UNIT-2

PRIYANKA GOEL
DEPT OF CSE

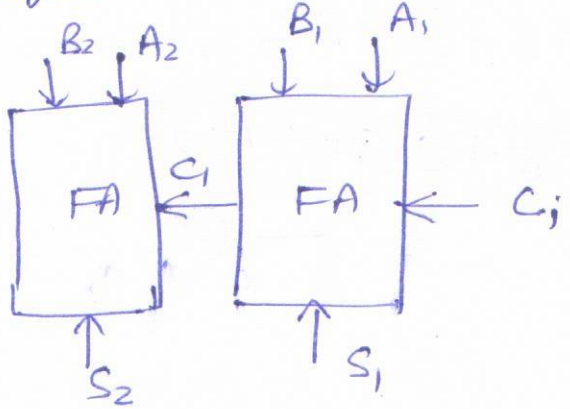
Carry Look Ahead Adders

* CPU performs arithmetic operations such as addition, subtraction, division & multiplication
* Among these, addition & subtraction are basic operations & multiplication and division are repeated addition & subtraction respectively
* Adder ckts are used to perform these basic operations
* Some adder ckts are — HA, FA, Ripple Carry Adder and CLA

About CLA

* fastest adder circuit
* reduces propagation delay (during addition)
* uses more complex h/w ckt
* designing by transforming ripple carry Adder ckt such that carry logic of adder is changed to two-level logic

4-bit Ripple Carry Adder



* To produce final steady state results, carry must propagate through all states.
* This increases carry propagation delay of ckt
* Propagation delay of adder is calculated as — "propagation delay of each gate × no. of stages in ckt"
* For computation of a large no. of bits, more stages have to be added

TT of Look Ahead Adder

A	B	C_i	Carry
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

→ C_i is 1 → Carry propagated through stage

$A, B \rightarrow X$

→ C_i is 1

→ A, B are 1 — C_i is dont care

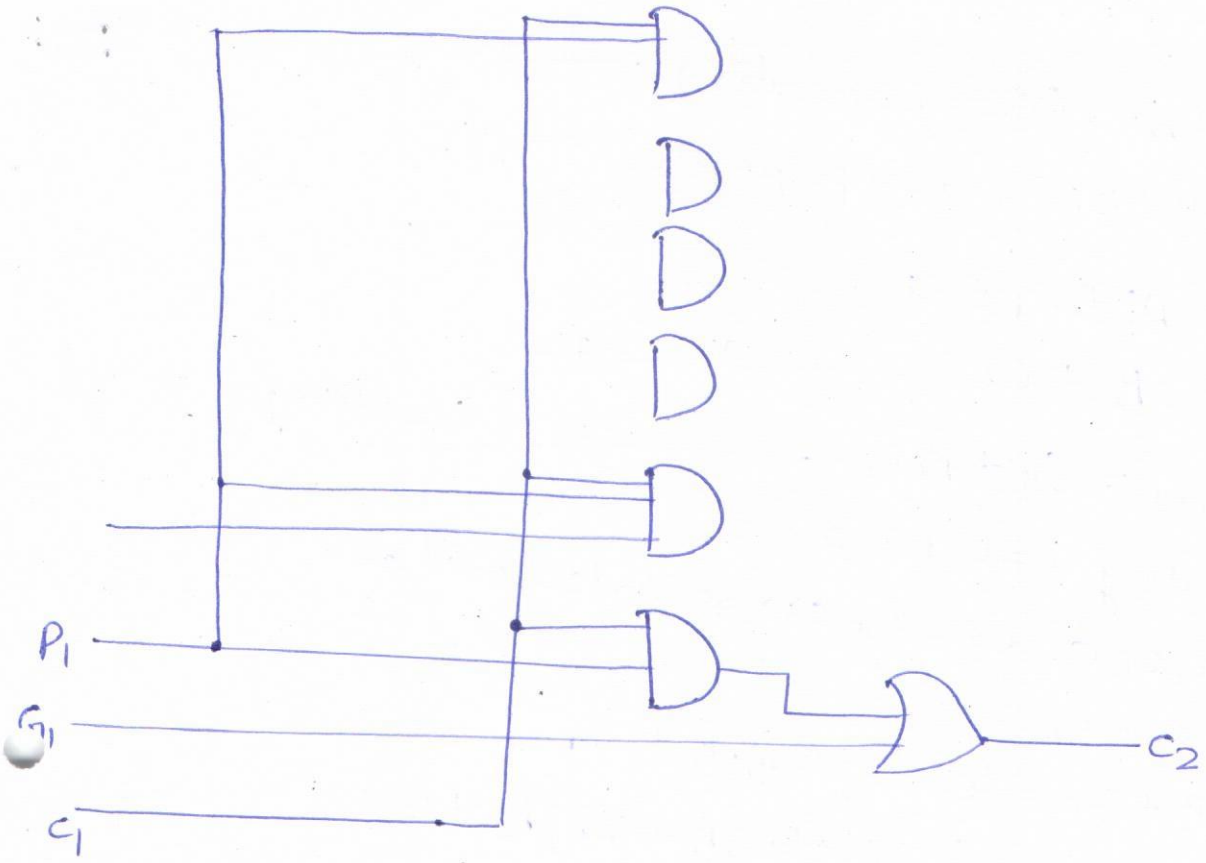
→ A, B are 1 ↳ known as carry generation stage

$$C_0 = A \cdot B + (A \oplus B) C_0$$

$$\boxed{C_{i+1} = A_i B_i + C_i (A_i \oplus B_i)} \quad \text{— (1)}$$

Put $i = 0, 1, 2, 3$

$$C_1 = C_0 (A_0 \oplus B_0) + A_0 B_0$$
$$C_2 = C_1 (A_1 \oplus B_1) + A_1 B_1$$
$$C_3 = C_2 (A_2 \oplus B_2) + A_2 B_2$$
$$C_4 = C_3 (A_3 \oplus B_3) + A_3 B_3$$



