ALU

Topics to be covered

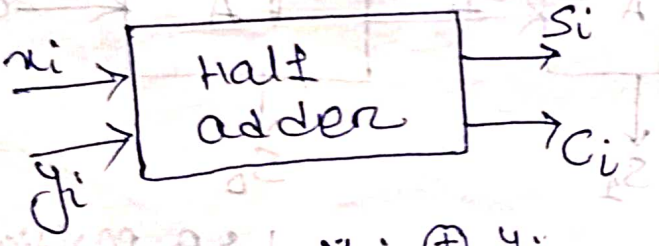
* Design of Adders (n-bit ripple carry adder, carry look ahead adder).
* Multiplication of positive numbers.
* Signed operand Multiplication
* Fast Multiplication
* Integer Division (Restoring and non-restoring)
* IEEE floating-point Numbers and its operations (single and Double precision).

Design of Adders :-

N-bit Ripple Carry Adder :-

Some Foundation :-

Addition and subtraction of sign numbers :-

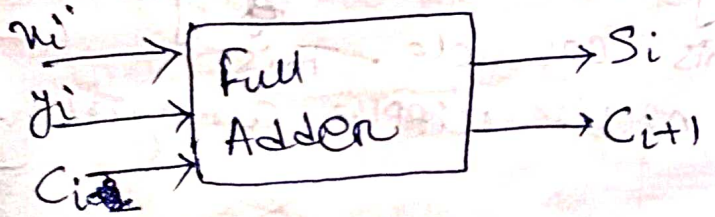


$S_i = x_i \oplus y_i$

$C_i = x_i y_i$

Truth table of Half adder

x_i	y_i	S_i	C_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



$S_i = \bar{x_i}\,\bar{y_i}\,C_i + x_i\,y_i\,\bar{C_i} + x_i\,\bar{y_i}\,\bar{C_i} + x_i\,y_i\,C_i$

$= x_i \oplus y_i \oplus C_i$

$C_{i+1} = \bar{x_i}\,y_i\,C_i + x_i\,\bar{y_i}\,C_i + x_i\,y_i\,\bar{C_i} + x_i\,y_i\,C_i$

Truth table of full adder

x_i	y_i	C_i	S_i	C_{i+1}
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

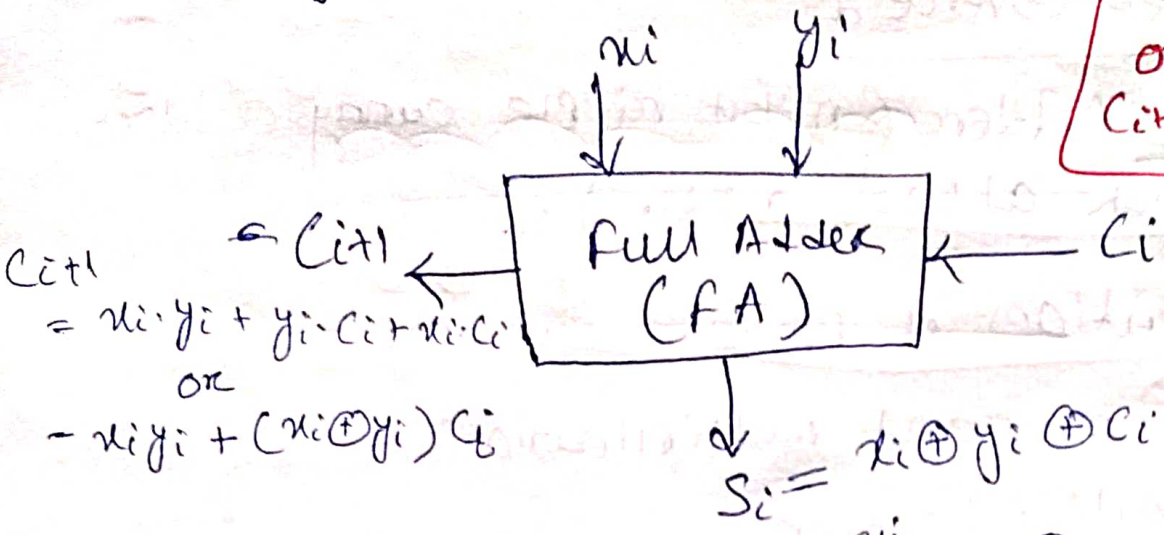
Block diagram of full adder

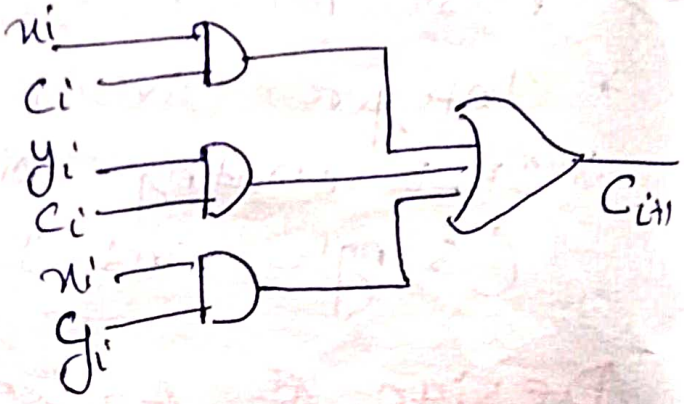
$S_i = x_i \oplus y_i \oplus c_i$

$C_{i+1} = x_i y_i + y_i c_i + x_i c_i$

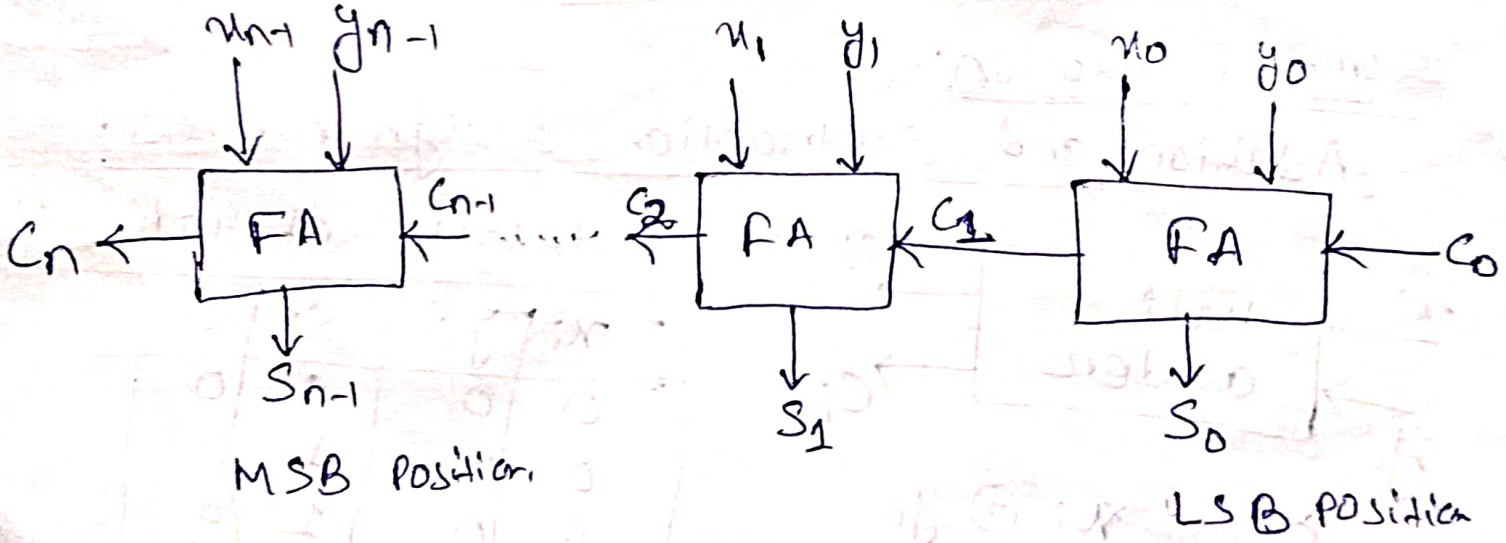
or

$C_{i+1} = x_i y_i + (x_i \oplus y_i) c_i$





N-bit ripple carry adder :-



→ To add two n bit numbers 'n' full adders are required. The carry must be propagate or ripple through this cascade. This configuration is called n-bit ripple carry adder.

flip a number

1 to 0
0 to 1

$1 \oplus 1 = 0$
$0 \oplus 1 = 1$

$y_i \oplus 0 = y_i$
$1 \oplus 0 = 1$
$0 \oplus 0 = 0$

$y_i \rightarrow \overline{y_i}$

$$y_i \oplus 1 = \overline{y_i}$$

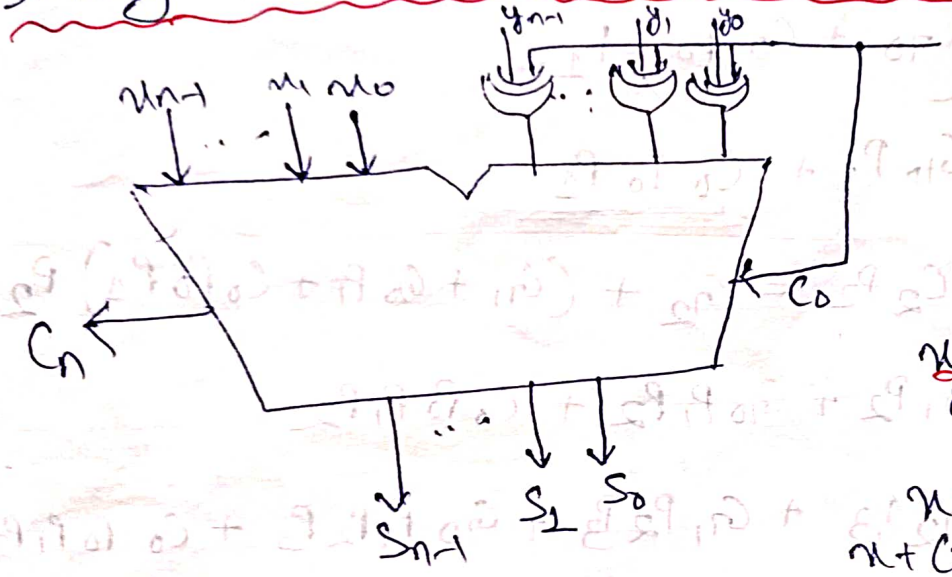
$$y_i \oplus 0 = y_i$$

$y_i \rightarrow -y_i$

$= \overline{y_i} + 1$

1's complement + 1 = 2's complement.

