Unit-02

Arithmetic & Logic Unit

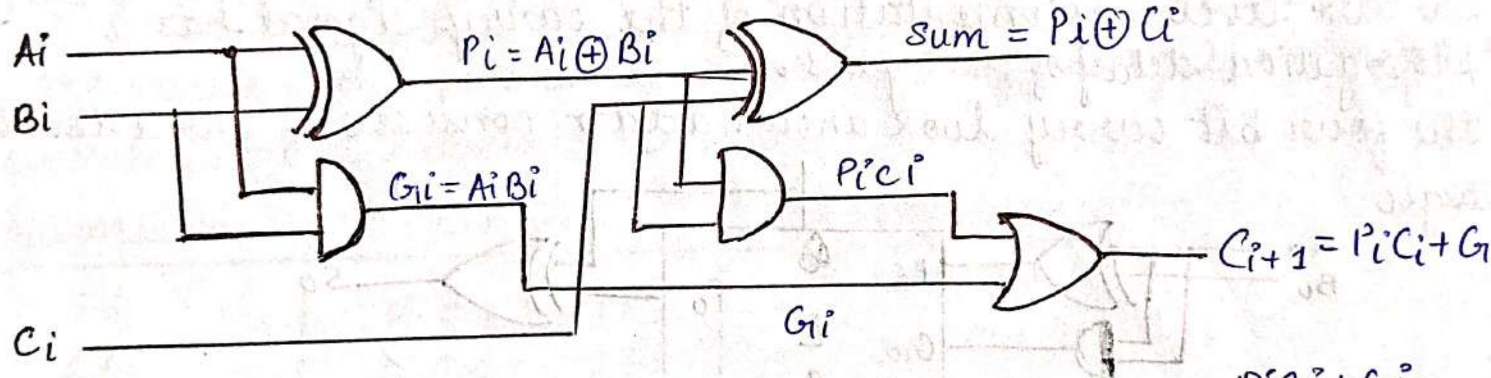
Look ahead Carry Adder

To reduce the propogation delay

To overcome the problem of excessive delay in ripple carry Adder, a fast adder known as look ahead carry adder can be design. In this adder the carry-in for various stages can be generated directly by the logical expressions.

- The general method for design the fast adder is to reduce the time required to form carry signals.
- It uses the logical gates to look at lower order bits of the data to see if a higher order carry is to be generated.
- It uses two functions
  ① carry generate
  ② carry propogate

Following figure shows the full adder circuit used to add a operand bits



From the given circuit $Sum = P_i \oplus C_i$ and Carry $C_{i+1} = P_iC_i + G_i$

- G_i is known as carry generated signal
- A carry C_{i+1} is generated whenever $G_i = 1$, regardless of the inpu carry.
- P_i is known as carry propogate whenever $P_i = 1$ the input carry is propogated to the output carry
- The boolean expression for the carry output of various stages can be written as follows :-

$i=0$ $\quad C_1 = G_0 + P_0 C_0$

$C_1 = [A_0 B_0] + [A_0 \oplus B_0] C_0$

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

$i=0$

$$\boxed{C_1 = P_0 C_0 + G_0}$$

$i=1$

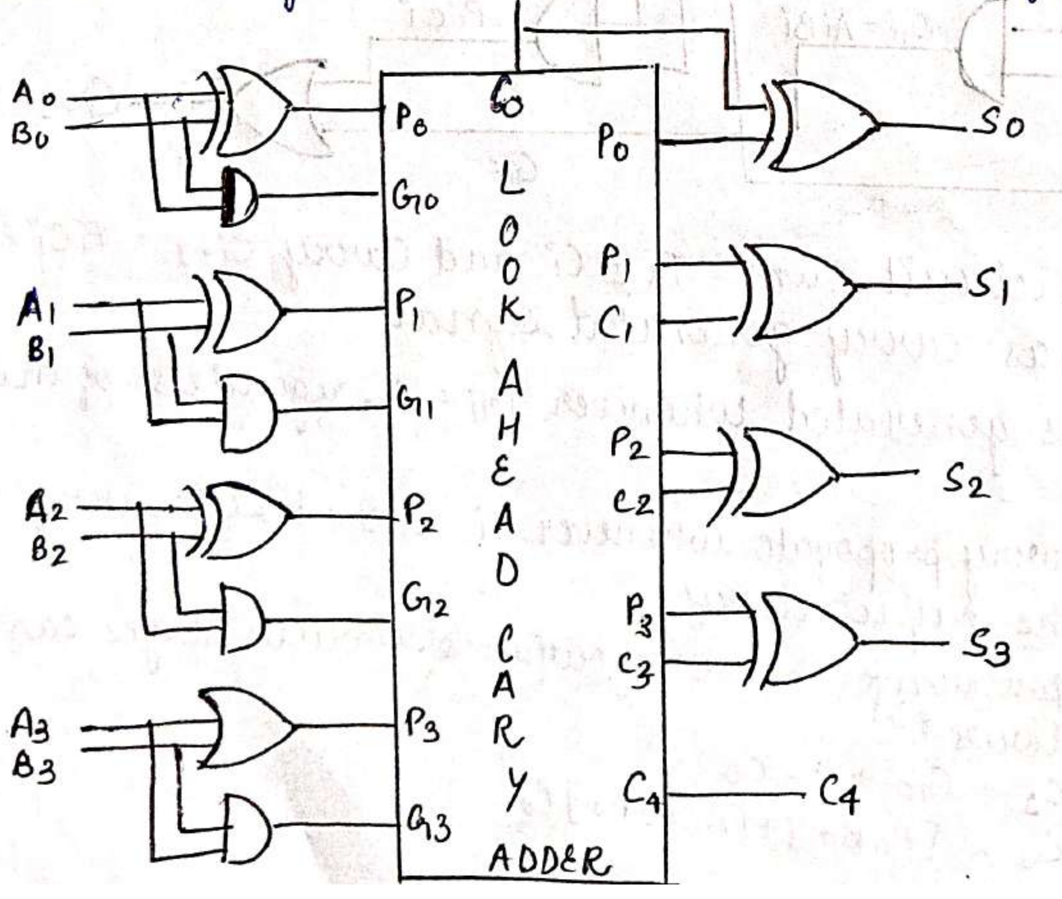
$$C_2 = P_1 C_1 + G_1$$
$$= P_1 [P_0 C_0 + G_0] + G_1$$
$$\boxed{C_2 = P_1 P_0 C_0 + P_1 G_0 + G_1}$$

$i=2$

$$C_3 = P_2 C_2 + G_2$$
$$= P_2 [P_1 P_0 C_0 + P_1 G_0 + G_1] + G_2$$
$$\boxed{C_3 = P_2 P_1 P_0 C_0 + P_1 P_2 G_0 + G_1 P_2 + G_2}$$

From the above expression each carry signal is expressed as a direct SOP form C_0.

- The two level implementation of the carry signal has a propogation delay of two gates.
- The four bit carry look ahead adder consists of three levels of logic



- In the first level it generates all the P and G signals and it provide t_{pd} delay.
- In the second level the carry look ahead adder consists of four two ~~in~~ level implementation logic circuit that generates carry signal C_1, C_2, C_3 and C_4 as defined by the above expression & it provides $2t_{pd}$ time delay.
- In the third level four XOR gates which generates the sum (S_0, S_1, S_2, S_2) and it provides t_{pd} time delay. Therefore the total time delay by the adder is $4t_{pd}$

Signed number Representation

- 2's representation is the best choice for addition of two signed numbers.
- If X & Y are the two positive numbers then
  - $X - Y = X + (-Y) = X + (2\text{'s comp}^n \text{ of } Y)$
  - $-X + Y = (2\text{'s comp}^n \text{ of } X) + Y$
  - $-X - Y = (2\text{'s comp}^n \text{ of } X) + (2\text{'s comp}^n \text{ of } Y)$

Find the value of the following expression by using 2's compliment.

(i) 55 + 27

consider the size of register to be 8 bits.

Number	Binary	2's complement
55	00110111	11001001
27	00011011	11100101

```
     1  1 111
  0 0 1 1 0 1 1 1
  0 0 0 1 1 0 1 1
  ---------------
  0 1 0 1 0 0 1 0
```

(ii) −55 + 27

```
      1 1   1 1
    1 1 0 0 1 0 0 1
  + 0 0 0 1 1 0 1 1
  -----------------
    1 1 1 0 0 1 0 0   ← -28
  -----------------
2's comp → 0 0 0 1 1 1 0 0   (28)
```

(iii) 55 − 27

```
   1   1 1 1
   0 0 1 1 0 1 1 1
   1 1 1 0 0 1 0 1
  -----------------
 (1)0 0 0 1 1 1 0 0
  └ Discard
```

Binary Multiplication (Unsigned number)

As compared to the addition and subtraction multiplication is a complex operation whether perform in a hardware or software.