Eqn shows that carry-in of any stage full adder depends only on -

a) Bits being added in previous stages

b) Carry bit which was provided in the beginning

Advantages

* It generates carry in for each full adder simultaneously
* It ~~generates pr~~ reduces propagation delay
* Fastest adder

Des advantages

* Complex hardware
* Costlier
* gets more complicated as no. of bits are increased
*

$G_i = A_i B_i$ (Carry Generator) —— (2)

$P_i = A_i \oplus B_i$ (Carry propagator) —— (3)

$Sum = A_i \oplus B_i \oplus C_i$

$= P_i \oplus C_i$ —————— (4)

$Carry_{i+1} = G_i + P_i C_i$ —————— (5) [Substituting in eqn (1)]

Substituting $i = 0, 1, 2, 3$

Carry bits C_1, C_2, C_3 & C_4 can be calculated as—

$C_1 = C_0 P_0 + G_0$

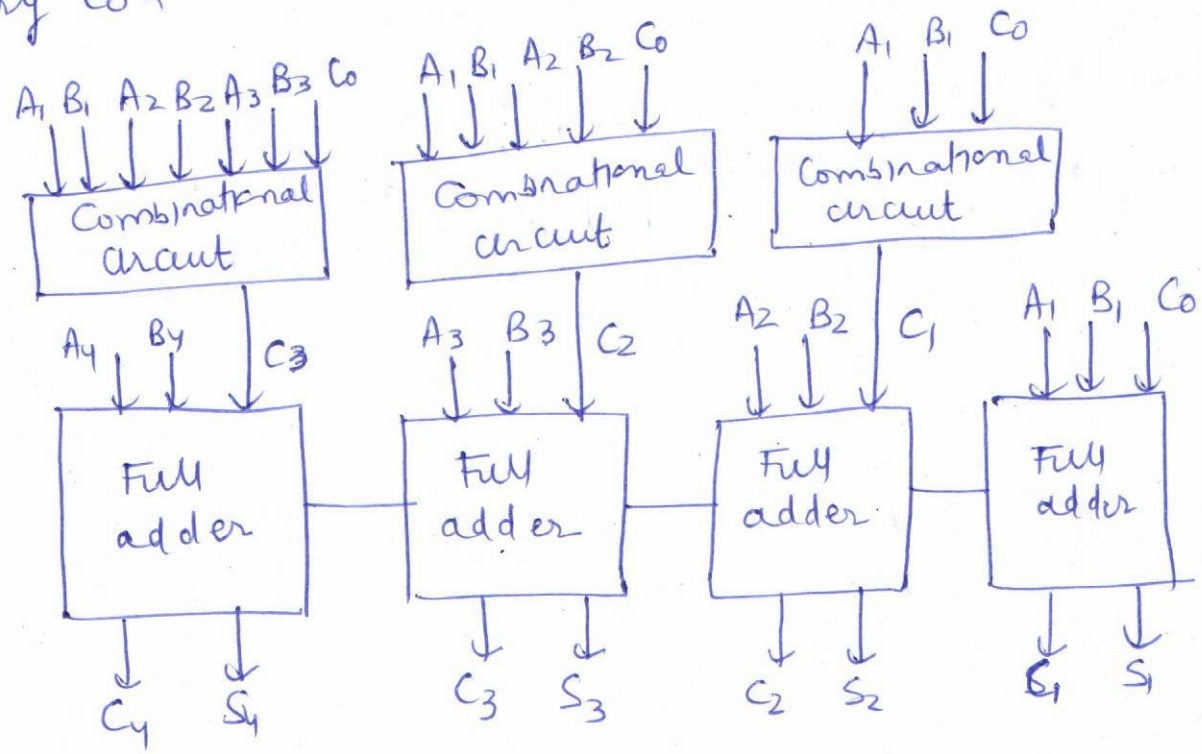
$C_2 = C_1 P_1 + G_1 = (C_0 P_0 + G_0) \cdot P_1 + G_1$

$C_3 = C_2 \cdot P_2 + G_2 = [(C_0 P_0 + G_0) P_1 + G_1] P_2 + G_2$

$C_4 = C_3 P_3 + G_3$

$= ((C_0 P_0 + G_0) P_1 + G_1] P_2) P_3 + G_3$

Thus it can be observed that C_{i+1} depends only on the carry C_0 & not on intermediate carry bits



8. Addition and Subtraction with Signed Magnitude Data

* The magnitude of the two numbers can be designated by A and B.
* When the signed numbers are added or subtracted, there are eight different conditions to consider, depending on the sign of numbers and the operation performed.

operation	Add Magnitude	Subtract Magnitudes		
		When A > B	When A < B	When A = B
(+A) + (+B)	+(A+B)			
(+A) + (−B)		+(A−B)	−(B−A)	+(A−B)
(−A) + (+B)		−(A−B)	+(B−A)	+(A−B)
(−A) + (−B)	−(A+B)			
(+A) − (+B)		+(A−B)	−(B−A)	+(A−B)
(+A) − (−B)	+(A+B)			
(−A) − (+B)	−(A+B)			
(−A) − (−B)		−(A−B)	+(B−A)	+(A−B)

showing actual operation to be performed with the magnitude of numbers

→ last column is needed to prevent a −ve 0

Addition Algorithm :* When the signs of A and B are identical, add the two magnitudes and attach the sign of A to the result.