Binary addition - subtraction logic network



$x+y \rightarrow$ Add/Sub control = 0

$x - y \rightarrow$ Add/Sub Control = 1

$x + (-y)$

$x + (2$'s complement of $y)$

Carry Look ahead Adder :-

→ Demerit of n-bit ripple carry adder :-

* take too much time to produce S_0 through S_{n-1} and C_n.
* carry & sum depends on the previous stage.
* This leads to a time delay in addition process
* This delay is known as propagation delay.

$S_i = x_i \oplus y_i \oplus c_i$

$C_{i+1} = x_i y_i + x_i c_i + y_i c_i$ or

$C_{i+1} = \underbrace{x_i y_i}_{G_i} + \underbrace{c_i (x_i \oplus y_i)}_{P_i}$

G_i ↓ Carry generation

P_i ↓ Carry Propagate

$G_i = x_i y_i$, $P_i = x_i \oplus y_i$

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

$C_1 = G_0 + C_0 P_0$: $G_0 = x_0 \cdot y_0$, $P_0 = x_0 \oplus y_0$

$C_2 = G_1 + C_1 P_1$: $G_1 = x_1 y_1$, $P_1 = x_1 \oplus y_1$

$= x_1 \cdot y_1 + \left(x_0 \cdot y_0 + C_0 (x_0 \oplus y_0)\right)(x_1 \oplus y_1)$

$= G_1 + \left(G_0 + C_0 P_0\right) \cdot P_1$

$= G_1 + G_0 P_1 + C_0 P_0 P_1$

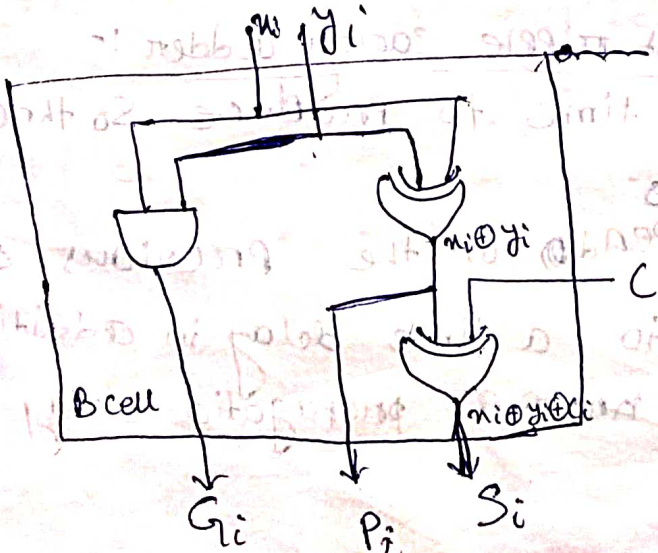
$C_3 = G_2 + C_2 P_2 = G_2 + (G_1 + G_0 P_1 + C_0 P_0 P_1) P_2$

$= G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2$

$C_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3$

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

$= G_i + P_i G_{i-1} + P_i P_{i-1} C_{i-1}$

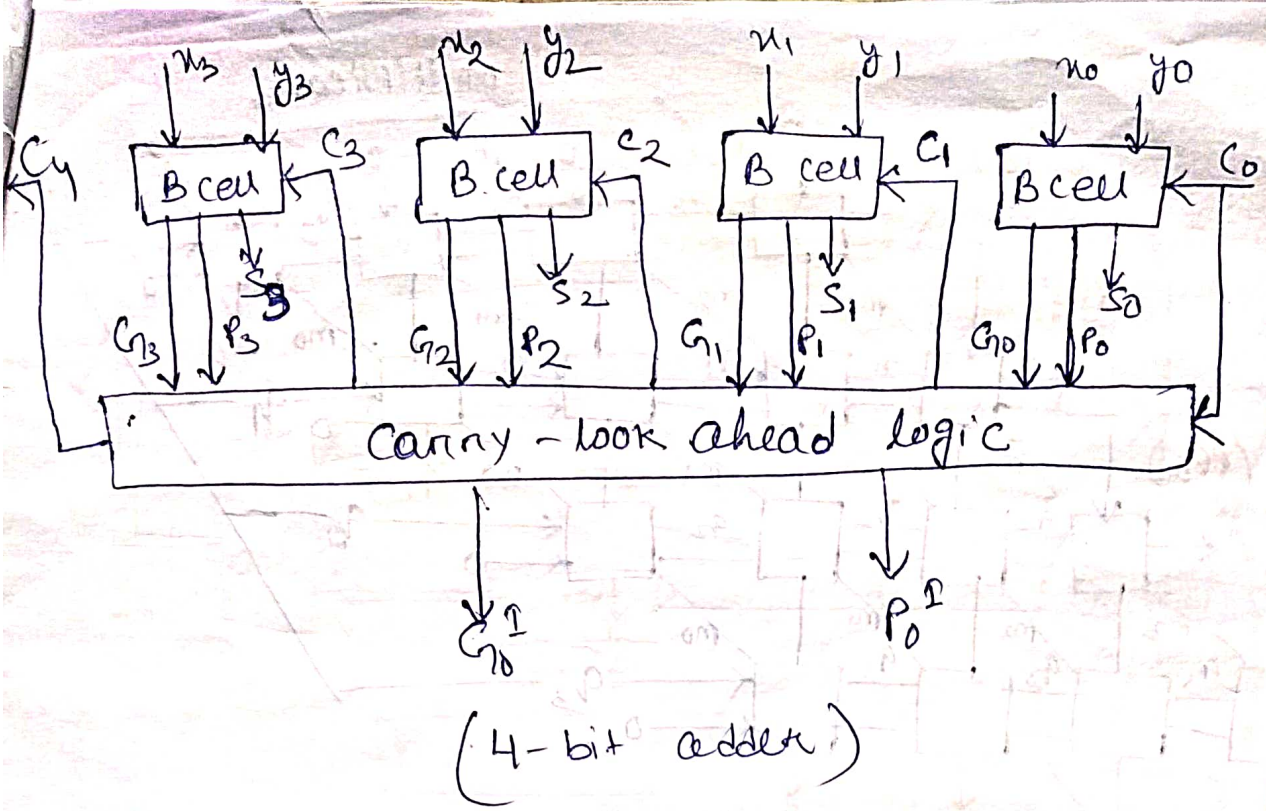


(Bit stage cell)

$S_i = x_i \oplus y_i \oplus C_i$

$G_i = x_i y_i$

$P_i = x_i \oplus y_i$



(4-bit adder)

Generate and Propagate function :-

$$P_0^I = P_3 P_2 P_1 P_0$$

$$G_0^I = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

Multiplication of Positive Numbers :—

→ Multiplication of Positive Numbers.
→ Multiplication of signed numbers.

$$\begin{array}{r} 1\ 1\ 0\ 1 \rightarrow \text{Multiplicand } M = 13 \\ 1\ 0\ 1\ 1 \rightarrow \text{Multiplier } Q \\ \hline 1\ 1\ 0\ 1 \rightarrow Q_0 M \\ 1\ 1\ 0\ 1\ X \rightarrow Q_1 M \times 2^1 \\ 1\ 0\ 0\ 0\ X\ X \rightarrow Q_2 M \times 2^2 \\ 1\ 1\ 0\ 1\ X\ X\ X \rightarrow Q_3 M \times 2^3 \\ \hline 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \end{array} \quad = 11$$

$$\begin{array}{r} 13 \\ 11 \\ \hline 13 \\ 13 \\ \hline 143 \end{array}$$

$$128 + 8 + 4 + 2 + 1$$
$$= 128 + 15 = 143$$

→ Multiply by 2 will give left shift of No.
Ex: $4 \times 2 = 8$ $\quad 4 = 100$ $\quad 8 = 1000$ $\quad 100 \xrightarrow{*2} 1000$