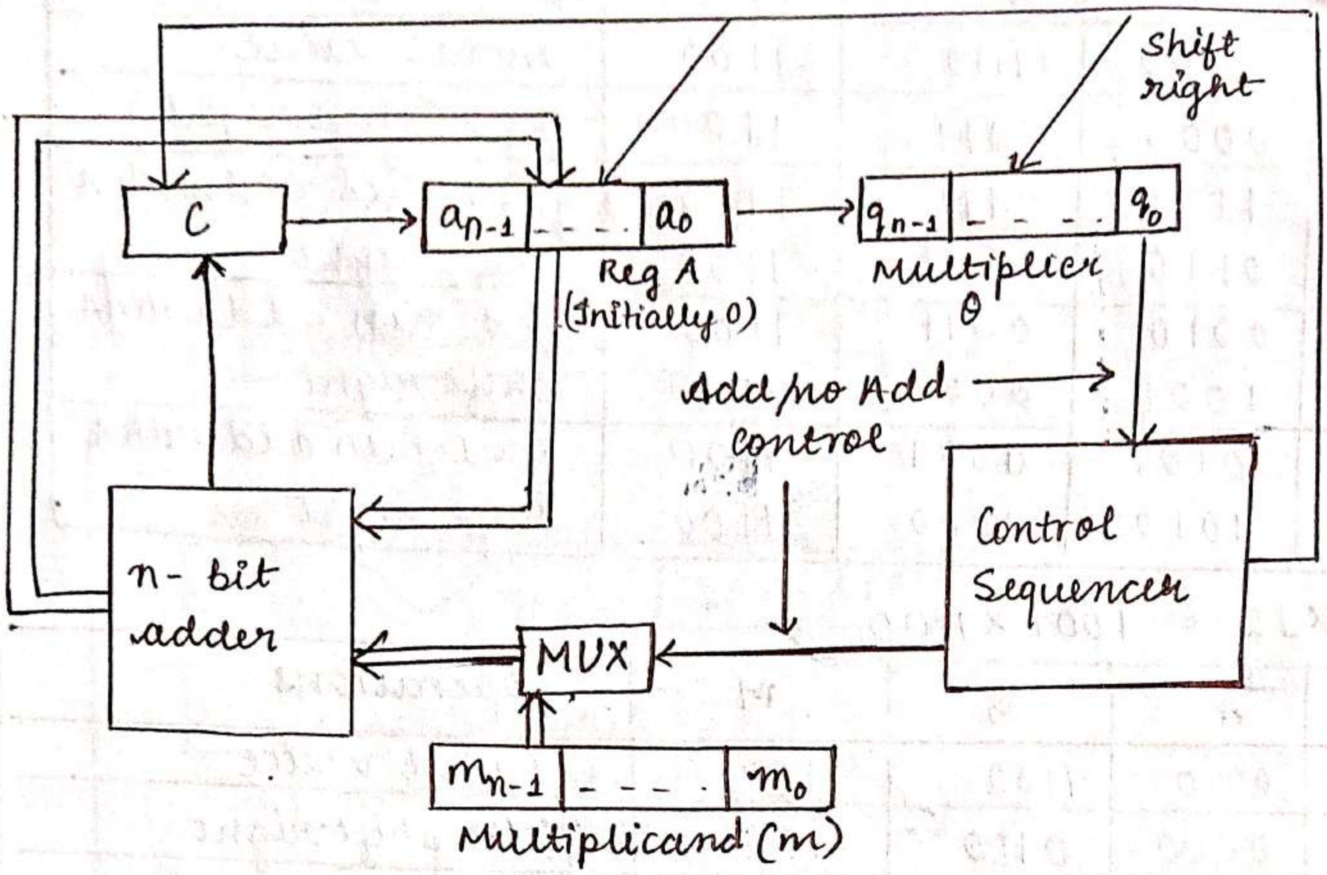
- For <u>multiplication of unsigned binary integers</u> following points are considered –

1) Multiplication involves the generation of partial product, one for each digit in the multiplier.
2) There partial products are then sumed to produce the final product.
3) The partial product are easily defined when the multiplier bit is zero. then partial product is zero.
4) The total product is produced by summing the partial products, for this operation each successful partial product is shifted one position to the left related to the preceding partial products.

Imp 5) The multiplication of two (n-bits) binary number (Integer) produces a product of upto 2n bits in length. Therefore the product of the two 4 bit numbers fit in the eight bit length.

```
        1 0 1 1    (Multiplicand)(M)
        0 1 0 1    Multiplier (Q)
     ----------
      0 1 0 1 1
      0 0 0 0
    1 0 1 1
  0 0 0 0
 --------------
 0 1 1 0 1 1 1
```

- In the binary system multiplication of the multiplicant by 1 bit at multplier is easy.
- If the multiplier bit is 1, the multiplicant is entered in the appropiate position to be added to the partial product.



The operation of the multplier is as follows:-

- Control sequencer reads the bits of the multiplier one at a time.
- If q_{n-1} then, multplicant (M) is added with the register A and the result is stored in the A register with the C bit is used ~~Q register~~ far overflow.
- Then, all the bits of the C, A and Q register are shifted to the right 1 Bit. So that the C bit goes to a_{n-1} and a_0 goes to q_{n-1} and q_n is last.
- If $q_0 = 0$ then no addition is performed just the shift right.

This process is repeated her each bit of the origenal multiplier.

Ex:- Multiply the 1011 X 1101

C	A	Q	M	Operation
0	0000	1101	1011	Initial value
0	1011	1101	1011	$q_0=1$ then add with A
0	0101	1110	1011	shift right
0	0010	1111	1011	shift right ($q_0=0$)
0	1101	1111	1011	$q_n=1$ then add with A
0	0110	1111	1011	shift right
1	0001	1111	1011	$q_0=1$ then add with
0	1000	1111	1011	[illegible] shift right

i) 12X19 = 1100 X 1110

C	A	Q	M	Operations
0	0000	1110	1100	Initial value
0	0000	0111	1100	$q_0=0$ (shift right)
0	1100	0111	1100	$q_0=1$ then add with A
0	0110	0011	1100	shift right
1	0010	0011	1100	$q_0=1$ then add with A
0	1001	0001	1100	shift right
1	0101	0001	1100	$q_0=1$ then add with A
0	1010	1000	1100	shift right

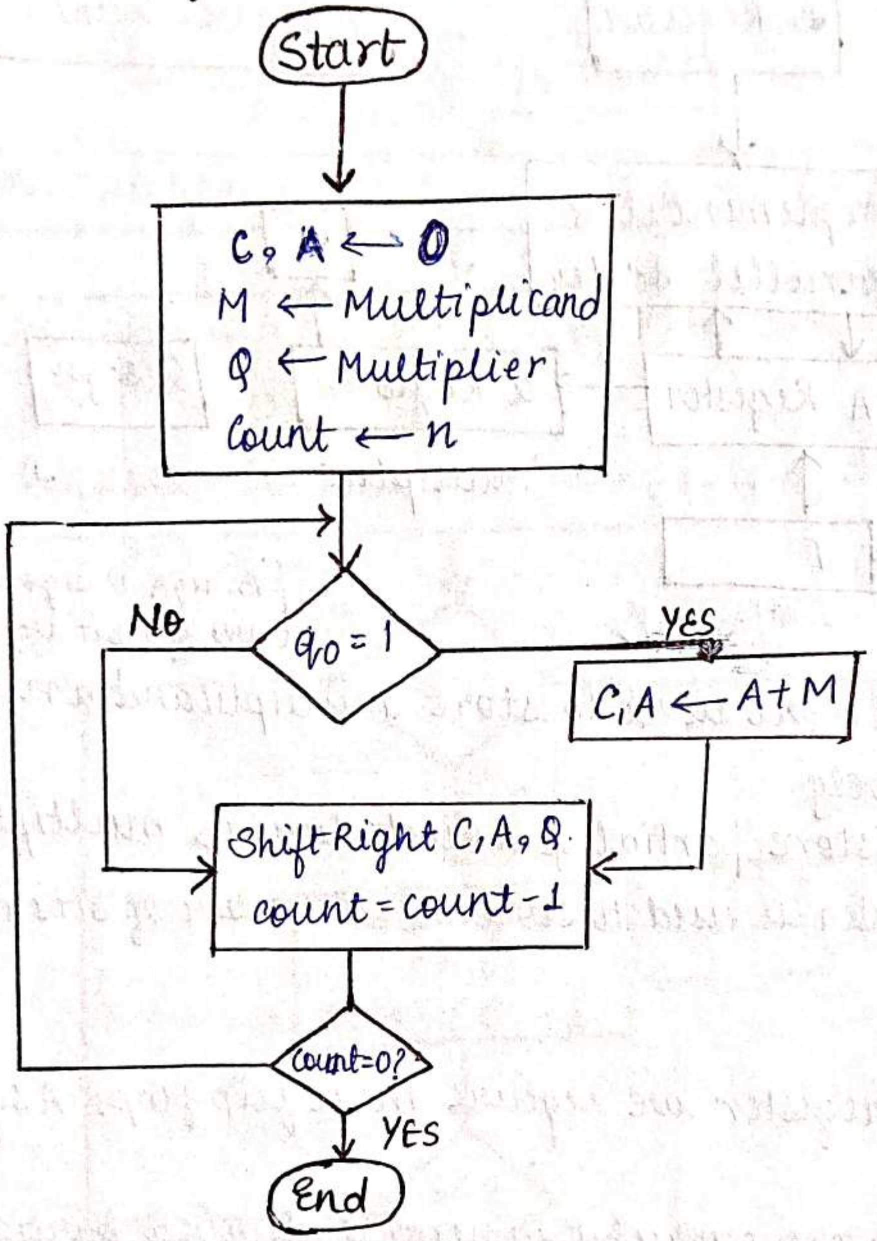
ii) 9X12 = 1001 X 1100

C	A	Q	M	Operations
0	0000	1100	1001	Initial value
0	0000	0110	1001	$q_0=0$ shift right
0	0000	0011	1001	$q_0=0$ shift right
0	1001	0011	1001	$q_0=1$ then add with A
0	0100	1001	1001	shift right
0	1101	1001	1001	$q_0=1$ then add with A
0	0110	1100	1001	shift right

iii) 19 X 21 10011 X 10101

C	A	Q	M	Operations.
0	00000	10101	10011	Initial value
0	10011	10101	10011	$q_0 = 1$ add with A
0	01001	11010	10011	shift right
0	00100	11101	10011	$q_0 = 0$ then shift right
0	10111	11101	10011	$q_0 = 1$ then add with A
0	01011	11110	10011	shift right
0	00101	11111	10011	$q_0 = 0$ shift right
0	11000	11111	10011	$q_0 = 1$ then add with A
0	01100	01111	10011	Shift Right