* When the signs of A and B are different, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B.
* If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

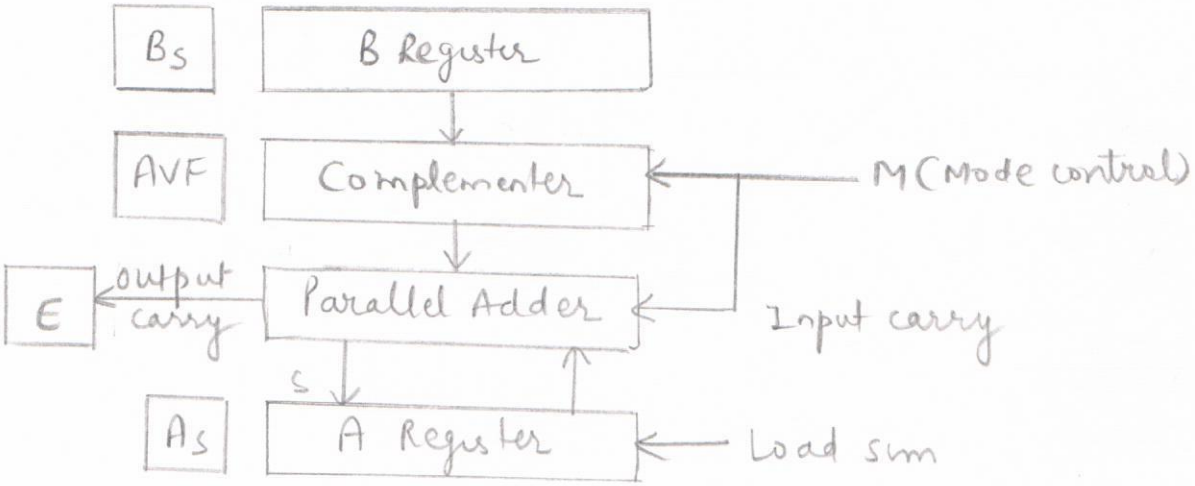
e.g.: (+5) + (+7) = +(5+7) = +(12)
(+5) + (-2) = +(5-2) = +(3)
(+5) + (-7) = -(7-5) = -(2)
(+5) + (-5) = +(5-5) = +0
(-5) + (-2) = -(5+2) = -(7)

Subtraction Algorithm :- * When the signs of A and B are identical, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as ~~that of larger number~~ A if A > B or the complement of sign of A if A < B.
* When the signs of A and B are different, add the two magnitudes and attach the sign of A to the result.
* If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

e.g. (+5) - (+7) = -(7-5) = -2
(-5) - (-7) = +(7-5) = 2
(+5) - (-7) = +(5+7) = +(12)
(-5) - (+7) = -(5+7) = -(12)
(+5) - (+5) = +(5-5) = +0
(-5) - (-5) = +(5-5) = +0