4 bit CLA

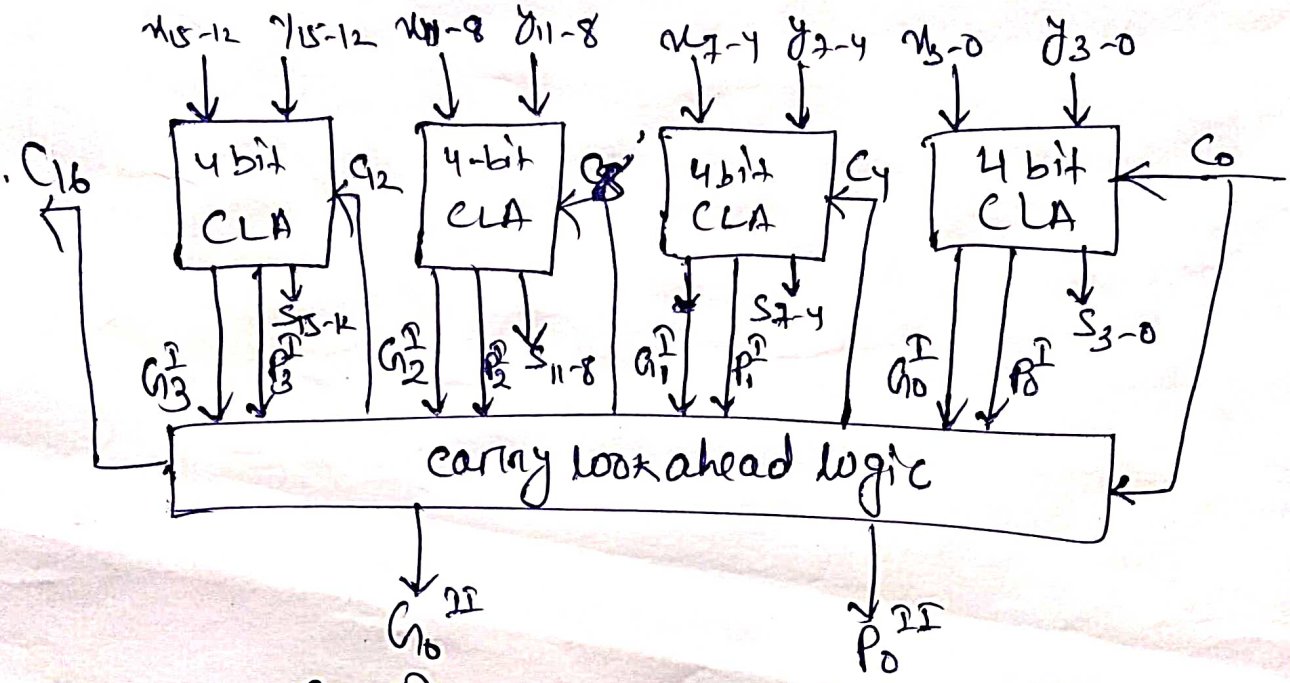
All G_i and P_i will be generated after 1 CC

Then at 3 CC all carry will be generated (C_1, C_2, C_3, C_4)

Then at 4 CC all sum will be generated. (S_0, S_1, S_2, S_3)

final result = $C_4\ S_3\ S_2\ S_1\ S_0$

16-bit CLA :-



Time $\rightarrow (1+2)$

at 3 CC C_4 will be generated. (C_1, C_2, C_3, C_4)

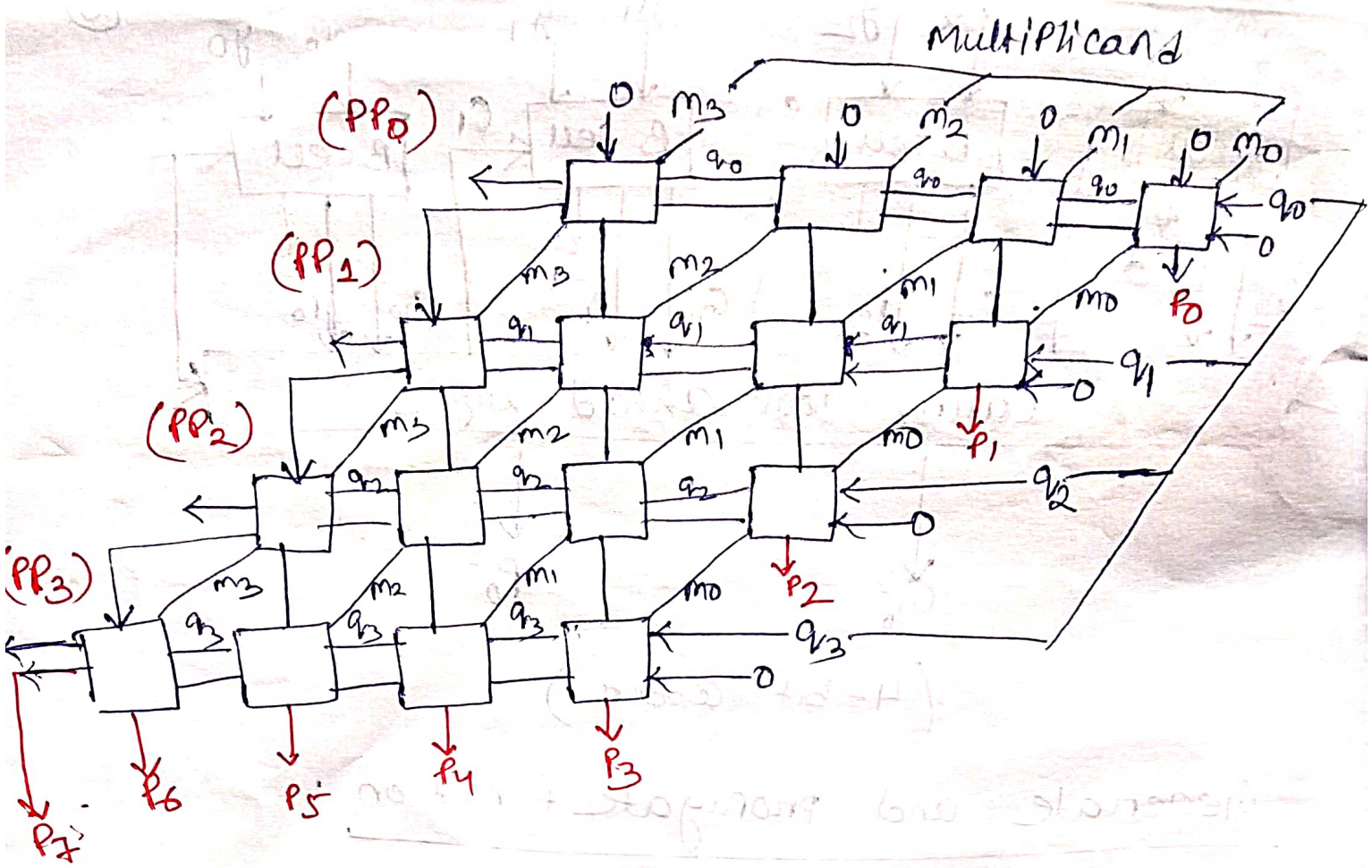
at 5 CC $\rightarrow (3+2)$, C_8 will be generated (C_5, C_6, C_7, C_8)

at 7 CC $\rightarrow (5+2)$, C_{12} will be generated $(C_9, C_{10}, C_{11}, C_{12})$

at 9 CC $\rightarrow (7+2)$, C_{16} will be generated. $(C_{13}, C_{14}, C_{15}, C_{16})$

at 10 CC, all sum be generated $(S_{12}, S_{13}, S_{14}, S_{15})$

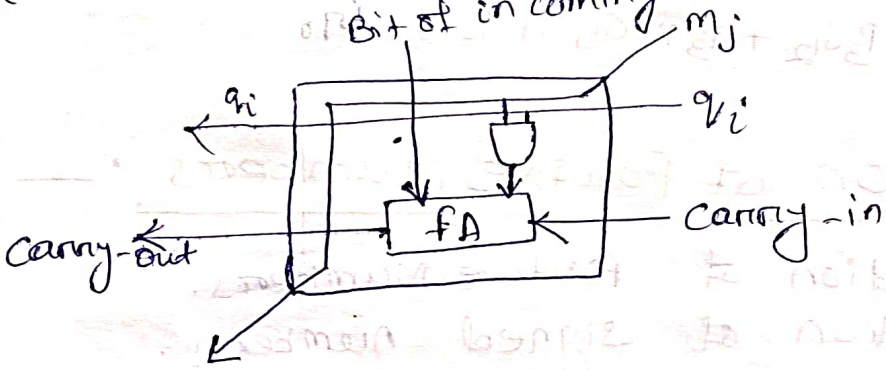
For 32 bit → Time ??



$PP_4 = P_7\ P_6\ P_5\ P_4 \cdots P_0$

PP → Partial Product.