A flowchart of the Multiplication is as shown:-

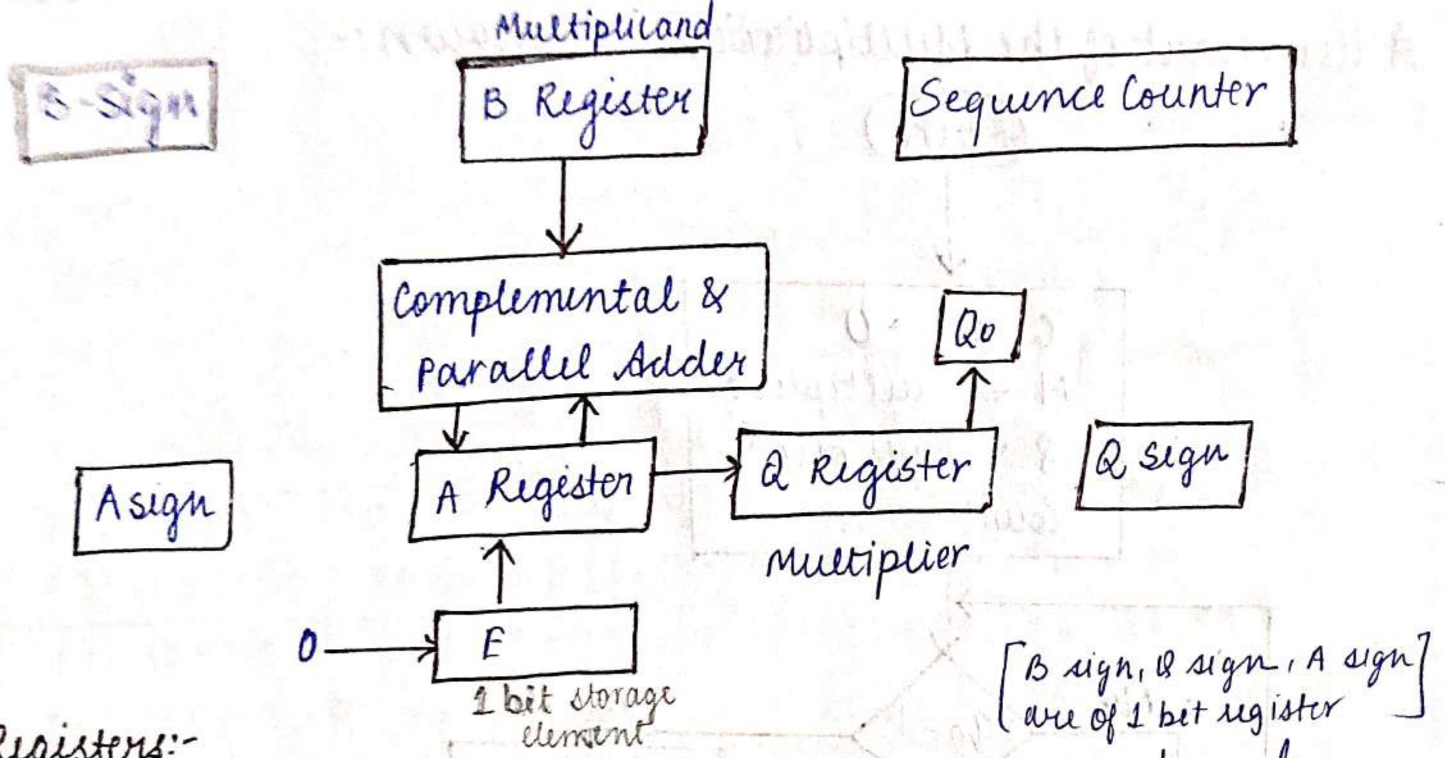


Multiplication algorithm in signed magnitude representation -

- The sign of the product is determined from the sign of the multiplicand and multipeier
- If they are same, sign of the product is positive else negative. (based on XOR gates).

Hardware Implementation

The components require for the hardware implementation of the multiplication algorithm is given by



Registers:-

- Two registers B and Q are used to store multiplicand and multiplier s respectively.
- Register A is used to store partial product during multiplication
- Sequence countor register is used to store the number of bits in the multiplier.

Flip flops:-

- To store the sign bit register we require three flip flops A sign, B sign or Q sign
- Flip flop E is used to store carry bit generated during partial product addition.

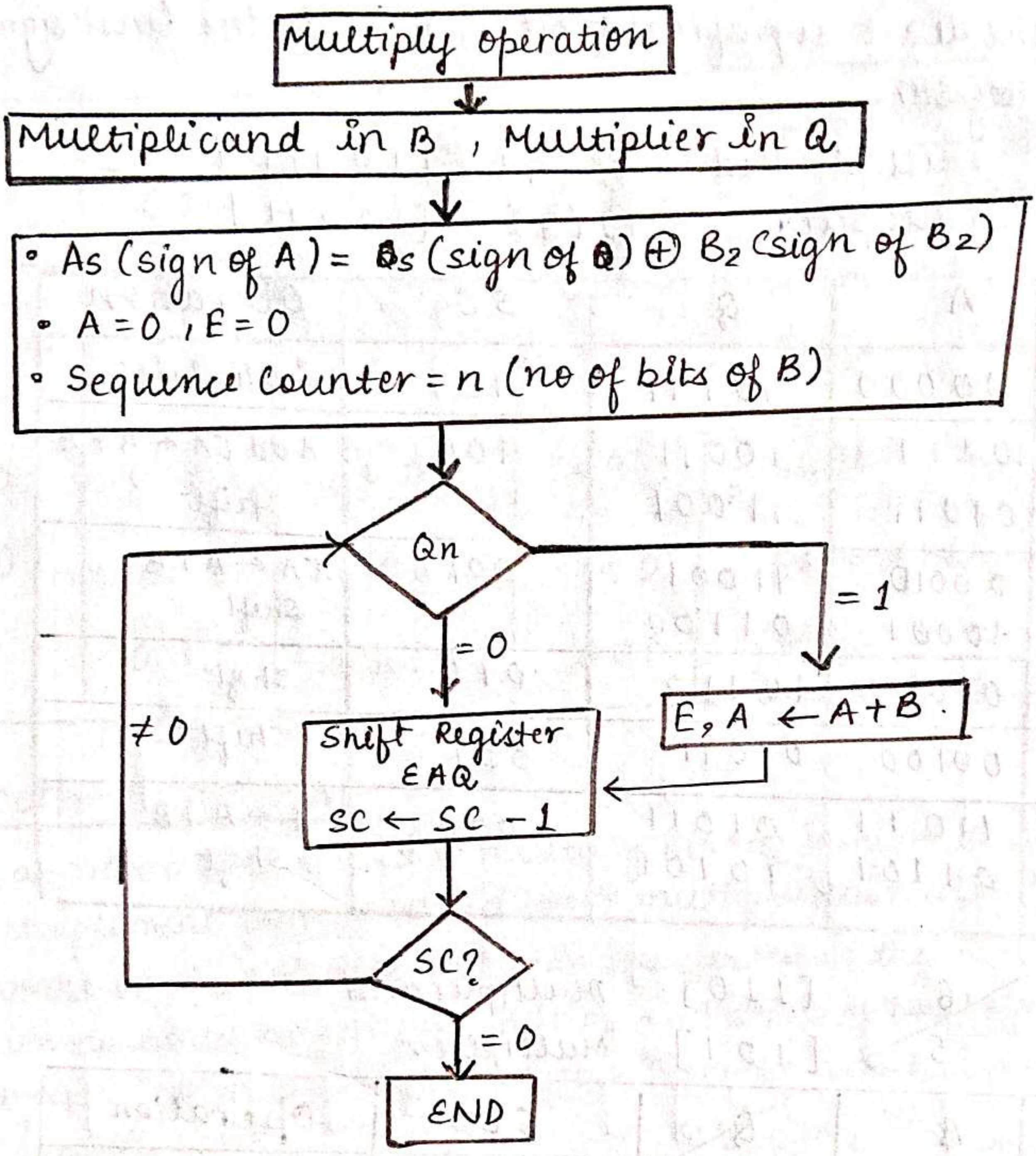
Complement and parallel Adder

This hardware unit is used to calculate partial products. (This is it performs the required addition)

operation:-

- Initially multiplicand is stored in regester B and the multiplier is stored in register Q.
- Sign of the B and Q are compared by using the XOR function & the output is stored in the A sign.
- Initially ~~*~~ 0 is assigned to register A and E flip flop.
- Sequence counter is initialized with value n. where n is the number of bits in the multiplicand.