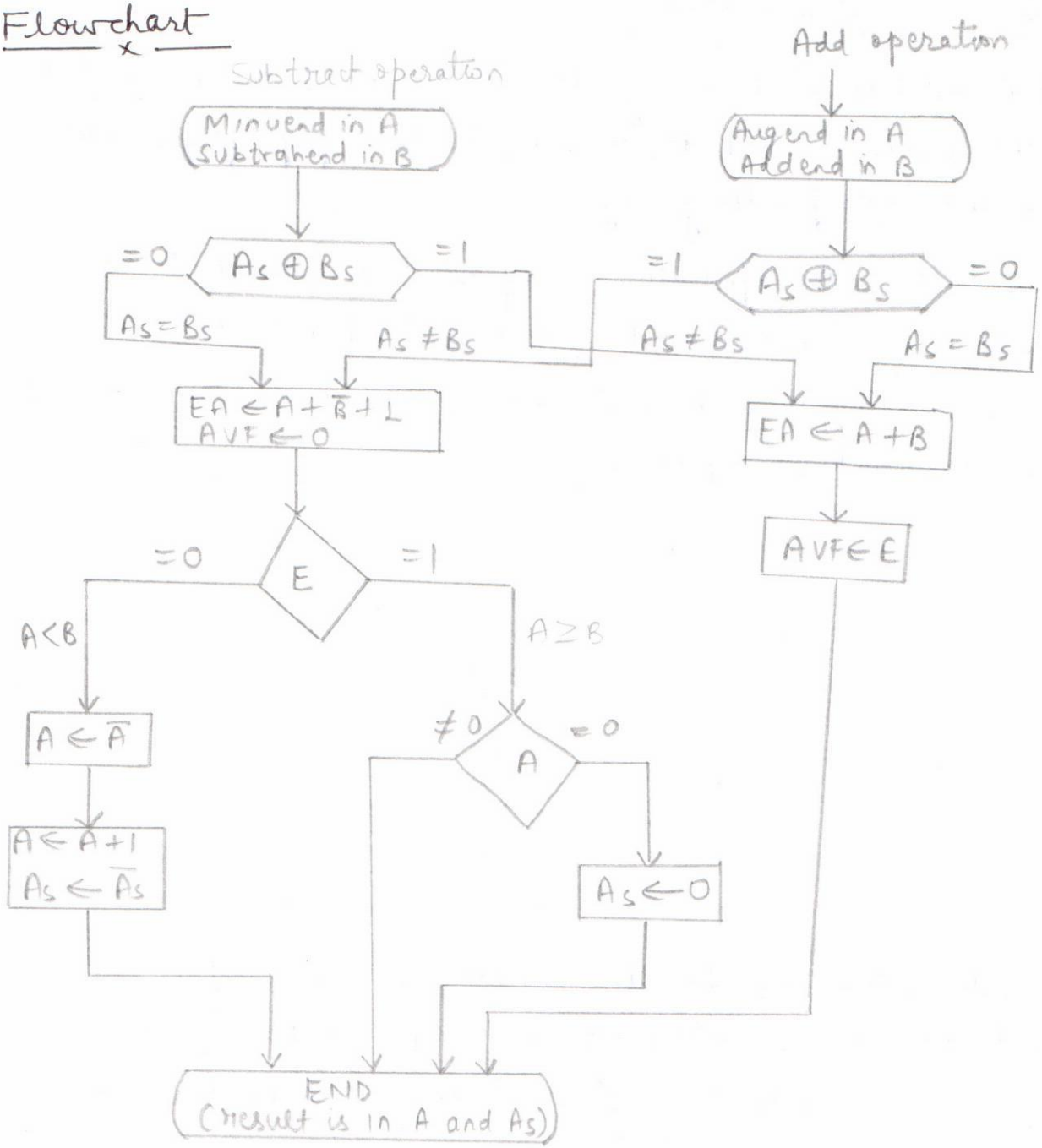
8.1. Hardware Implementation

* Let A and B be two registers that hold the magnitudes of the numbers and A_s and B_s be two flip-flops that hold the corresponding signs.
* The result of the operation may be transferred to a third register — if the result is transferred into A and A_s then a saving is achieved. Thus, A and A_s together form an accumulator register.



* The mode control M decides whether value of B is given to parallel adder or its complement.
* When M=0, output of B is transferred to the adder and the input carry is 0 (∵ The M signal is also applied to the input carry of the adder).
  In this case, output of adder is equal to the sum A+B.
* When M=1, 1's complement of B is applied to the adder, the input carry is 1 and the output is A+B+1.

Flowchart



eg. 1 (+5) + (+7)

A = 0 1 0 1 $\quad A_S = 0 \quad A_S \oplus B_S = 0$

B = + 0 1 1 1 $\quad B_S = 0$

EA = 1 1 0 0

AVF = 0

Result | A = 1 1 0 0 |
| A_S = 0 |

e.g.2 :- $(-5)+(-7)$

$A = 1\,0\,1\,0\,1$ $\quad A_S = 1$ $\quad A_S \oplus B_S = 0$ (Add)

$B = 0\,1\,1\,1$ $\quad B_S = 1$

$+$

$1\,1\,0\,0$

$E = 0 \Rightarrow AVF = 0$

Result :- A : 1100
$A_S = 1$

e.g.3 :- $(-5)+(+7)$

$A = 0\,1\,0\,1$ $\quad A_S = 1$ $\quad A_S \oplus B_S = 1$ (subtract)

$B = 0\,1\,1\,1$ $\quad B_S = 0$

$\bar{B} = 1\,0\,0\,0$

$\bar{B}+1 = \quad +1$

$1\,0\,0\,1$

$\longrightarrow$

$0\,1\,0\,1$

$1\,0\,0\,1$

A: $1\,1\,1\,0$

$AVF = 0,\ E = 0$

$A < B$

$\bar{A} = 0001$

$\bar{A}+1 = \quad +1$

0010

$A_S = A_S + 1 = 0$

Result :-
A : 0010
$A_S = 0$

e.g.4 :- $(+7)+(-5)$

$A = 0\,1\,1\,1$ $\quad A_S = 0$ $\quad A_S \oplus B_S = 1$ (subtract)

$B = 0\,1\,0\,1$ $\quad B_S = 1$

$\bar{B} = 1\,0\,1\,0$

$\bar{B}+1 = \quad +1$

$1\,0\,1\,1$

$\longrightarrow$

$0\,1\,1\,1$

$1\,0\,1\,1$

$1\,0\,0\,1\,0$

$AVF = 0,\ E = 1 \xrightarrow[A \geq B]{} A \neq 0$

Result :- 0010

e.g.5 : (-5)-(-7)

$A = 1101$ $A_S = 1$

$B = 0111$ $B_S = 1$

$A_S \oplus B_S = 0$ (add)

$$\begin{array}{r} 1101 \\ +\ 0111 \\ \hline 1100 \\ \hline \end{array}$$

Result :-

AVF = 0, E = 0

$A = 0101$ $A_S = 1$

$B = 0111$ $B_S = 1$

$A_S \oplus B_S = 0$ (add)