Bit of in coming partial Product (PP_i)



$$PP_{i+1} = PP_i + q_i 2^i M$$

$$\begin{array}{r} 1\ 1\ 0\ 1 \leftarrow M \\ \underline{1\ 0\ 1\ 1} \leftarrow Q \\ \underset{q_3\ q_2\ q_1\ q_0}{} \\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \rightarrow PP_0 = 0 \end{array}$$

$PP_1 = PP_0 + q_0 2^0 M$
$= PP_0 + q_0 \cdot M$

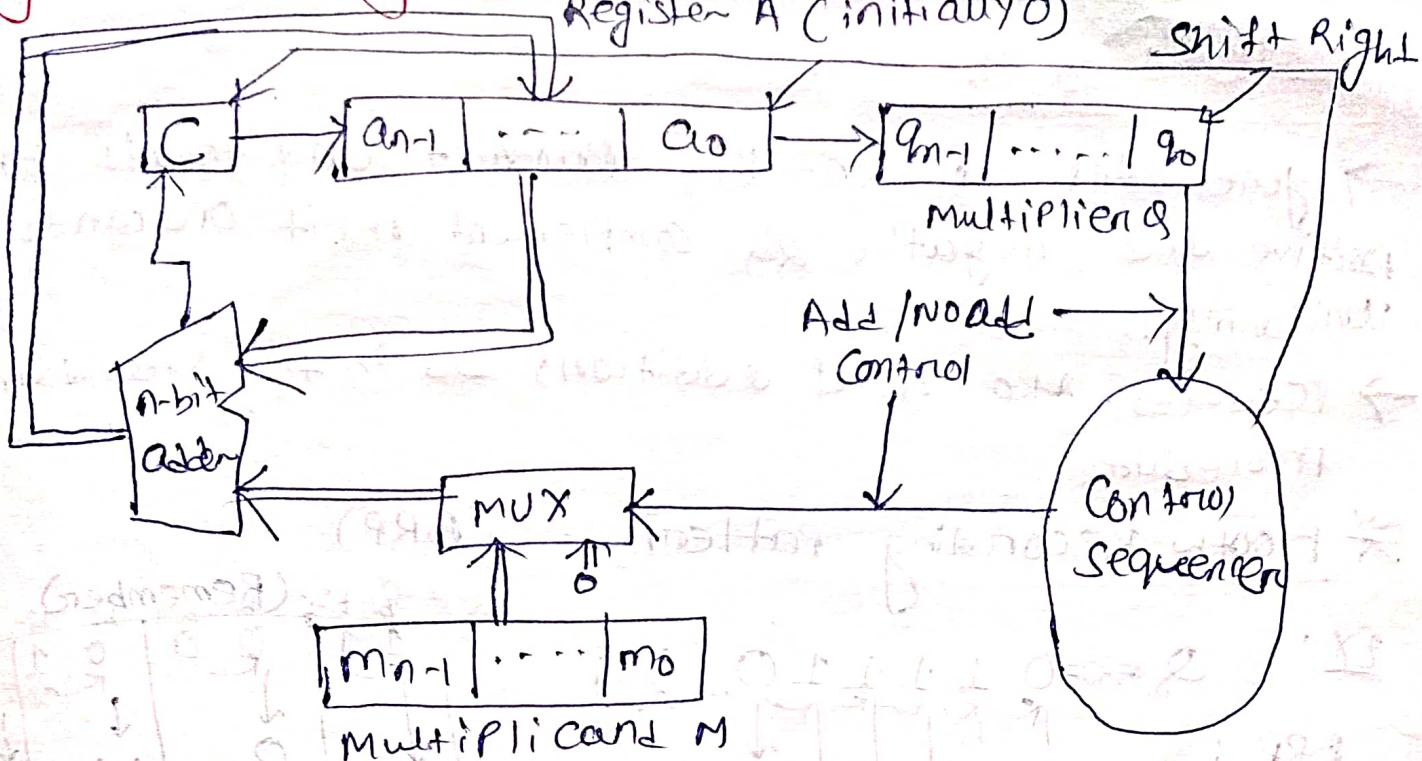
$$\begin{array}{r} 1\ 1\ 0\ 1\ (q_0 M) \\ \underline{0\ 0\ 0\ 0\ 1\ 1\ 0\ 1} \rightarrow PP_1 \\ 1\ 1\ 0\ 1\ 0 \rightarrow 2 \cdot q_1 \cdot M \\ \underline{0\ 0\ 1\ 0\ 0\ 1\ 1\ 1} \rightarrow PP_2 \\ 0\ 0\ 0\ 0\ 0\ 0 \rightarrow 2^2 \cdot q_2 \cdot M \\ \underline{0\ 0\ 1\ 0\ 0\ 1\ 1\ 1} \rightarrow PP_3 \\ 1\ 1\ 0\ 1\ 0\ 0\ 0 \rightarrow 2^3 \cdot q_3 \cdot M \\ \underline{1\ 0\ 0\ 0\ 1\ 1\ 1\ 1} \rightarrow PP_4 \end{array}$$

$PP_2 = PP_1 + q_1 2^1 \cdot M$
$= PP_1 + q_1 2 \cdot M$

$PP_3 = PP_2 + q_2 2^2 M$

$PP_4 = PP_3 + q_3 \cdot 2^3 M$

Register configuration of Multiplication Unit.



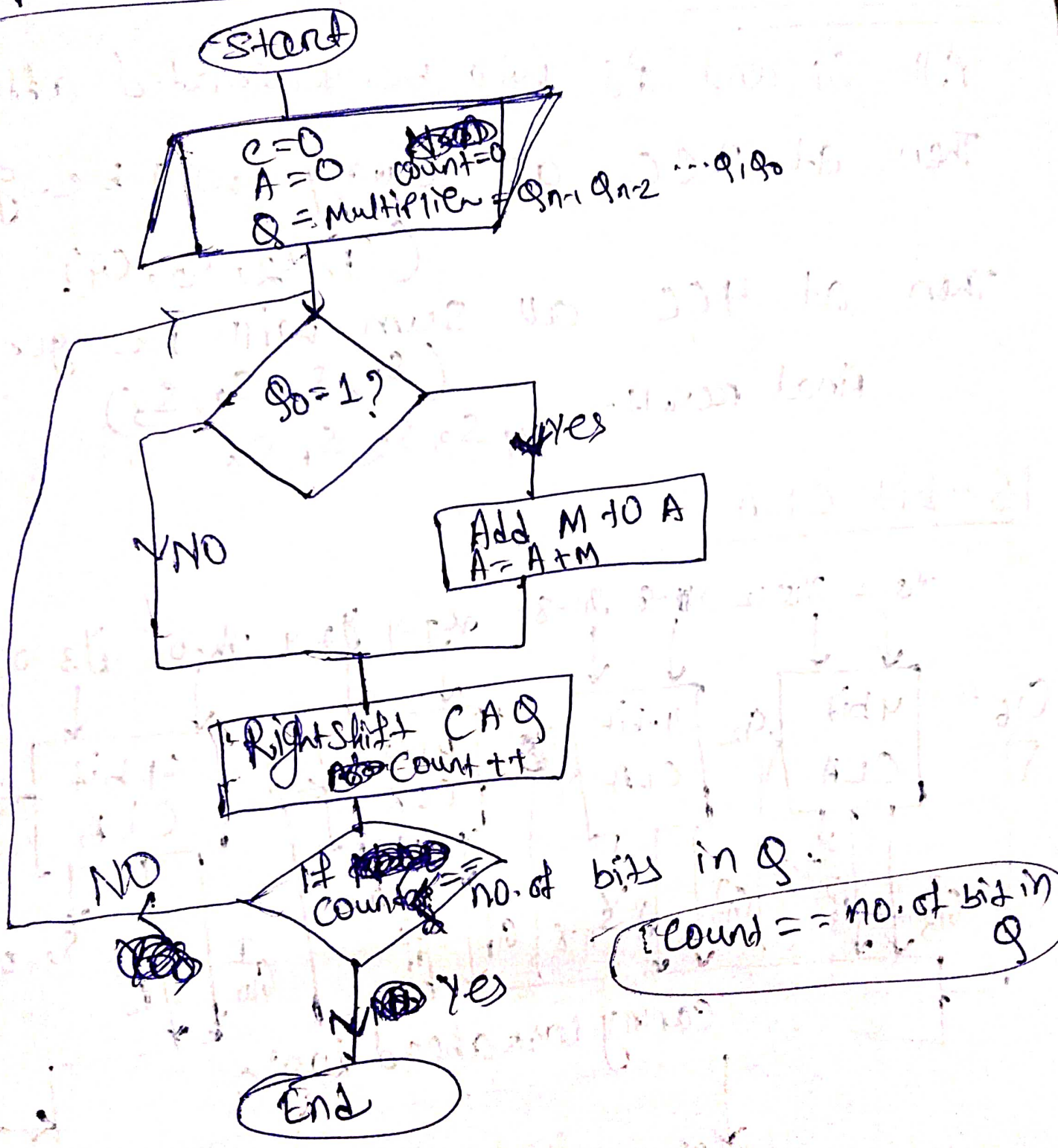
* Multiply two numbers M = 1101 , Q = 1011

	C	A	Q	
		0000	1011	
q_0 = 1, Add M	0	1101	1011	1st cycle
Right shift CAQ	0	0110	1101	
q_0 = 1, Add M	1	0011	1101	2nd cycle
Right shift CAQ	0	1001	1110	
q_0 = 0, No Add M	0	1001	1110	3rd cycle
Right shift CAQ	0	0100	1111	
q_0 = 1, Add M	1	0001	1111	4th cycle
Right shift CAQ	0	1000	1111	

Product

Product = 10001111 .