Flowchart for multiplication



Multiply operation
Multiplicand in B , Multiplier in Q
• As (sign of A) = Bs (sign of B) ⊕ B2 (sign of B2)
• A = 0 , E = 0
• Sequence Counter = n (no of bits of B)
Qn
= 1
= 0
E, A ← A+B.
≠ 0
Shift Register EAQ
SC ← SC − 1
SC?
= 0
END

• Now the LSB of the multiplier is checked.
• If it is 1 add the content of Register A with B multplicand and then result in assigned in register A with the carry bit in flip flop E.
• Now the content of the EAQ is shifted to the right by 1 position.
• If Qn = 0 then only shift right operation on the content of EAQ is performed and the content of sequence counter decremented by 1.
• Now check the content of sequence counter if it is 0 end the process and the final product is present in the register A & Q.

else repeat the process.

- A hold the most significant bit & Q holds the least significant bit of the register.

Example:- Multiplicand = 23 (Q) [10111]
Multiplier = 19 (Q) [10011]

E	A	Q	SC	Operation
0	00000	10111	101	Initial value
0	10111	10011	100	Add $EA \leftarrow A+B$
0	01011	11001		Shift
1	00010	11001	001	$EA \leftarrow A+B$
0	10001	01100		Shift
0	01000	10110	010	Shift
0	00100	01011	001	Shift
0	11011	01011	000	~~A~~ $\leftarrow A+B$
0	01101	10101		Shift

6 X 5 = 6 = [1 1 0]
5 = [1 0 1]

E	A	Q	SC	Operation
0	000	101	011	Initial value
0	110	101	010	EA ← A+B
0	011	010		Shift
0	001	101	001	Shift
0	111	101	000	EA ← A+B
	011	110	000	Shift

Booth Algorithm

The algorithm that is used to multiply binary integer in signed 2's compliment form is called booth multiplication algorithm.

- It works on the principle that string of zeroes in the multiplier require no addition but just shifting and string of ~~one~~ 1's in the multiplier from bit weight 2^k to 2^m can be treated as $(2^{k+1} - 2^m)$

For Example:- $\overset{}{0}\,\overset{k=4}{1}\,1\,\overset{m=2}{1}\,0\,0$ $(2^{k+1} - 2^m)$
$(2^5 - 2^2)$
$= 32 - 4 = 28$

Therefore the multiplication M X 28, where M is the multiplicand and 28 is the multiplier that can be done by $M \times (2^5 - 2^2)$
$\Rightarrow (M \times 2^5) - (M \times 2^2)$
Therefore the product can be obtained by shifting the binary multiplicand (M) five times to the left and subtracting m shifted left two times

- Booth algorithm requires the examination of the multiplier bits and shifting of the partial product prior to the shifting the multiplicand may be added to the partial product subtracted from the partial product or left unchanged according to the following rules:-

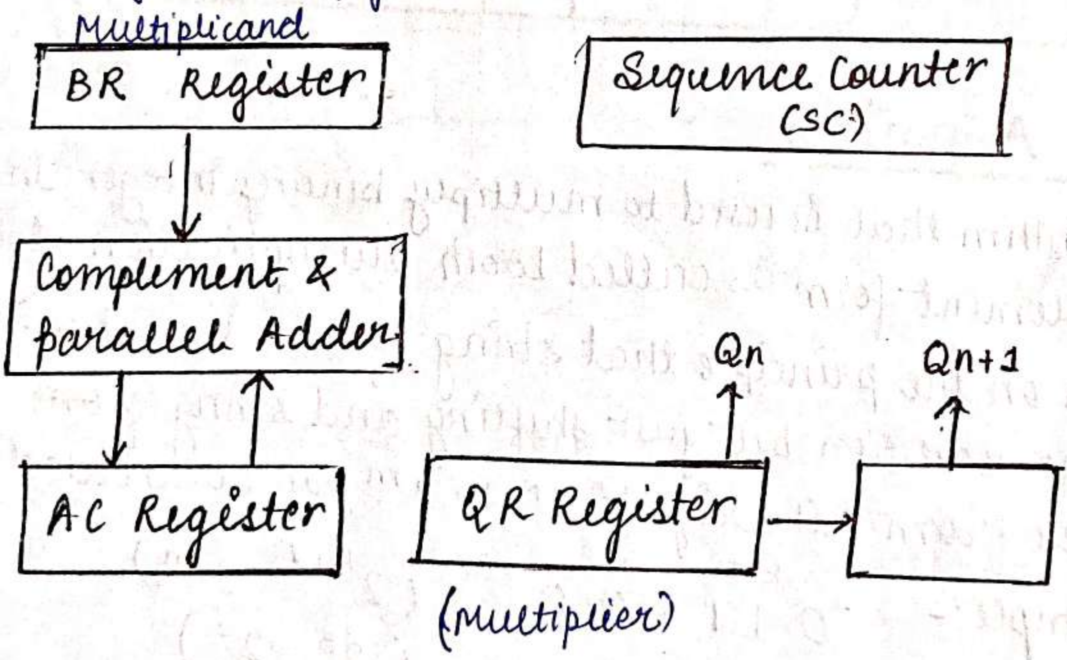
1) The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

2) The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in the string of 1's in the multiplier.

3) The multiplicand is added to the partial product upon encountering the first zero (provided that there was a previous one) is a string of zeroes in the multiplier.

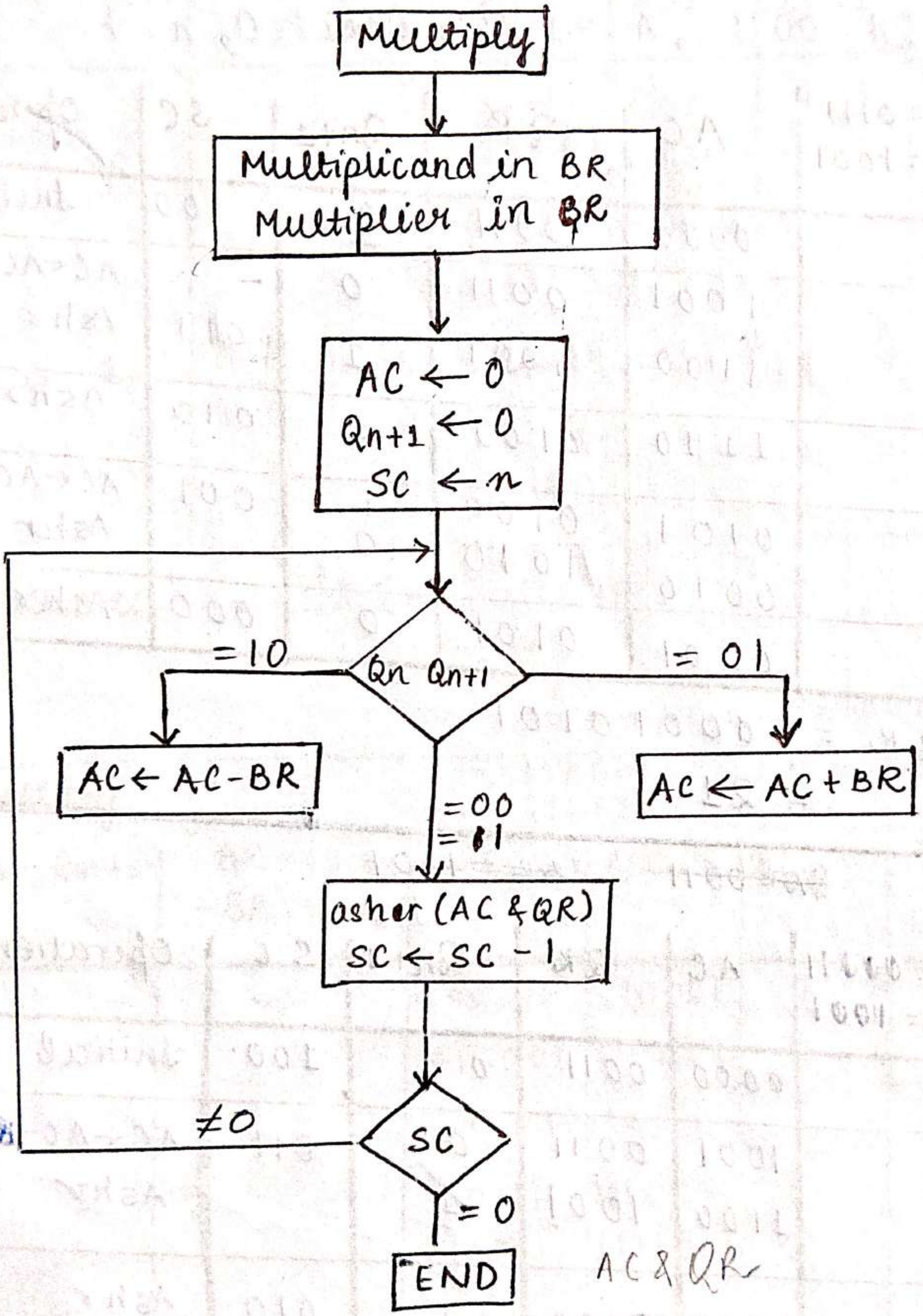
00/11 → unchanged
10 → Subt
01 → Add

The Hardware implementation of the booth alg. requires the register configuration as shown:-



An extra flip flop Qn+1 is include to QR for the double bit insuction of the multiplier.

Flowchart of Booth Algorithm



Q) 7X3.