$\overline{B} = 1000$

$\overline{B}+1 = 1001$

$$\begin{array}{r} 0101 \\ +\ 1001 \\ \hline 1110 \\ \hline \end{array}$$

E = 0, AVF = 0

$A < B$

$\overline{A} = 0001$

$\overline{A}+1 = 0010$

$A_S = A_S + 1 = 0$

Result :- $A = 0010$, $\bar{A}_S = 0$

e.g.6 :- (-5)+(+7)

$A = 0101$ $A_S = 1$

$B = 0111$ $B_S = 0$

$A_S \oplus B_S = 1$

$\overline{B} = 1000$

$\overline{B}+1 = 1001$

e.g.6 :- (−5)−(+7)

A = 0L0L $A_S = L$ $A_S \oplus B_S = L$

B = 0LLL $B_S = 0$

+

LL00

E=0, AVF=0

Result :- LL00
A_S : 1

9. Addition and subtraction with Signed 2's Complement Data

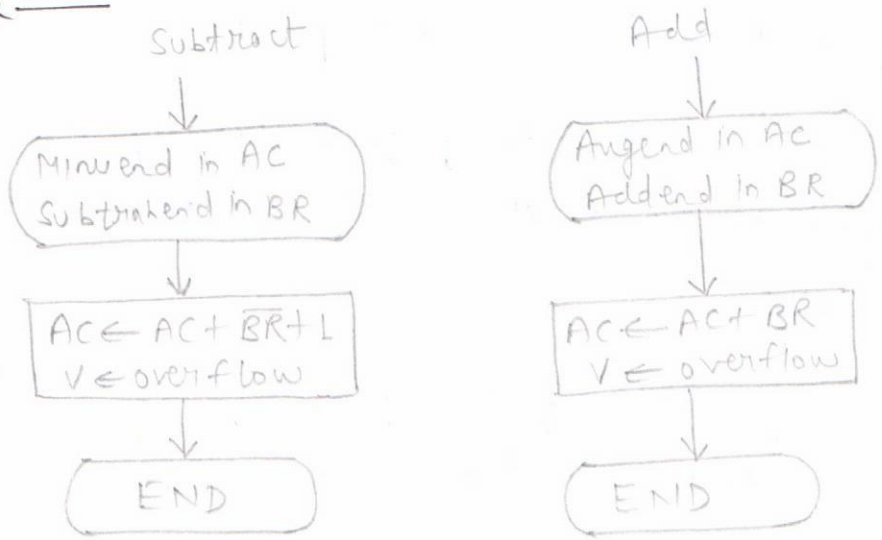
* The addition of two numbers in signed 2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number.
* A carry-out of the sign-bit position is discarded.
* The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.
* When two numbers of n-digits each are added and the sum occupies n+1 digits, an overflow occurs.

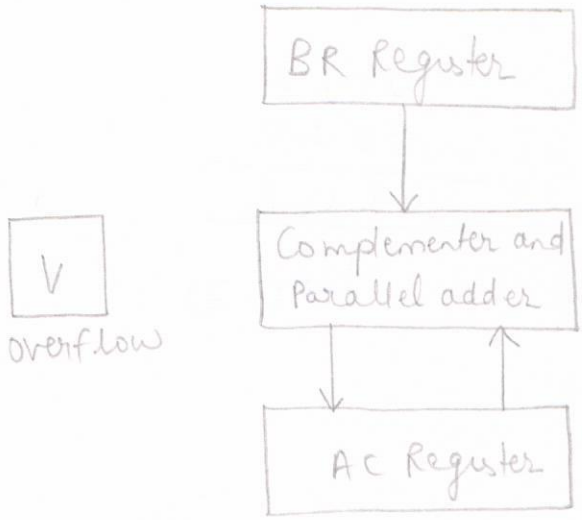
Overflow :- It is a problem in digital computers because the width of registers is finite. A result that contains n+1 bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow & when it occurs, a corresponding flip-flop is set which can then be checked by the user.

An overflow cannot occur after an addition if one number is positive and the other is negative since adding a positive number to a negative number produces a result that is smaller than the larger of two original numbers. An overflow may occur if the two numbers added are both positive or both negative.

* An overflow can be detected by inspecting the last two carries out of the addition.
* When the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1.

Flowchart





Hardware for signed 2's complement addition and subtraction

* Comparing this algorithm with its signed-magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed 2's complement representation.

* For this reason most computers adopt this representation over the more familiar signed-magnitude.

e.g. 1 (−5) + (−7)

A = 0101 B = 0111 1 1
Ā = 1010 B̄ = 1000 1011
+1 +1 1001
1011 1001 10100
discard

Ans:- 0100 or (−12)