Flow chart of multiplication of the no.



Signed Operand Multiplication :-

Booth Algorithm :-

→ generates a 2n-bit Product and treats both positive and negative 2's complement n-bit operands uniformly.

→ reduces the no. of additions as in the standard procedure.

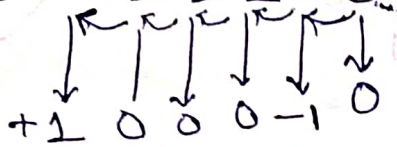
* Booth Recording Pattern :- (BRP)

** b_i b_{i-1} (Remember)

b_i b_{i-1}	1 1	0 0	0 1	1 0
	0	0	+1	-1

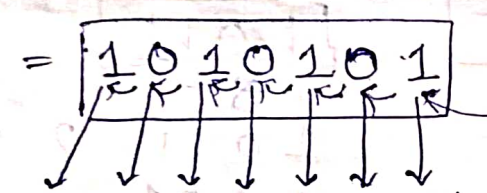
Ex : Q = 30 = 0 1 1 1 1 0 ... 0

BRP :



Booth recording Pattern is : +1 000 -1 0

Ex : Q = 1 0 1 0 1 0 1 : 0



BRP : -1 +1 -1 +1 -1 +1 -1

Q. Multiply 2 nos -9 x -13.

M = -9

Q = -13

+9 = 01001

-9 = 2's complement of 9 = 1's complement + 1

1's complement of 9 =

+9 = 0 1 0 0 1

1 0 1 1 0 → 1's complement.

+ 1

1 0 1 1 1 → 2's complement.

-9 = 10111

M = -9 = 10111

Q = -13 =

+13 = 01101 → 1's complement
10010
+1
-13 → 10011 2's complement

Q = 10011

M = -9 ⇒ -M = 9

Subtract M from X ⇒ X - M

Otherwise we can tell Add (-M) to X

X - M = X + (-M)

M = -9 = 10111
-M = +9 = 01001
Q = -13 = 10011

A - M = A + (-M)
= A + 9

Now multiplication Procedure:-

	A	Q	E
Initialization	00000	10011	0
Sub M (Add -M)	+ 01001		
	01001	10011	0
AShift R (Arithmatic Shift Right)	00100	11001	1
NO Add/Sub		0 → NO operation	
AShift R	00010	01100	1
Add M	+ 10111	+1 → Add M	
	11001	01100	1
AShift R	11100	10110	0
NO Add/Sub		0 → NO operation	
AShift R	11110	01011	0
Sub M (Add -M)		-1 → Sub M	
	+ 01001		
	00111	01011	0
AShift R	00011	10101	1

1st cycle, 2nd cycle, 3rd cycle, 4th cycle, 5th cycle

AQ = Product Result

product = 00011 10101 = 117 = (-13 * -9)
A Q

Q. Multiply $-15 * -9$ using booth's Algorithm

$M = -15 = 10001$

$Q = -9 = 10111$

$-M = +15 = 01111$

$+15 \rightarrow 01111$

$-15 \rightarrow 10000 + 1 = 10001$

$+9 \rightarrow 01001$

$-9 \rightarrow 10110 + 1 = 10111$

Operation	A	Q	C	
Initialization:	00000	10111	0	
Sub M (Add -M)	+01111		-1 → sub M	
	01111	10111	0	
AShift R	00111	11011	1	1st cycle
NO Add/Sub AShift R	00011	11101	1	2nd cycle
NO Add/Sub AShift R	00001	11110	1	3rd cycle
Add M	+10001		+1 → Add M	
	10010	11110	1	
AShift R	11001	01111	0	4th cycle
Sub M (Add -M)	+01111		-1 → sub M	
	Discard 01000	01111	0	
AShift R	00100	00111	1	5th cycle

$AQ = 0010000111 = 128 + 4 + 2 + 1$

128 64 32 16 8 4 2 1

$= 135$

Which is equal to $-15 * -9$

* When to stop

the no. of cycles = no. of bits present in Q.

means. the MSB of Q is shifted to Sign position.

Q. Multiply -15×-3 using booth Algorithm

$M = -15 = 10001$

$-M = +15 = 01111$

$Q = -3 = 101$

$+3 \rightarrow 011$

$-3 \rightarrow 100$

$+1$

101

$-4 + 1 = -3$

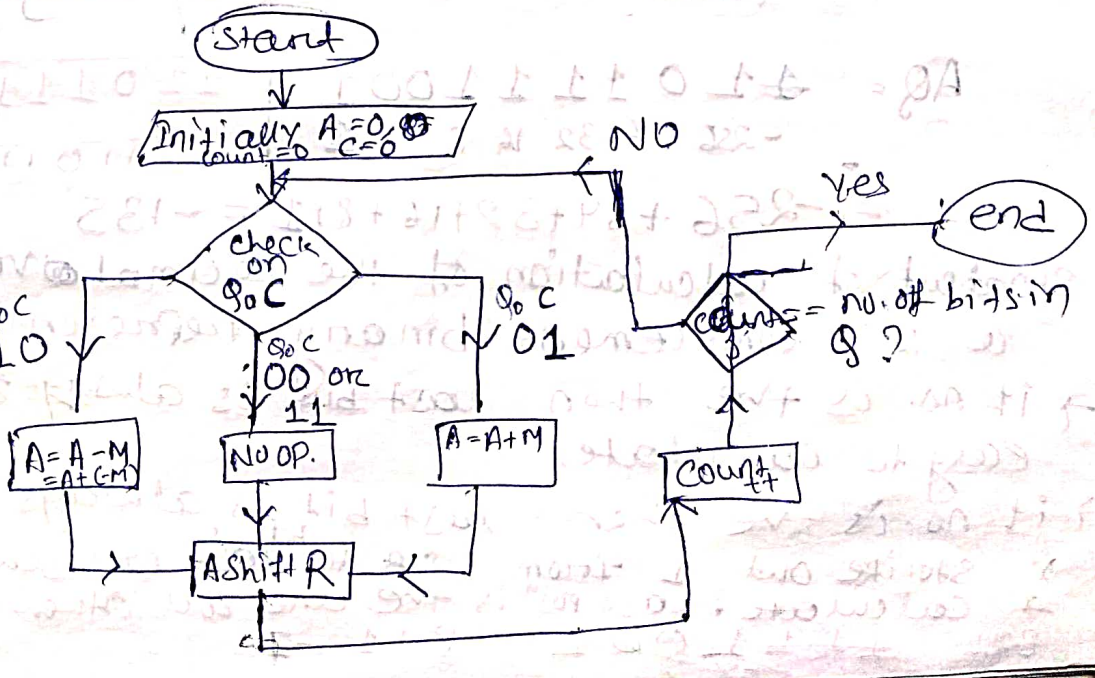
Operation	A	Q	C	
Initialization	00000	101	0	
	+01111		-1	
Sub M	01111	101	0	} cycle 1
Ashift+R	00111	110	1	
	+10001		+1	
Add M	11000	110	1	} cycle 2
Ashift+R	11100	011	0	
	+01111		-1	
Sub M	01011	011	0	} cycle 3
Ashift+R	00101	101	1	

00101101

$32 + 8 + 4 + 1 = 45 = -15 \times -3$

$= 45$

Flow Chart of Booth Algorithm for Multiplication



Q. Multiply $+15 \times -9$ using Booth's Algorithm

$M = +15 = 01111$

$-M = -15 = 10001$

$Q = -9 = 10111$

$+15 = 01111$

1's comp $= 10000$

$+1$

$-15 = 10001$

$+9 = 01001$

1's comp $= 10110$

$+1$

$-9 = 10111$ *

Operation	A	Q	C	
Initial	00000	10111	0	
Sub M, A + (M)	+10001			
	10001		-1	
A shift R	11000	11011	1	1st cycle
A shift R	11100	01101	1	2nd cycle
A shift R	11110	00110	1	3rd cycle
Add M, A + M	+01111		+1	
	01101	00110	1	
A shift R	00110	10011	0	4th cycle
Sub M, A + (-M)	+10001		-1	
	11011	10011	0	
A shift R	11011	11001	1	5th cycle

$AQ = 1\,1\,0\,1\,1\,1\,1\,0\,0\,1$

$-256\ \ 64\ 32\ 16\ 8\ \ 1$

$= -256 + 64 + 32 + 16 + 8 + 1 = -135$

1's: 1101111001 − 1 = 1101111000; 2's: 0010000111

Shortcut of calculation of the decimal value of a 2's complement binary representation

→ if no. is +ve then last bit is always zero and easy to calculate

→ if no. is -ve then last bit is always one

* Strike out 1 from LSB to before last consecutive 1's.

→ Calculate. Last pos is -ve and all other +ve (after that)

ex: 1111001 = −8 + 1 = −7

→ The Booth algorithm has two attractive features.
* First it handles both positive and negative multipliers uniformly.
* Second It achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1's.

→ The speed gained by skipping over 1s depends on the data.

→ On average, the speed of doing multiplication with the Booth Algorithm is same as with the normal algorithm.

