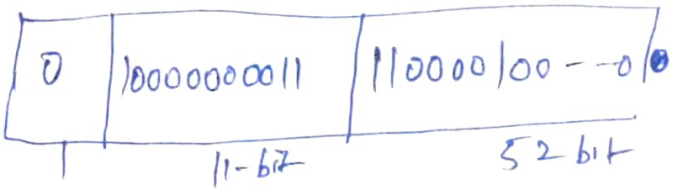
$S = 0$



Single precision.

In Double $E' = E + 1023$

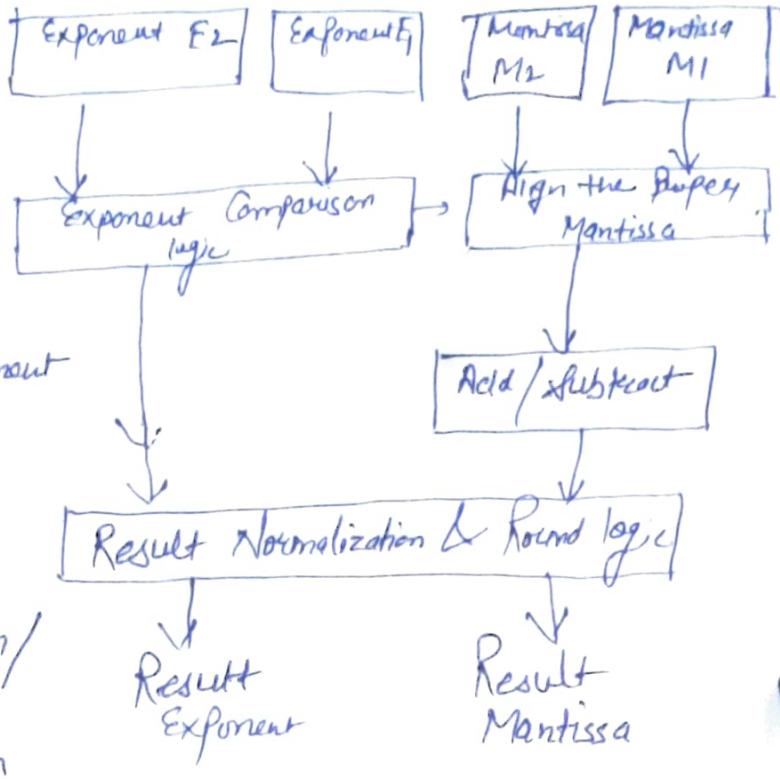
$4 + 1023 = 1027 = 10000000011$



Floating Point Arithmetic : Add & Subtract

Steps to add/subtract two floating-point Numbers:-

1. Compare the magnitudes of the two exponents & make suitable alignment to the No. with the smaller magnitude of exponent
2. Perform the addition/subtraction.
3. Perform Normalization by shifting the resulting mantissa & adjusting the resulting exponent.



Example → Add 1.1100×2^4 & 1.100×2^2

Alignment:- 1.1000×2^2 has to be aligned to 0.0110×2^4

Addition → Add two No.