BR = 0111 , QR = 0011 , AC = 0000, Qn+1 = 0, n = 4

Qn	Qn+1	BR=0111 -BR=1001	AC	QR	Qn+1	SC	Operation
			0000	0011	0	100	Initial
1	0		1001 1100	0011 1001	0 1	011	AC←AC-BR Ashr
			1110	0100	1	010	Ashr
1	1		0101 0010	0100 1000	1 0	001	AC←AC+BR Ashr
0	1						
00	0		0001	0101	0	000	Ashr

Result = AC & QR = 00010101
= 21

Multiply +7X3. ~~QR = 0011~~ ~~BR = 1001~~

Qn	Qn+1	BR= 0111 -BR= 1001	AC	QR	Qn+1	SC	Operation
			0000	0011	0	100.	Initial
1	0		1001 1100	0011 1001	0 1	011	AC←AC−BR. Ashr
1	1.		1110	0100	1	010.	Ashr
0	1.		0[illegible]11 0010	0100 1010	1 0	001	AC←AC+BR Ashr.
0	0		0001	0101	0	000	Ashr

Multiply −7 X 3 BR = 1001

Qn Qn+1	BR = 1001 −BR = 0111	AC	QR	Qn+1	SC	Operation
		0000	0011	0	100	Initial
1 0		0111	0011	0	011	AC←AC−BR
		0011	1001	1		Ashr
		0001	1100	1	010	Ashr
1 1						
0 1		1010	1100	1	001	AC←AC+BR
		1101	0110	0		Ashr
0 0		1110	1011	0	000	Ashr

Result = AC & QR = 1110 1011
= −21
↓
2's complement ⇒ 00010101 ⇒ 21

Multiply −17 X −15 17 = 010001 15 = 001111
BR X QR −17 = 101111 −15 = 110001

Qn Qn+1	BR = 101111 −BR = 010001	AC	QR	Qn+1	SC	Operation
		000000	110001	0	110	Initial
1 0		010001	110001	0	101	AC←AC−BR
		001000	111000	1		Ashr
0 1		110111	111000	1	100	AC←AC+BR
		111011	111100	0		Ashr
0 0		111101	111110	0	011	Ashr
0 −0		111110	111111	0	010	Ashr
1 0		001111	111111	0	001	AC←AC−BR
		000111	111111	1		Ashr
1 1		000011	111111	1	000	Ashr

Result → 000011111111

Solve (+15 X −13) using booth Algor.

15 = 01111 13 = 01101
−15 = 10001 −13 = 10011

Qn	Qn+1	BR = 01111 −BR = 10001	AC	QR	Qn+1	SC	operation
			00000	10011	0	101	Initial
1	0		10001	10011	0	100	AC←AC−BR
			11000	11001	1		Ashr
1	1		11100	01100	1	011	Ashr
0	1		01011	01100	0	010	AC←AC+BR
			00101	10110	0		Ashr
0	0		00010	11011	0	001	Ashr
1	0		10011	11011	0	000	AC←AC−BR
			11001	11101	1		

Result → AC & QR = 1100111101

−15 X −12 15 = 01111 12 = 01100
−15 = 10001 −12 = 10100

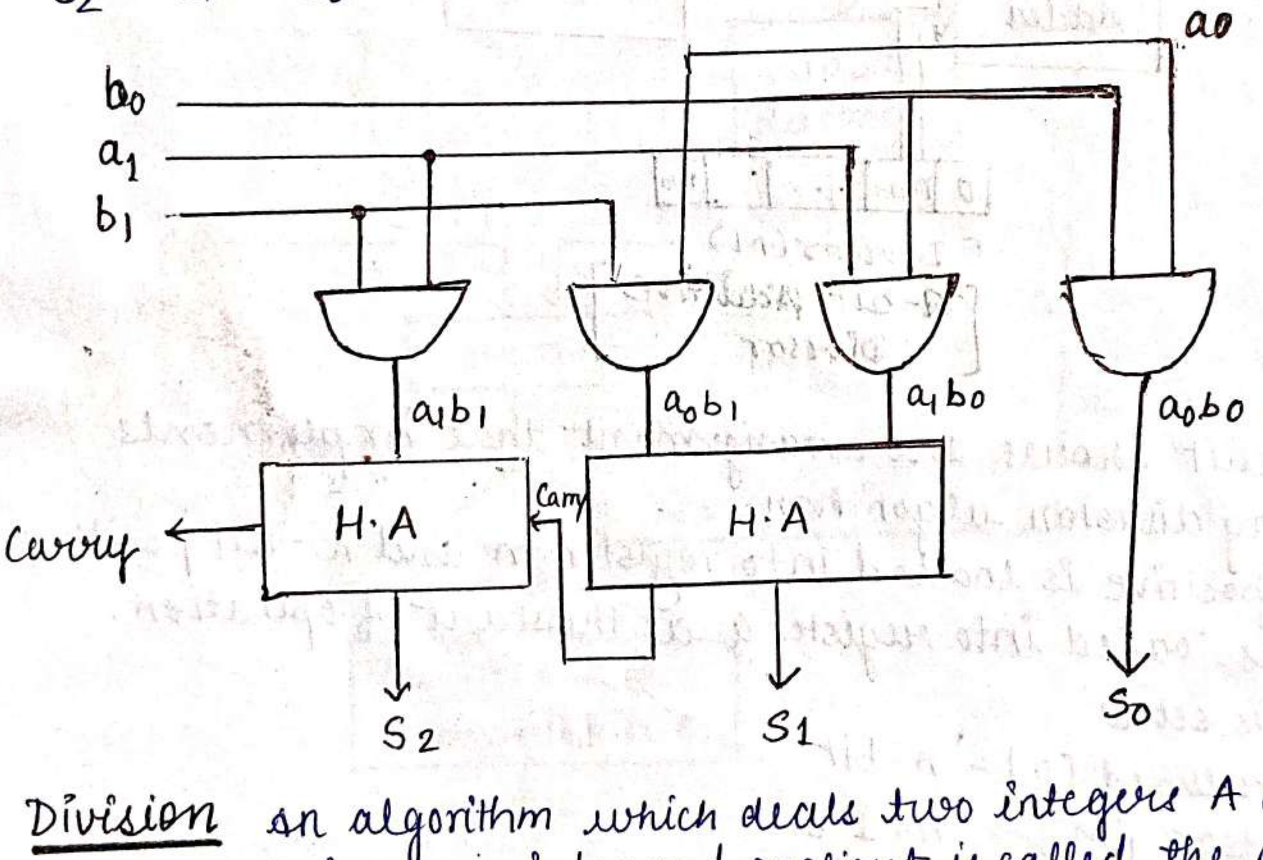
Qn	Qn+1	BR = 10001 −BR = 01111	AC	QR	Qn+1	SC	operation
			00000	10100	0	101	Initial
0	0		00000	01010	0	100	Ashr
0	0		00000	00101	0	011	Ashr
1	0		01111	00101	0	010	AC←AC−BR
			00111	10010	1		Ashr
0	1		11000	10010	1	001	AC←AC+Br
			11100	01001	0		Ashr
1	0		01011	01001	0	000	AC←AC−BR
			00101	10100	1		Ashr

Result :- 0010110100

<u>Array Multiplier</u> An array multiplier is a digital circuit (combinational circuit) used for multiplying two binary numbers by inclined an array of full adders and half adders. ~~This~~

- array is used for simultaneous addition for various product term in word.
- To form the various product term an array of AND gate is used before the adder carrier.
- Consider the multiplication of ~~two~~ ~~of~~ 2 bit numbers as shown –

$$\begin{array}{ccc} & a_1 & a_0 \\ & b_1 & b_0 \\ \hline & a_1b_0 & a_0b_0 \\ a_1b_1 & a_0b_1 & \\ \hline S_2 & S_1 & S_0 \end{array}$$



<u>Division</u> An algorithm which deals two integers A and B computes their remainder and quotient is called the division algorithm.

- Division algorithm are of two types :-

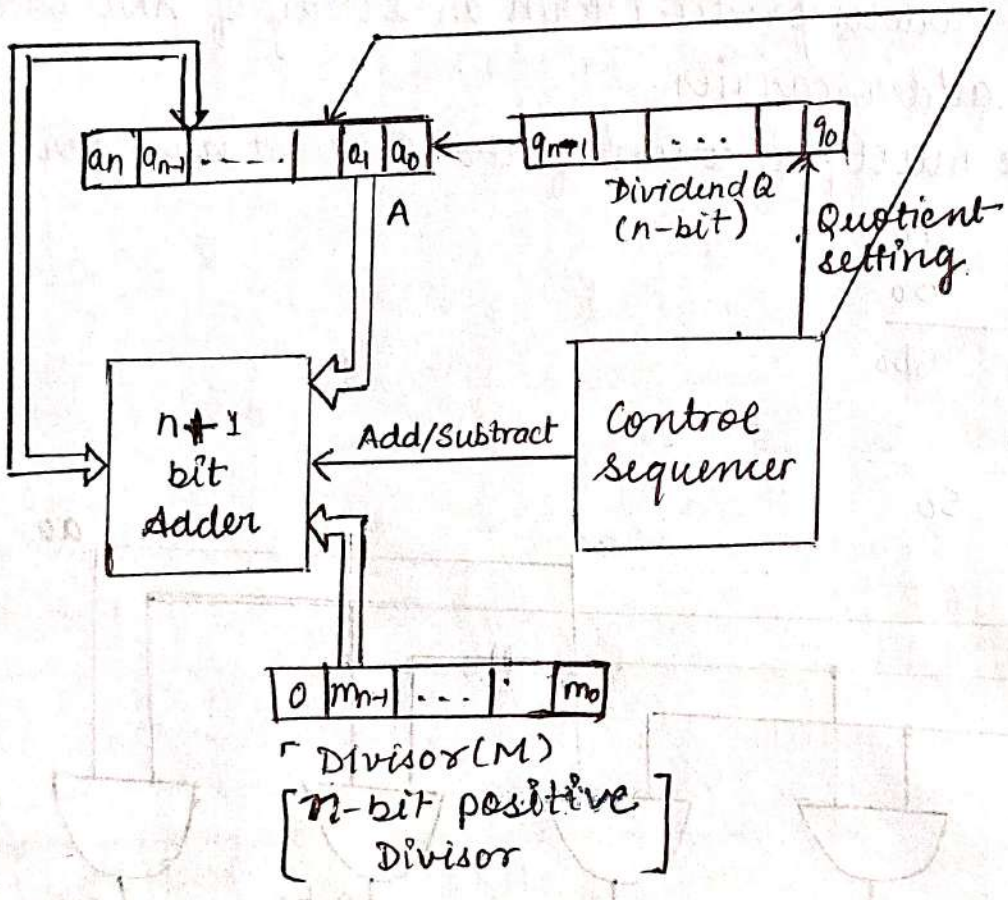
1) Slow algorithm
   a) Restoring algorithm
   b) Non-restoring algorithm

2) Fast algorithm
   a) Newton - Raphson Algo.
   b) SRT division

Restoring division Algo.