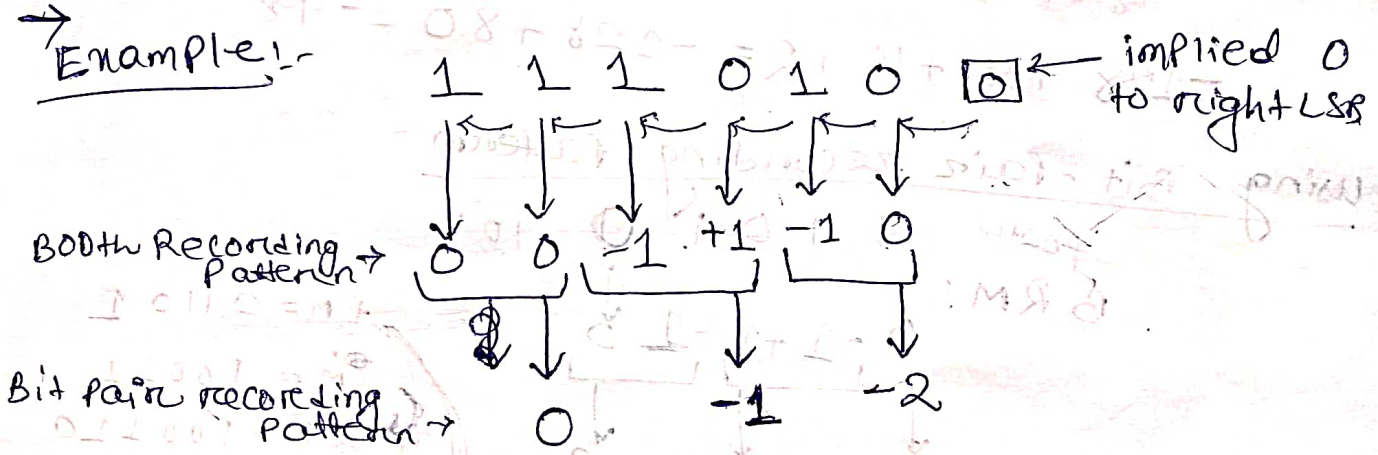
Fast Multiplication :-

→ To speed up the multiplication operation 2 techniques are used.

* Bit Pair recording of multipliers (maximum no. of summands that must be added is n/2 for n-bit operands).
* Carry save addition of summands (reduces the time needed to add the summands).

*Bit Pair recording of multipliers :-

→ Example :-



Multiplier bit-pair		Multiplier bit on right	Multiplicand selected at position i
i+1	i	i-1	
0	0	0	0 × M
0	0	1	+1 × M
0	1	0	+1 × M
0	1	1	+2 × M
1	0	0	-2 × M
1	0	1	-1 × M
1	1	0	-1 × M
1	1	1	0 × M

→ In this technique, summands is n/2 for n bits operation.

→ It halves the maximum no. of summands.

Example:-

+13 → 0 1 1 0 1 (M) −M = −13 = 10011

−6 → 1 1 0 1 0 (Q)

Using Booth's Algorithm:-

Booth Recording pattern

1 1 0 1 0 0

0 −1 +1 −1 0

0 −1 +1 −1 0

0 1 1 0 1

(X) 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 0 1 1 0 0 1 0 = −48

128 64 32 16 8 4 2 1

−128 +32 +16 +2 = −128 + 80 = −48

Using Bit-Pair recording pattern

Booth 1 1 0 1 0 0

BRM:

0 −1 +1 −1 0

0 −1 −2

+M = 0 1 1 0 1

−M = 1 0 0 1 1

−2M = 1 0 0 1 1 0

+2M = 0 1 1 0 1 0

left shift

0 1 1 0 1

(X) 0 −1 −2

1 1 1 1 1 0 0 1 1 0 −2M

1 1 1 1 0 0 1 1 − − −M

0 0 0 0 0 0 − −

1 1 1 0 1 1 0 0 1 0 = −48

* Carry save addition of summands :-

→ Reduces the time needed to add summands.

→ Carry save addition technique speed up of the addition process.

→ In this technique carrys are saved and introduced in the next row at the correct position.

45 → 1 0 1 1 0 1 (M)

63 → 1 1 1 1 1 1 (Q)

```
      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
```

1 0 1 1 0 0 0 1 0 0 1 1 → Product.

= 2835

```
  1
  1
  1
  1
+ 1
-----
  100 ?

  1
  1
-----
 10
+10
-----
 100
```

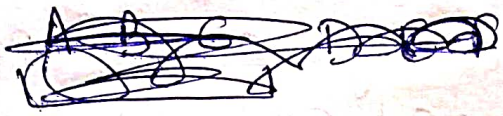
→ This multiplication can be performed using CSA.

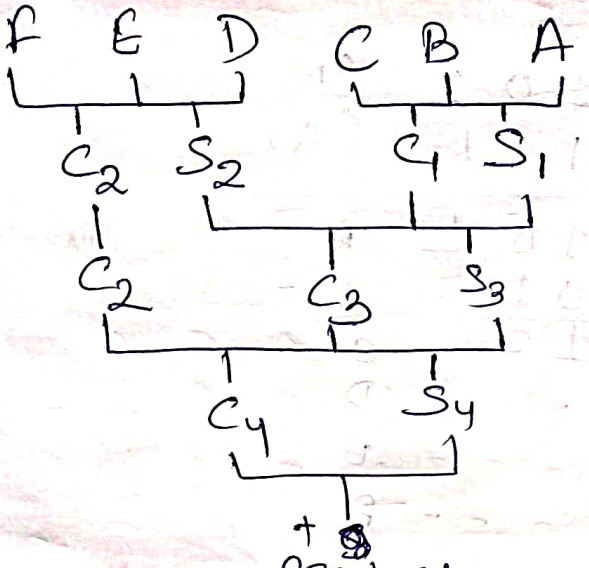
```
    1 0 1 1 0 1
    1 1 1 1 1 1
-------------------
    1 0 1 1 0 1   A
  1 0 1 1 0 1     B
1 0 1 1 0 1       C
-------------------
1 1 0 0 0 0 1 1   S1
0 0 1 1 1 1 0 0   C1
```

```
  1 0 1 1 0 1   D
1 0 1 1 0 1     E
1 0 1 1 0 1     F
---------------
1 1 0 0 0 0 1 1   S2
0 0 1 1 1 1 0 0   C2
```

```
                1 1 0 0 0 0 1 1   (S1)
              0 0 1 1 1 1 0 0     (C1)
            1 1 0 0 0 0 1 1       (S2)
          1 1 0 1 0 1 0 0 0 1 1   S3
        0 0 0 0 1 0 1 1 0 0 0     C3
      0 0 1 1 1 1 0 0             C2
    0 1 0 1 1 0 1 0 0 0 1 1       S4
  0 0 1 0 1 0 1 0 0 0 0 0         C4
```

$$
\begin{array}{r}
0\,1\,0\,1\,1\,1\,0\,1\,0\,0\,1\,1 \quad (S_4) \\
+\,0\,1\,0\,1\,0\,1\,0\,0\,0\,0\,0 \quad (C_4) \\
\hline
1\,0\,1\,1\,0\,0\,0\,1\,0\,0\,1\,1 \longrightarrow \text{Product}
\end{array}
$$





→ After 1 gate delay, all multiplier bits are available as input to CSA level.
Assume each CSA level takes 2 gate delay

∴ C_4 and S_4 are available after 6 gate delay

∴ Now $C_4 + S_4$ using carry-look-ahead adder

S_4 = 12 bits (0 to 11) → 3-4 bit adder.

All C_i & P_i → After 1 gate delay

Then (C_1, C_2, C_3, C_4 takes 2 gate delay) → After 1+2 = 3 gate delay } 1st 4-bit adder
(S_0, S_1, S_2, S_3) → After 4 gate delay. }