14

$(-5)+(2)$

$A = 0101$ $\quad B = 0010$

$\overline{A} = 1010$

$+1$

1011

1011

0010

1101

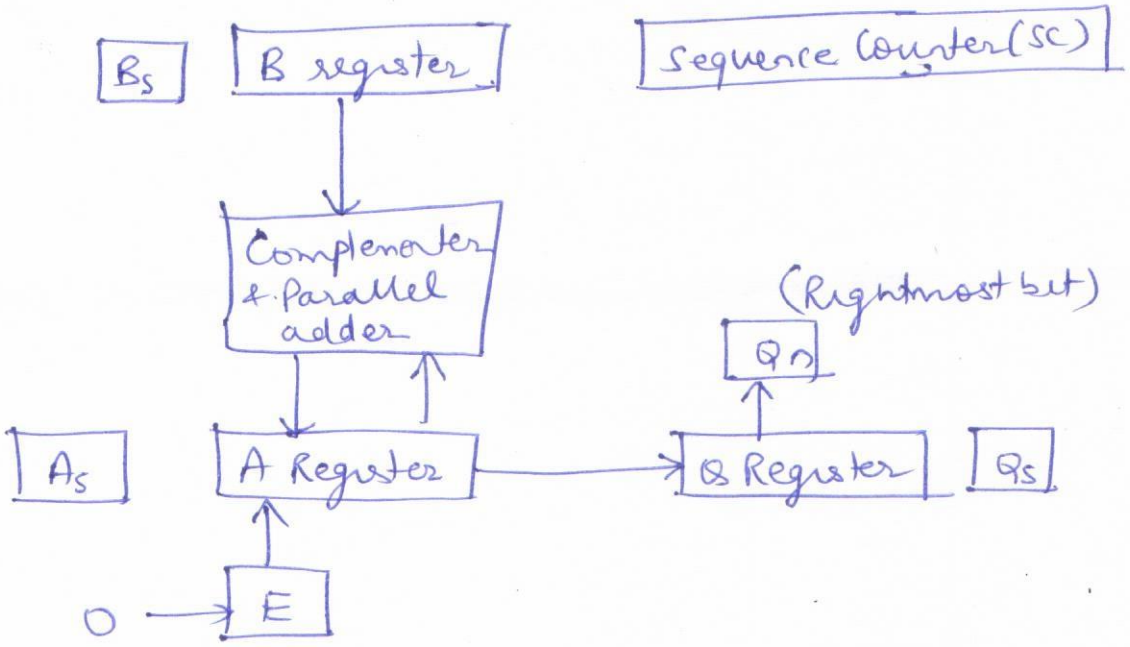
$\hookrightarrow (-3)$

Hardware Implementation for Multiply operation

B ← multiplicand , B_s ← sign

Q ← multiplier Q_s ← sign

* Instead of shift left (as used in manual operation), shift right on partial product will be followed.
* If the multipler bit is 0 then there is no need to copy all the bits (as we do in manual working)



Working

* Successively accumulate partial products & shift it right
* SC ← no. of bits in multiplier
* SC is decremented after forming each partial product
* When SC is 0, the process halts & final product is formed

10. Multiplication with Signed Magnitude Data

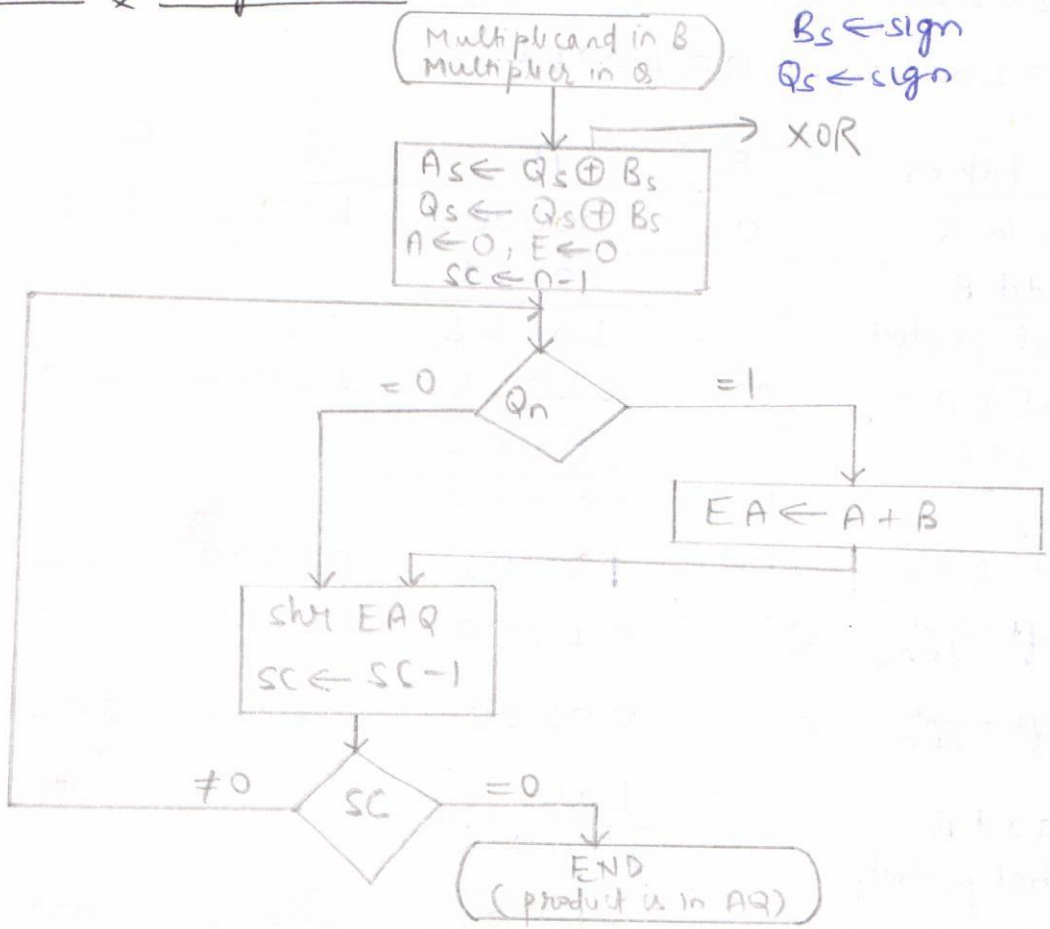
Manually, two numbers (let A and B) are multiplied as–

```
   23          10111     Multiplicand
 × 19        × 10011     Multiplier
 ----        -------
                10111
               10111x     ← one shift left
              00000xx     ← shift left
             00000xxx     ← shift left
addition  +  10111xxxx     ← shift left
 ----        -------
 437         110110101     Product
```

successive shift and add operation

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is positive. If they are unlike, the sign of the product is negative.