Restoring division Algo is a traditional algo for performing binary division.

- In restoring division we restore the partial remainder after each subtraction if the result is negative.



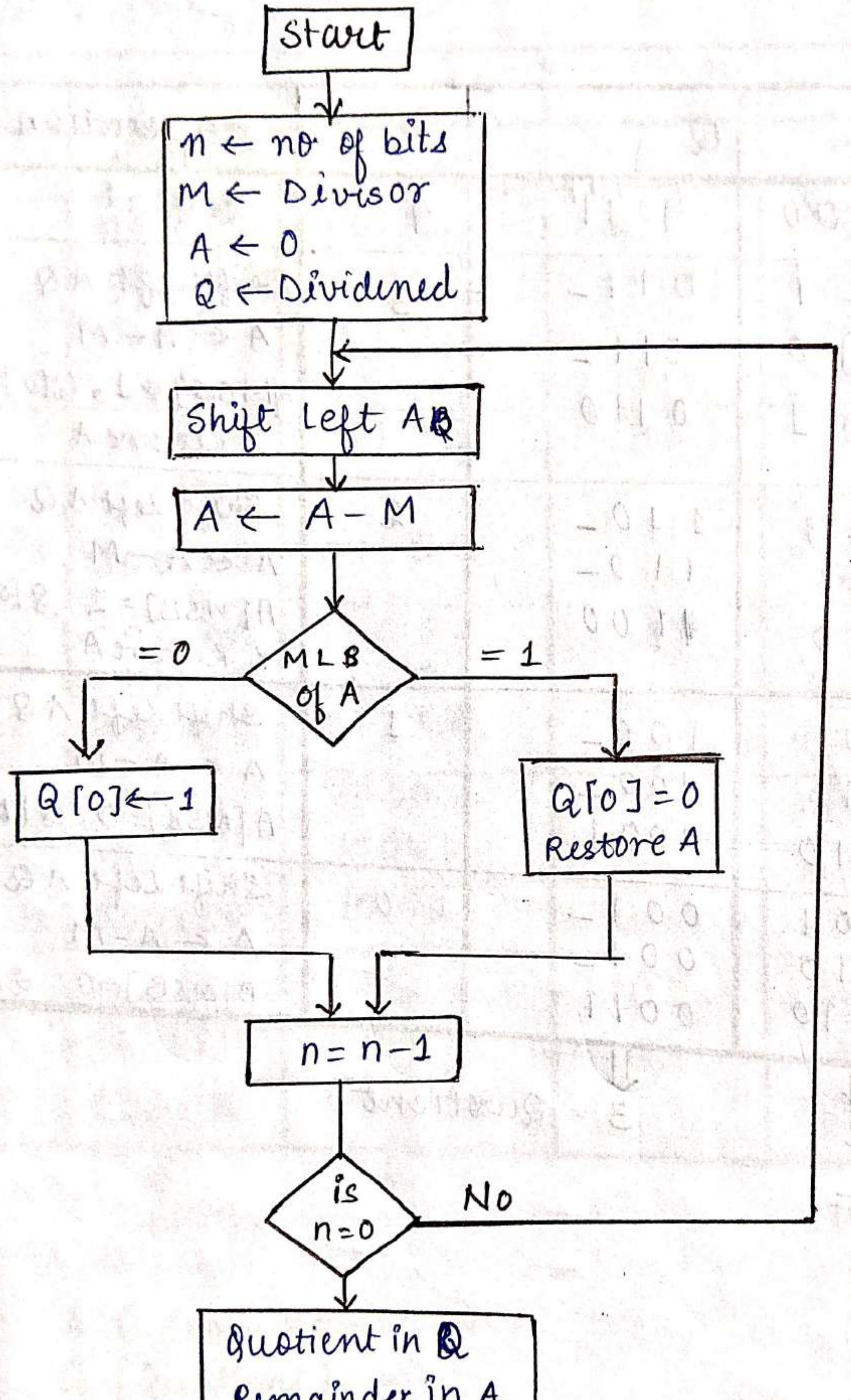
Above logical circuit shows the arrangement that impements the restoring division algorithm.

- An n-bit positive is loaded into register m and n-bit positive dividend is loaded into register Q at the start of operation.
- Register A is set to

  Dividened [Q] = n-bit

  Divisor [M] = n+1 bit

  A = n+1 [initially reset]


Start
n ← no of bits
M ← Divisor
A ← 0.
Q ← Dividened
Shift Left AQ
A ← A − M
MLB of A
= 0
= 1
Q[0] ← 1
Q[0] = 0
Restore A
n = n − 1
is n = 0
No
Quotient in Q
Remainder in A
STOP

Q) Divide
11 by 3

M	A	Q	n	operation
00011	00000	1011	4	Initial

	00001
	11110
	00001
		011_
		011_
		0110
			3	Shift left A Q
				A ← A − M
				A[MSB] = 1, Q[0] = 0
				& Restore A

	00010
	11111
	00010
		110_
		110_
		1100
			2	Shift left A Q
				A ← A − M
				A[MSB] = 1, Q[0] = 0
				& Restore A

	00101
	00010
	00010
		100_
		100_
		1001
			1	Shift left A Q
				A ← A − M
				A[MSB] = 0, Q[0] = 1

	00101
	00010
	00010
		001_
		001_
		0011
			0	Shift left A Q
				A ← A − M
				A[MSB] = 0 Q[0] = 1

$\Downarrow$ Remainder = 2

$\Downarrow$ 3 = Quotient

Q) Divide 19 by 5

M	A	Q	n	Operation
-M = 111011				
000101	000000	10011	5	Initial
	000001 111100 000001	0011_ 0011 00110	4	Shift left AQ $A \leftarrow A - M$ A[MSB] = 1, Q[0] = 0 and Restore A
	000010 111101 000010	0110_ 0110_ 01100	3.	Shift left AQ $A \leftarrow A - M$ A[MSB] = 1, Q[0] ← 0 Restore A
	000100 111111 000100	1100_ 1100_ 11000	2	Shift left AQ $A \leftarrow A - M$ A[MSB] = 1, Q[0] ← 0 Restore A
	001001 000100 000100	1000_ 1000_ 10001	1	Shift left AQ $A \leftarrow A - M$ A[MSB] = 0, Q[0] = 1
	001001 000100 000100	0001_ 0001_ 00011	0	Shift left AQ. $A \leftarrow A - M$ A[MSB] = 0, Q[0] ← 1

$\Downarrow$ 4 $\qquad$ $\Downarrow$ 3

A (Remainder) = 4
Q (Quotient) = 3

ide
55 by 13

M =1110011	A	Q	n	operation
001101	0000000	110111	6	Initial

	0000001
	1110100
	0000001
		10111 _
		10111 _
		101110
			5	Shift Left AQ.
				A ← A − M
				A[MSB] = 1 Q[0] = 0
				Restore A

	0000010
	1110110
	0000011
		01110 _
		01110 _
		011100
			4	shift Left AQ
				A ← A − M
				A[MSB] = 1, Q[0] = 0
				Restore A

	0000110
	1111001
	0000110
		11100 _
		11100 _
		111000
			3	shift Left AQ.
				A ← A − M
				A[MSB] = 1, Q[0] = 0
				& Restore A

	0001101
	0000000
	0000000
		11000 _
		11000 _
		110001
			2	shift Left AQ.
				A ← A − M
				A[MSB] = 0, Q[0] = 1