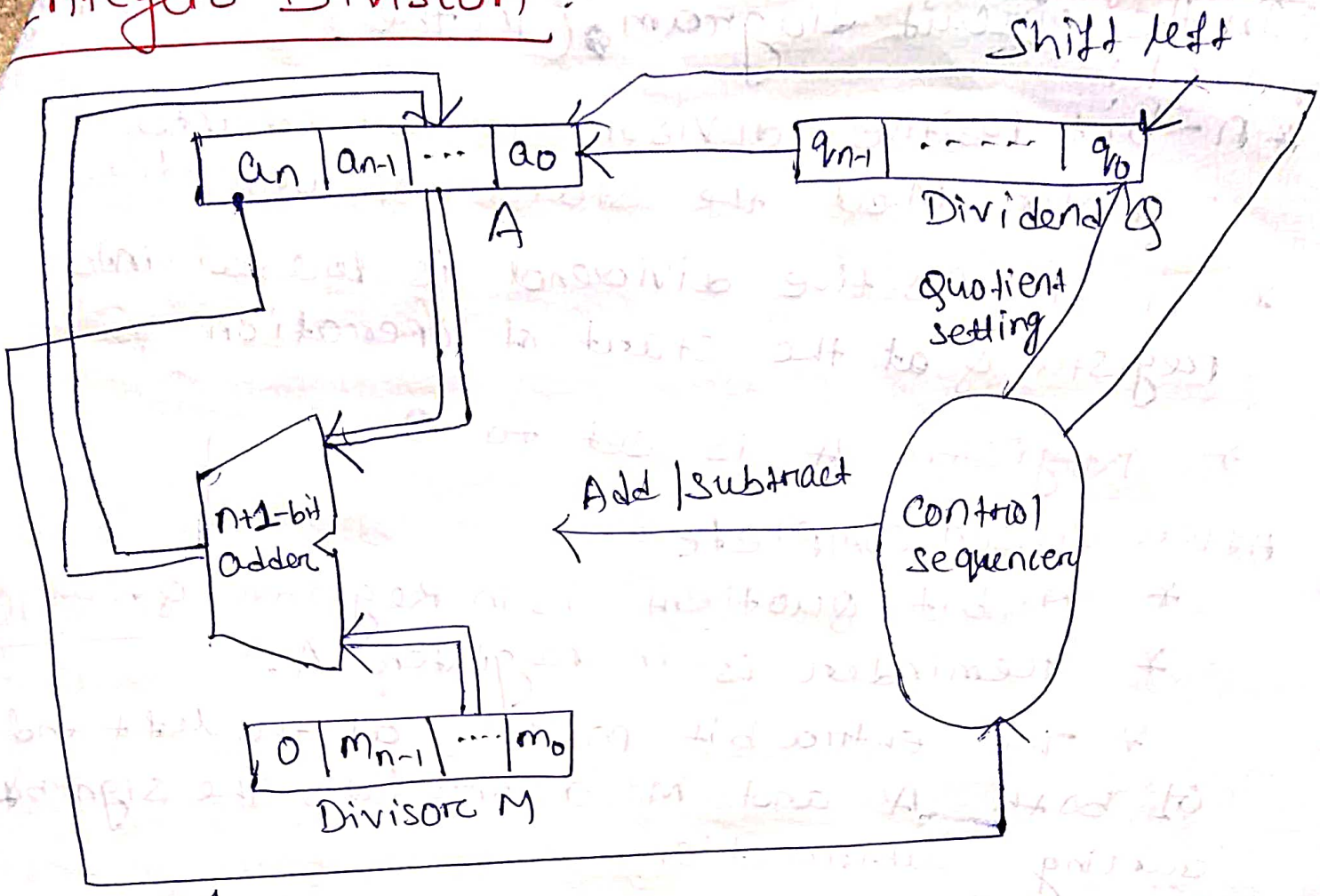
Then C_5, C_6, C_7, C_8 → After 3+2 = 5 gate delay } 2nd 4-bit adder
S_4, S_5, S_6, S_7 → After 6 gate delay

Then $C_9, C_{10}, C_{11}, C_{12}$ → After (5+2) = 7 gate delay } 3rd 4-bit adder
S_8, S_9, S_{10}, S_{11} → After 8 gate delay

∴ After 8 gate delay $C_4 + S_4$ = Solution will be available.

∴ Total = 1 + 6 + 8 = 15 gate delay for multiplication

Integer Division :-



$$\begin{array}{r} 21 \\ 13\overline{)274} \\ -26 \\ \hline 14 \\ -13 \\ \hline 1 \end{array}$$

$274 \rightarrow 100010010$

$13 \rightarrow 1101$

$16+4+1 = 21$

$$\begin{array}{r} 10101 \\ 1101\overline{)1000100\,10} \\ 1101\downarrow \\ \hline 1000 \\ -0000 \\ \hline 10000 \\ -1101 \\ \hline 00111 \\ -0000 \\ \hline 1110 \\ -1101 \\ \hline 0001 = 1. \end{array}$$

Some basic knowledge

* Quotient only contain 0 or 1.
* first multiply with 1 and subtract divisor from dividend
  → if reminder gives +ve means we can divide (Put 1 in quotient)
  → if reminder gives -ve mean we can't divide (put 0 in quotient)
  ↳ you will restore it

Restoring Division :-

→ In the circuit diagram (given in the previous page)

Initial :

* n-bit positive divisor is loaded into register M at the start of operation.
* n-bit positive dividend is loaded into register Q at the start of operation
* Register A is set to '0'.

After divison complete

* n-bit quotient is in Register Q.
* reminder is in register A.
* The extra bit position at the left end of both A and M accomodates the sign bit during subtraction.

Algorithm for Restoring division :-

→ DO the following steps for n times.

1. Shift A and Q left one binary position
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A; otherwise set $q_0 = 1$.

→ This method is called restoring division because after subtraction, if the reminder is negative then we restore partial dividend by adding divisor to the Reminder which is a -ve value.

Ex :- 13 | 1

-9

-13 → -ve value → (+13) → Add divisor → partial dividend

-4 → 9 → get original divisor

-4

ample :- Divide 8 ÷ 3 by using restoring division.

M : Divisor → 3 → 0011

Q : Divident → 8 → 1000

A = A-M+M

A - M → A + (-M)

-M → -3 → 11101

3 → 00011
1's comp: 11100
+1
-3 → 11101

sign bit

Operation	A	Q	
Initially	0 0 0 0 0	1 0 0 0	
Left Shift	0 0 0 0 1	0 0 0 □	
Sub M (A + (-M))	+ 1 1 1 0 1		
	(1) 1 1 1 0		
	sign bit		
Set q_0 and Add M	+ 0 0 1 1		
	(X) 0 0 0 0 1	0 0 0 0	★
	Discard		

Cycle 1

After subtraction operation (if sign bit of A is 1) that means -ve, so Set $q_0 = 0$ and restore

Left Shift	0 0 0 1 0	0 0 0 □	
Sub M (A + (-M))	+ 1 1 1 0 1		
	(1) 1 1 1 1		
	sign bit		
Set q_0 and Add M	+ 0 0 1 1		
	(X) 0 0 0 1 0	0 0 0 0	
	Discard		

Cycle 2

Left Shift	0 0 1 0 0 (4)	0 0 0 □	
Sub M (A + (-M))	+ 1 1 1 0 1 (-3)		
	(X) (0) 0 0 0 1		
	Discard, sign bit		
Set q_0	0 0 0 0 1	0 0 0 1	★

Cycle 3

In this cycle the sign bit is 0 that means the (4-3 = 1) so set $q_0 = 1$ and No restore

Left Shift	0 0 0 1 0	0 0 1 □	
Sub M (A + (-M))	+ 1 1 1 0 1		
	(1) 1 1 1 1		
Set q_0 and Add M	+ 0 0 1 1		
	(X) 0 0 0 1 0	0 0 1 0	
	Reminder	Quotient	

Cycle 4

3 | 8 | 2 → Quotient
-6
2 → reminder

Non-Restoring Division :-

→ The restoring division can be improved by avoiding the need for restoring A after an unsuccessful subtraction.

→ Subtraction is said to unsuccessful if the result is negative.

→ Consider the sequence of operation that takes place after the subtraction operation in restoring algorithm :

* If A is positive, set q_0 to 1 and in the next cycle we perform left shift operation and subtract M

i.e finally we perform → **2A − M**

Left shift A : 2A (NOW A ← 2A)
Sub M : A − M (NOW A ← 2A − M)

left shift means Multiply by 2.

Ex 100 = 4
left shift by 1
1000 = 8 (4×2)

Ex : No : 5 = 101
left shift by 1
1010 = 10 (5×2)

* If A is negative, set q_0 to 0 and add M then in the next cycle we perform left shift operation and subtract M.

i.e we perform

Add M to A : A + M (NOW A ← A + M)
Left Shift A : 2(A + M) (NOW A ← 2A + 2M)
Sub M : A − M (NOW A ← 2A + 2M − M
A ← 2A + M)

i.e finally we perform **2A + M**

Algorithm for non-restoring division

Step-1: Do the following n times:

1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise shift A and Q left and add M to A.
2. Now, if the sign of A is 0, set q_0 to 1; otherwise set q_0 to 0.

Step-2:- If the sign of A is 1, add M to A.

Example: Divide 8 ÷ 3 by using non-restoring division algorithm.

M → 3 → 0011
-M → -3 → 11101
Q → 8 → 1000

Operation	A	Q	
Initially	00000	1000	
Left Shift	00001	000□	cycle 1
Sub M	+11101		
A+(-M)	11110	0000	
Left Shift	11100	000□	
Add M	+0011		
	11111	0000	cycle 2
Left Shift	11110	000□	
Add M	+0011		
	00001	0001	cycle 3
Left Shift	00010	001□	
Sub M	+11101		
A+(-M)	11111	0010	cycle 4

Sign bit of A=1 then we perform 2A+M left shift then and add M

Sign bit of A = 1 then we perform left shift then add M.

Sign bit of A = 0 then we perform left shift then sub M.

Now cycle 4 is the end of cycle.

Now perform step-2. if sign bit of A is 1 then add M to A:

```
  11111
+ 0011
  00010
```

Quotient: Q: 0010 = 2

Reminder = 00010 = 2

Assignment 1-

① $10 \div 4$ using restoring.