10.1. Hardware Algorithm



Working

- Initially multiplicand is in register B and the multiplier in Q.
- The sum of A & B forms a partial product which is transferred to EA register.
* Both partial product and multiplier are shifted to the right.
* The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A and 0 is shifted to E.
* After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right.
* In this manner, the rightmost flip flop in register Q, designated by Q_n will hold the bit of the multiplier, which must be inspected next.

e.g.1 10111 x 10011

B = 10111 Q = 10011

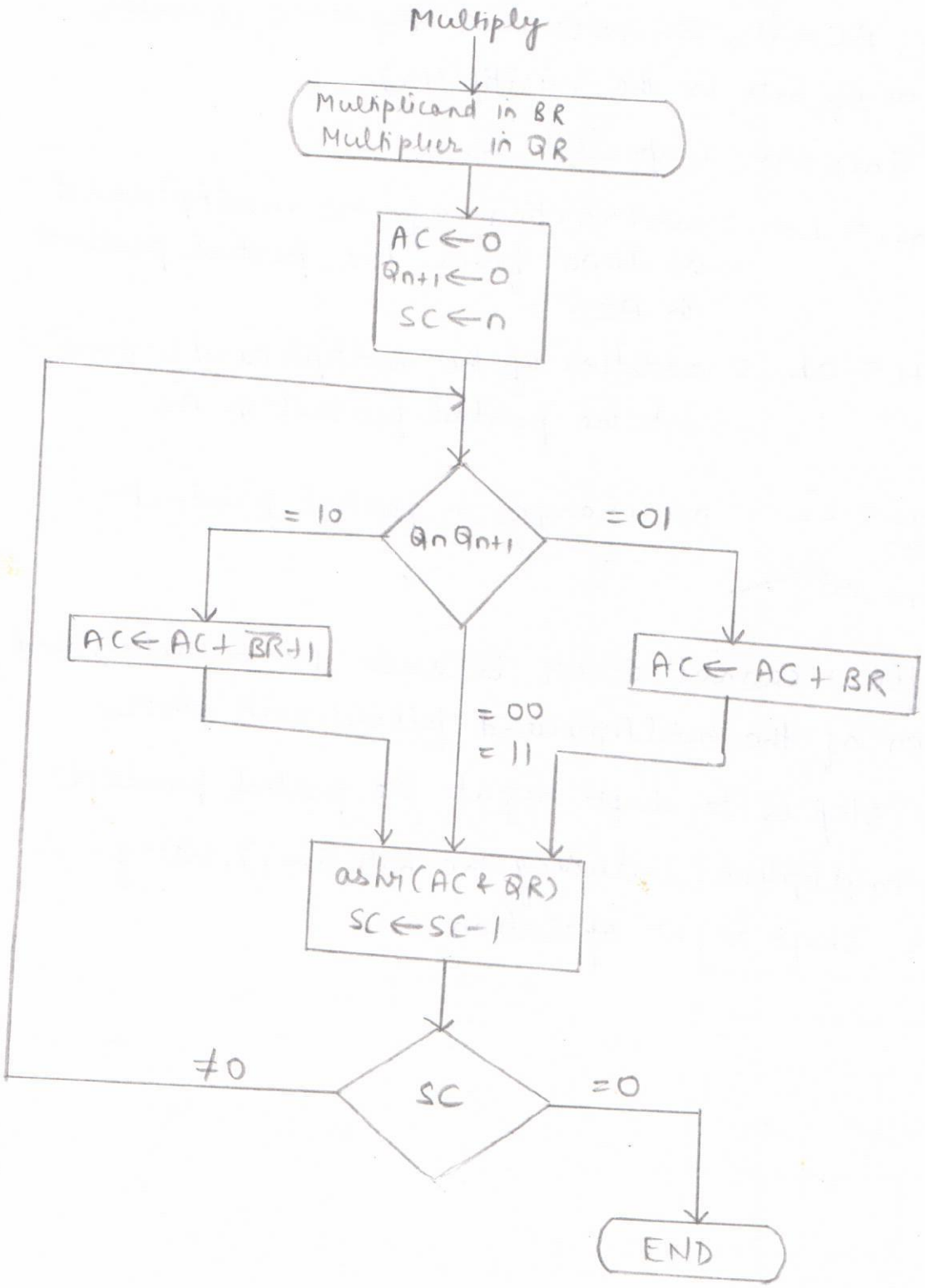
Action taken	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n=1, add B		10111		
first partial product		10111		
Shift right EAQ	0	01011	11001	100
Q_n=1, add B		10111		
	1	00010		
Shift right EAQ	0	00001	01100	011
Q_n=0, shift right EAQ	0	00000	10110	010
Q_n=0, shift right EAQ	0	00100	01011	001
Q_n=1, add B		10111		
fifth partial product		11011		
Shift right EAQ	0	01101	10101	000

e.g.2 (-9) x (-13) = ? → page no. 60

10.2.