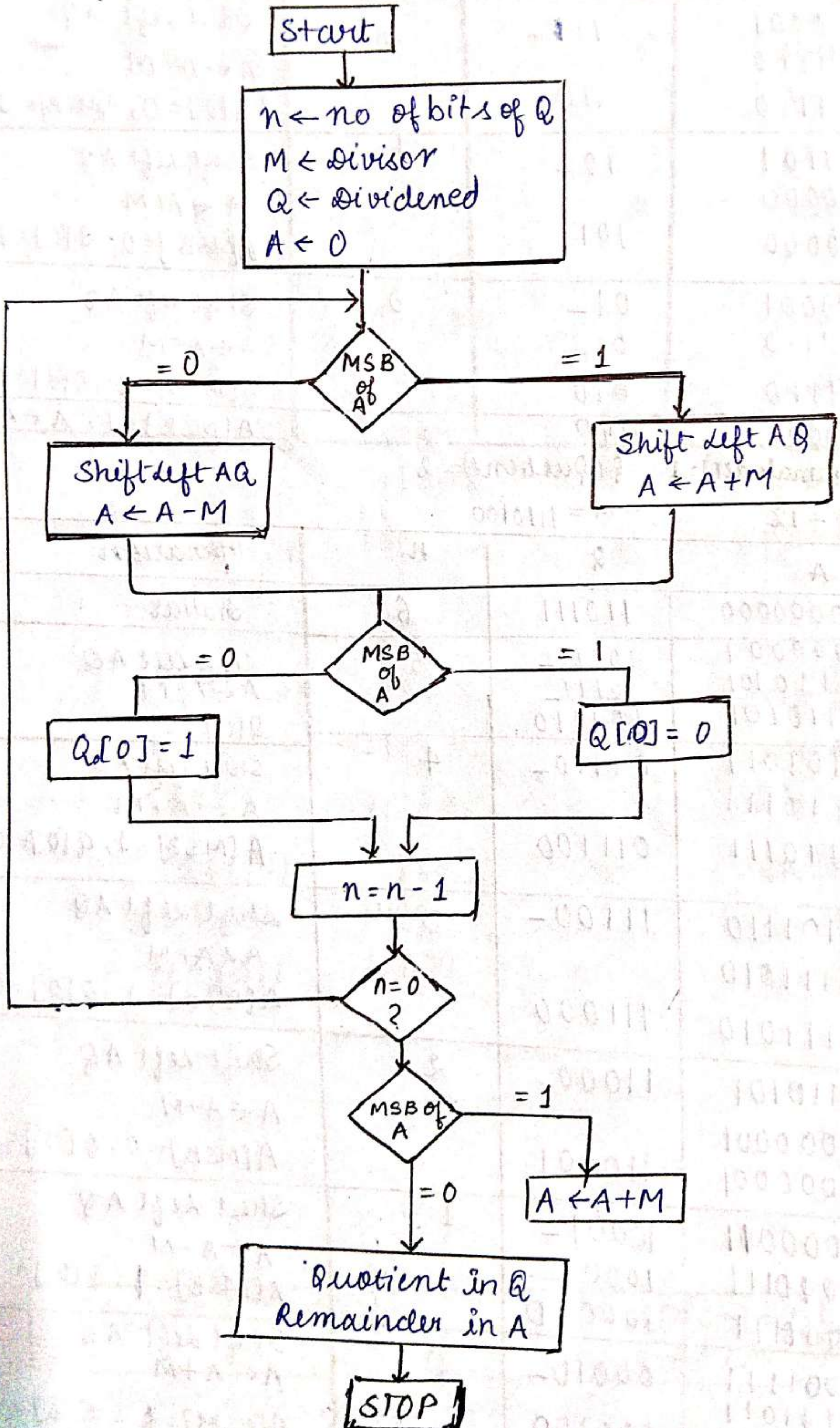
	0000001
	1110100
	0000001
		10001 _
		10001 _
		100010
			1.	shift Left AQ
				A ← A − M
				A[MSB] = 1, Q[0] = 0
				Restore A

	0000011
	1110110
	0000011
		00010 _
		00010 _
		000100
			0	shift Left AQ
				A ← A − M
				A[MSB] = 1, Q[0] = 0
				Restore A

Remainder(A) = 3 4 = Quotient (Q)

Non Restoring

Flowchart of unsigned integer division by using Non restoring method



) Divide 7 by 3 −M = 1101

M	A	Q	n	Operation
0011	0000	111	3	Initial

	0001
	1110
	1110
		11_
		
		110
			2	Shift left AQ
				A←A−M
				Q[0] = 0, A[MSB] = 1

	1101
	0000
	0000
		10_
		
		101
			1	Shift left AQ
				A←A+M
				A[MSB] = 0, Q[0] = 1

	0001
	1110
	1110
		01_
		01_
		010
			0	Shift left AQ
				A←A−M
				A[MSB] = 1, Q[0] = 0

	0001	010		A[MSB] = 1, A←A+M

A(Remainder) = 1 Q(Quotient) = 2

Q Divide 55 ÷ 12 −M = 1110100

M	A	Q	n	Operation
0001100	0000000	110111	6	Initial

	000001
	1110101
	1110101
		10111_
		10111_
		101110
			5	Shift left AQ
				A←A−M
				Q[0] = 0

	1101011
	1110111
	1110111
		01110_
		
		011100
			4	Shift left AQ
				A←A+M
				A[MSB] = 1, Q[0] = 0

	1101110
	1111010
	1111010
		11100_
		
		111000
			3	Shift left AQ
				A←A+M
				A[MSB] = 1, Q[0] = 0

	1110101
	0000001
	0000001
		11000_
		
		110001
			2	Shift left AQ
				A←A+M
				A[MSB] = 0, Q[0] = 1

	0000001
	1110111
	1110111
		10001_
		10001_
		100010
			1.	Shift left AQ
				A←A−M
				A[MSB] = 1, Q[0] = 0

	1101111
	1111011
	1111011
		00010_
		
		000100
			0	Shift left AQ
				A←A+M
				A[MSB] = 1, Q[0] = 0

0000111	000100		A[MSB] = 1
			A ← A+M

7 4

Q=4 R =

Floating point representation

We can represent the floating point numbers in the form of $\pm S \times B^{\pm E}$

where, S = significant or Mantissa

E = Exponent

B = Is the base (for binary B = 2)

I = Sign bit

→ The base (binary number), it is same for all numbers and need not be stored.

- It is assumed that radix point is to the right of the left most or most significant bit of the significants.
- There is one bit to the left of the radix point.

- Any floating point number can be represented in many ways-

~~a)~~ 0.110×2^5

b) 110×2^2

c) 0.0110×2^6

IEEE754 format :-

1) Single precision (32 bit)

2) Double precision (64 bit)

IEEE 754 format	Sign bit	Biased Exponent Bias	Bias	Fraction bits
single bit precision (32 bit)	1	8	127	23 bits
Double bit precision (64 bit)	1	11	1023	52 bits

Sign bit	Exponent bits	fraction bits or mantissa

To simplify the operation on floating point number they are non-zero.

- A normalised number is one in which the most significant digit of the significant is non zero.
- For Base-2 representation a normalised number is therefore one in its MSB of the significant is one. Therefore a normalised non-zero no. is in the form of $\pm 1.bbbb\ldots b \times 2^{\pm E}$ where b is either binary digit (one or zero).
- Because MSB is always one it is unnecessary to store this bit rather it is implicit.
- If the given no is not normalize the no. may be normalize by shifting the radix point to the right of the left most one bit and adusting the exponent accordingly.

<u>Biasing</u>:-

- Exponent is stored in the biased representation.
- In single precision 8 bit field produces the number. ($2^8 = 256$ (0 – 255)
- with the bias of 127 the truth exponent values are in the range of –127 to +128.
- In double precision is 11 bit field produces the numbers from (0–2047)
- with the bias 1023 the truth exponent values in the range: –1023 TO +1024.

<u>Working</u>

- Normalization
- Storage
- Interpretation

Q) Represent the following no. by using single precision & double precision. $(-0.0000010001100110 1)_2$

Sol) 1) <u>Normalization</u>

$1.0001100110 1 \times 2^{-6}$

2) <u>Storage</u> (<u>Single precision</u>):-

Bias exponent $(\bar{E}) = E + \text{Bias}$
$= -6 + 127 = 121$

= 121
=0111001

single precision

−	0111001	00011001101000000000000
sign bit	Exponent($\bar{E}$) ← 8bit →	(Mantissa) ← 23 bit →

2) Double Precision

Biased Exponent ($\bar{E}$) = E + Bias
= −6 + 1023
= 1017
= 01111111001

−	01111111001	0001100110 1 000
sign bit	← → Exponent($\bar{E}$) 11 bit	← 52 bits →

3) Represent $(-1234.125)_{10}$ into single precision and double precision.

10011010010.001

.125 × 2 = 0.250
.250 × 2 = 0.500
.500 × 2 = 1.000

1) Normalization:-

$1.0011010010001 \times 2^{10}$

2) single precision:-

Bias Exponent ($\bar{E}$) = E + Bias
= 10 + 127 = 137
=

+		0011010010001 0
sign bit	← 8bit → Exponent($\bar{E}$)	(Mantissa) ← 23bit →

Logic micro-operations

Logic micro-operations specify binary operation string of bit stored in a register

These operations consider each bit of the register separately & treat them as a binary variable.