$Q = 10 = 1010$, $M = 4 = 0100$, $-M = 11100$

Operation	A	Q	
Initialize	00000	1010	
Left Shift	00001	010□	C1
Sub M	+11100		
	(1)11101	010□	
set Q_0 and Add M	+0100		
	~~0~~00001	0100	
Left Shift	00010	100□	C2
Sub M	+11100		
	(1)11110	100□	
Set Q_0 and Add M	+0100		
	~~0~~00010	1000	
Left Shift	00101	000□	C3
Sub M	+11100		
	~~0~~(1)00001	000□	
Set Q_0	00001	0001	
Left Shift	00010	001□	
Sub M	+11100		
	(1)11110	001□	
Set Q_0 an Add M	+0100		
	~~0~~00010	0010	

Remainder = 2 Quotient = 2

② $10 \div 4$ using non-restoring

$Q = 10 = 1010$, $M = 0100$, $-M = 11100$

Operation	A	Q	
Initialize	00000	1010	
Left Shift	00001	010□	cycle1
Sub M (A+(-m))	+11100		
	(1)11101	010[0]	
Left Shift	11010	100□	cycle2
Add M	+00100		
	(1)11110	100[0]	
Left Shift	11101	000□	cycle3
Add M	+0100		
	~~1~~(0)00001	000[1]	
Left Shift	00010	001□	cycle4
Sub M (A+(-M))	+11100		
	11110	001[0]	

A = 11110
+M = 0100
~~1~~00010

A = 00010 = 2

Q = 0010 = 2

$4 \overline{)10}\ 2$
$\quad 8$
$\quad\ 2$

Floating Point Numbers And Operation :-

IEEE standard for floating point numbers :-

General form and size of floating point numbers in decimal system is

$$\pm X_1 \cdot X_2 X_3 X_4 X_5 X_6 X_7 \times 10^{\pm Y_1 Y_2}$$

(multiply)

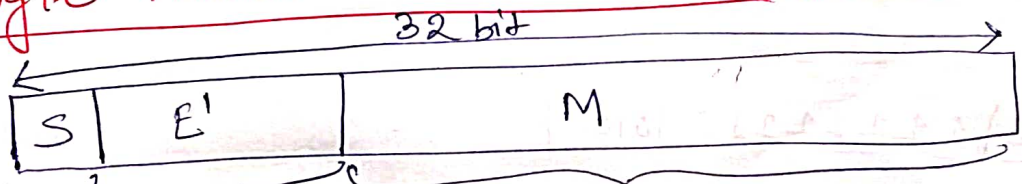
where X_i and Y_i are decimal digits.

→ Basic components of IEEE floating point no. are

(i) sign (S)
(ii) Exponent (E)
(iii) Mantissa (M)

→ floating point no. < single precision (word length = 32 bits)
Double precision (word length = 64 bits)

single precision IEEE floating point number :-



1 — Sign of number 0 → + 1 → −

8-bit signed exponent in excess-127 representation

23-bit Mantissa fraction

Value represented = $\pm 1 \cdot M \times 2^{E'-127}$

$E' = E + 127$ (in Excess-127) $\quad -127 \le E \le 128$

E' is in the range $0 \le E' \le 255$

→ 0 and 255 are used to represent special values.

→ Therefore, the range of E' for normal values is $1 \le E' \le 254$. Actual exponent E is in the range $-126 \le E \le 127$.

→ The scale factor range 2^{-126} to 2^{+127}.

→ Example :- $V = \pm 1.M \times 2^{E'-127}$

① | 0 | 00101000 | 0010100... |
S E' M

Value represented $= +1.001010\ldots \times 2^{-87}$

$E' = 00101000$
$= 32+8 = 40$

$E = E' - 127$
$= 40 - 127$
$= -87$

②

| 0 | 11011000 | 010101... |
S E' M

$V = +1.010101\ldots \times 2^{+89}$

$E' = 11011000$
$= 128+64+16+8$
$= 216$

$E = E' - 127$
$= 216 - 127$
$= 89$

③

| 1 | 11111110 | 01000... |

$V = -1.01000\ldots \times 2$

$E' = 11111110$
$= 128+64+32+16+8+4+2$
$=$

* MSB of the mantissa is always equal to 1. This bit is not explicitly represented; it is assumed to be to the immediate left of the binary point.

* Hence, 23 bits stored in the M field actually represent the fractional part of the mantissa i.e the bits to the right of the binary point.

2-

$V = 312$

$= 100111000$ (128, 64, 32, 16, 8, 4, 2, 1; 256)

$= 1.00111000 \times 2^8$

312
-256
56
-32
24
-16
8

$E = E' - 127$

$\Rightarrow E' = E + 127$

$= 8 + 127$

$= 135$

-128

7

0	10000111	00111000
S	E'	M

$S = 0, \quad E' = 10000111$

$M = 00111000$

Normalization of floating point :-

Ex-1 :

→ Suppose a no. is $0.0010110\cdots \times 2^9$ which is a unnormalized no. because before the binary point 1 must present (i.e $1.\cdots$).

→ The corresponding normalized value is

$0.0010110\cdots \times 2^9$ (unnormalized)

$\Downarrow$

$1.0110\cdots \times 2^6$ (normalized

Now the memory representation of this no. is

$E = 6 \quad E' = E + 127 = 6 + 127 = 133$

$128 + 5$

0	10000101	0110...
S	E'	M

Ex-2 :

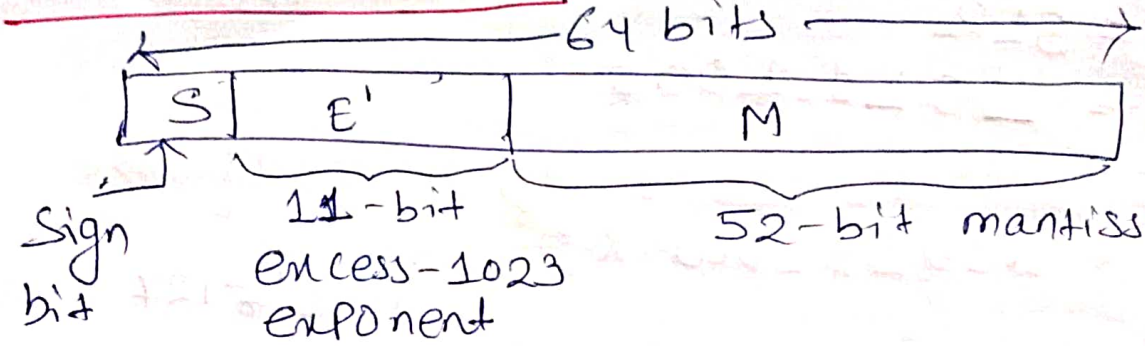
A no : $101.01101\cdots \times 2^7$ (un normalized)

$1.01011101\cdots \times 2^9$

$E = 9, \quad E' = E + 127 = 9 + 127 = 138$

0	10001000	01011101...

Double precision :-



$$\text{Value} = \pm 1.M \times 2^{E' - 1023}$$

$0 \leq E' \leq 2047$

→ However 0 & 2047 are used to indicate special values.

→ For normal values $1 \leq E' \leq 2046$

→ The actual exponent E is in the range $-1022 \leq E \leq 1023$, providing scale factor of 2^{-1022} to 2^{1023}.

Overflow and underflow :-

→ In single precision the scale factor is 2^{-126} to 2^{+127}

→ If a number (its normalized representation requires an exponent less than -126) then we can say that underflow has occured.

→ If a number whose normalized representation requires an exponent greater than +127 then we can say that overflow has occured.

→ Both underflow and overflow are considered as arithmatic exceptions.

ecial values!-

→ The end values 0 and 255 of the excess-127 exponent E' are used to represent special values

* When E' = 0 and the mantissa fraction M is zero, the value exact 0 is represented.

* When E' = 255 and M = 0, the value ∞ is represented, ($\infty = \frac{\text{normal no.}}{\text{zero}}$)

* Due to sign bit, ±0, ±∞ are there

* When E' = 0, and M ≠ 0, denormal numbers are presented.

$$V = \pm 0.M \times 2^{-126}$$

→ gradual underflow.

→ smaller than smallest normal number.

$V = \pm 1.M \times 2^{-127}$ → underflow

to remove underflow.

$V = \pm 0.M \times 2^{-126}$ → comes in range

But it is a denormal number

* When E' = 255, M ≠ 0, the value represents a Not a Number (NaN)

→ A NaN is the result of performing an invalid operation such as 0/0 or $\sqrt{-1}$

E'	M	Description
0	0	0
255	0	∞
0	≠0	Denormal no.
255	≠0	NaN

E' = 0 → 00000000 → All zeros
E' = 255 → 11111111 → All ones

Exceptions :-

→ A Processor must set exception flags if any one of the following occur in performing operations: underflow
overflow
divide by zero
inexact
invalid.

→ underflow : (Discussed)
→ overflow : (Discussed)
→ divide by zero : (Discussed)

→ Inexact :- Inexact is the name for a result that requires rounding in order to be represented in one of the normal formats

→ An Invalid exception occurs if operations such as $0/0$ or $\sqrt{-1}$ are attempted.

→ When exceptions occure, the results are set to special values.

Arithmatic operations on floating point numbers

① Add / subtract rule :-

(1) Choose the no. with the smaller exponent and shift its mantissa right a no. of steps equal to the difference in ~~exception~~ exponent.
(2) Set the exponent of the result equal to the largen exponent.
(3) Perform Add / Subtraction on the mantissas and determine the sign of the result.
(4) Normalize the result value, if necessary.

Multiply Rule :-

1. Add the exponents and subtract 127.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide rule :-

1. subtract the exponents and add 127.
2. Divide the mantissas and determine the sign of result.
3. Normalize the resulting value, if necessary.