For Ex :- ① AND (^) ⑤ NAND
② OR (V) ⑥ XOR
③ NOT (–) ⑦ XNOR
④ NOR

Ex :- ①

R1	0	1	0	1	0	0	0	1
R2	1	0	1	0	0	0	1	1
R3	0	0	0	0	0	0	0	1

R1 AND R2

$R_3 \leftarrow R_1 \wedge R_2$

Shift micro operations

- Shift micro operations are used for serial ~~to~~ transfer of the data.
- Types of shift operations

① Logical shift — Shift Left / Shift Right

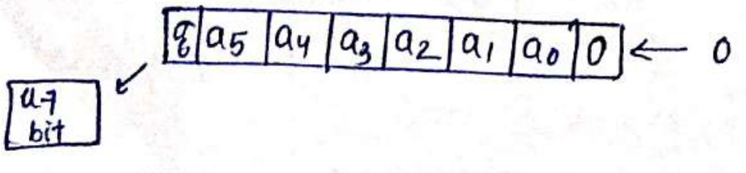
② Circular shift or Rotate — Circular shift Left / Circular shift Right

③ Arithmetic shift — Shift Left / Shift Right

1) Logical shift

A =	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

a) Left shift



Shift Right

0	a_0	a_1	a_2	a_3	a_4	a_5	a_6

→ a_7 bit

3) <u>Arithmetic shift</u>

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

a) <u>Shift Left.</u>

a_6	a_5	a_4	a_3	a_2	a_1	a_0	0

a_7 lost

b) <u>Shift right</u>

a_7	a_7	a_6	a_5	a_4	a_3	a_2	a_1

→ a_0 lost

<u>Circular Shift</u>

In the circular shift the circulation of the bit of the register around the two ends without the loss of information

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

• <u>Circular shift left.</u>

a_6	a_5	a_4	a_3	a_2	a_1	a_0	a_7

• <u>Circular Right</u>

a_0	a_7	a_6	a_5	a_4	a_3	a_2	a_1

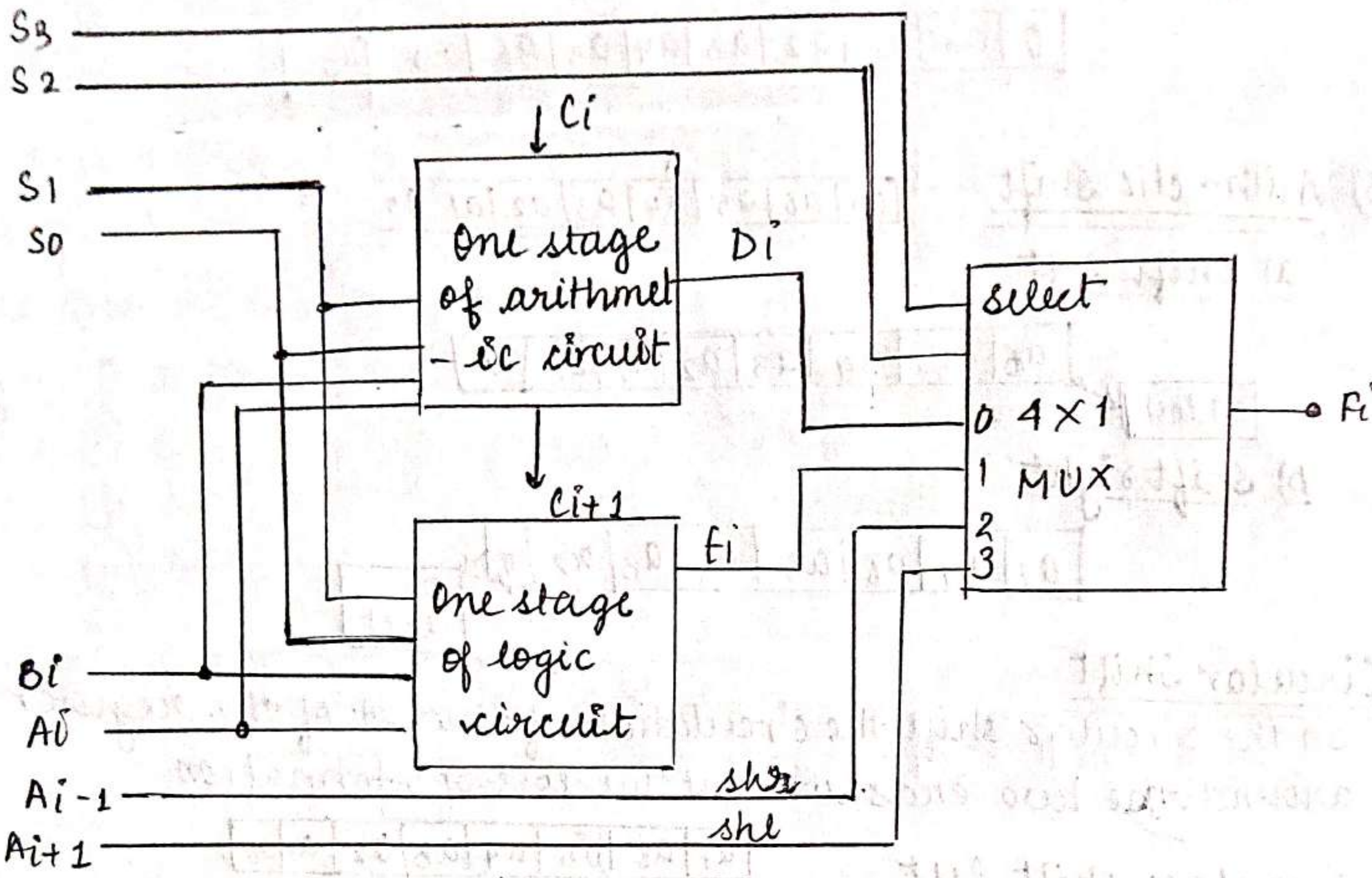
~~a_0~~

ALU DESIGN

CPU contains CU, ALU, registers. ALU is responsible for arithmetic & logical operations basically ALU is a digital circuit that performs arithmetic operations like addition, subtraction, multiplication, division and logical operations like AND, OR, NOT

Types of ALU
1) Combinational ALU
2) Sequential ALU

<u>Arithmetic and logic shift unit:-</u> The ALU is a combinational circuit is that the entire register transfer operation from the source register through the ALU and into the destination register can be performed during one clock pulse period.



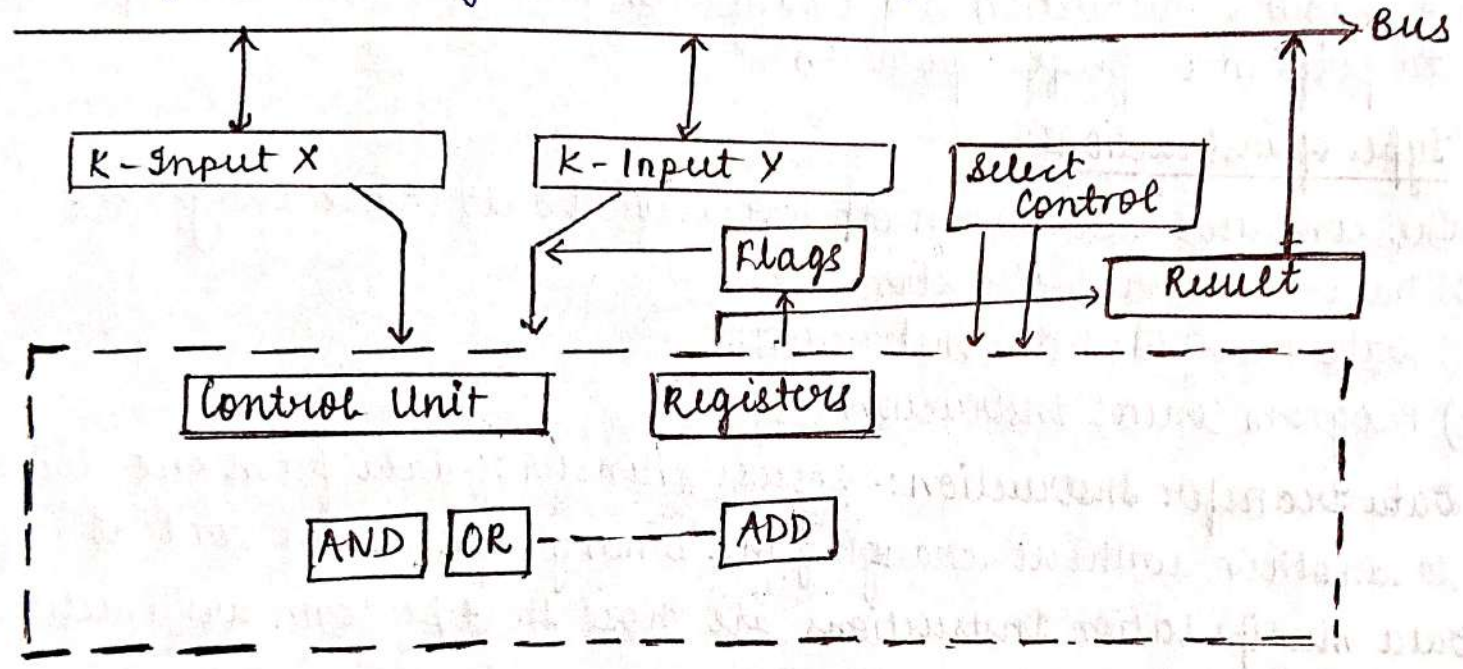
One stage of arithmetic logic shift unit

Operation select

S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A+1$	Increment A
0	0	0	1	0	$F = A+B$	Addition
0	0	0	1	1	$F = A+B+1$	Add with carry
0	0	1	0	0	$F = A+\bar{B}$	subtract with borrow
0	0	1	0	1	$F = A+\bar{B}+1$	Subtraction
0	0	1	1	0	$F = A-1$	~~Dt~~ Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	X	$F = A \wedge B$	AND
0	1	0	1	X	$F = A \vee B$	OR
0	1	1	0	X	$F = A \oplus B$	XOR
0	1	1	1	X	$F = \bar{A}$	complement A
1	0	X	X	X	$F = \text{shr } A$	Shift right A into F
1	1	X	X	X	$F = \text{shl } A$	shift left A into F

Sequential Logic Circuit based ALU:-

Two registers X and Y stores data on operands.
Select/control selects the appropriate arithmetic or logic operation which is performed over the data stored on register X and Y.
After execution of operation, the result will be stored in result register.
After execution of operation flags may ʟ set such as carry, zero result, positive or negative result, overflow, division by zero etc.



Bus
K-Input X
K-Input Y
Select control
Flags
Result
Control Unit
Registers
AND
OR
ADD