Booth Multiplication Algorithm for signed 2's complement representation

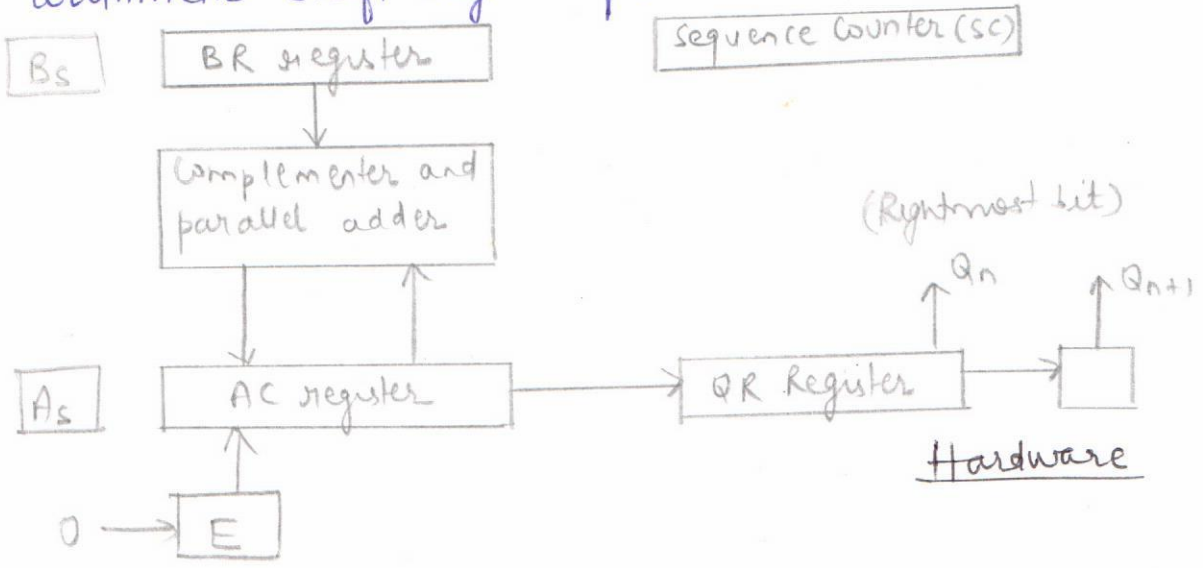
* This algorithm works for positive or negative multipliers in 2's complement representation.



19

Working

* Q_n designates the least significant bit of the multiplier in register QR.
* An extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier.
* Initially, AC = 0, Q_{n+1} = 0 and sequence counter SC = n (no. of bits in the multiplier)
* Q_n and Q_{n+1} are inspected as —
  - → $Q_n Q_{n+1}$ = 10 : subtraction of the multiplicand is done from the partial product in AC.
  - → $Q_n Q_{n+1}$ = 01 : addition of the multiplicand is done, into the partial product in AC.
  - → $Q_n Q_{n+1}$ = 11 : no change in partial product
    or
    $Q_n Q_{n+1}$ = 00
* An overflow cannot occur because the addition and subtraction of the multiplicand follow each other.
* The next step is to shift right the partial product and the multiplier (including the bit Q_{n+1}), using arithmetic shift right operation.



e.g. 1 $(-9) \times (-13) = ?$

BR = ~~10111~~ BR = 01001

$\overline{BR}$ = 10110 + 1

BR (in 2's comp.) : 10111

QR = 01101 $\overline{QR}$ = 10010 + 1 → 10011 (QR in 2's comp)

AC = 0, Q_{n+1} = 0, SC = 101

$Q_n Q_{n+1}$	BR = 10111, $\overline{BR}$ = 01000 + 1, $\overline{BR}$+1 = 01001	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
10	subtract BR	01001			
		01001			
	ashr (AC & QR)	00100	11001	1	100
11	ashr (AC & QR)	00010	01100	1	011
01	add BR	10111			
		11001			
	ashr (AC & QR)	11100	10110	0	010
00	ashr (AC & QR)	11110	01011	0	001
10	subtract BR	01001			
		100111			
	discard ashr (AC & QR)	00011	10101	1	000

→ Answer i.e. $(117)_{10}$

e.g. 2 $(-5) \times (+4) = ?$

BR = 00101 QR = 00100 Q_{n+1} = 0

$\overline{BR}$ = 11010 + 1

A = 0

SC = 101

BR (in 2's comp) → 11011

$Q_n Q_{n+1}$	BR = 11011, $\overline{BR}$ = 00100 + 1, $\overline{BR}$+1 = 00101	AC	QR	Q_{n+1}	SC
	Initial	00000	00100	0	101
00	ashr AC & QR	00000	00010	0	100
00	ashr AC & QR	00000	00001	0	011
10	subtract BR	00101			
		00101			
	ashr (AC & QR)	00010	10000	1	010
01	add BR	11011			
		11101			
	ashr (AC & QR)	11110	11000	0	001
00	ashr (AC & QR)	11111	01100	0	000

$(-9) \times (-13)$

$B = 1\ 0110\ 1$ $\quad Q = 1\ 0110\ 1$ $\quad A_s = 1 \oplus 1 = 0$

$B_s = 1 \oplus 1 = 0$

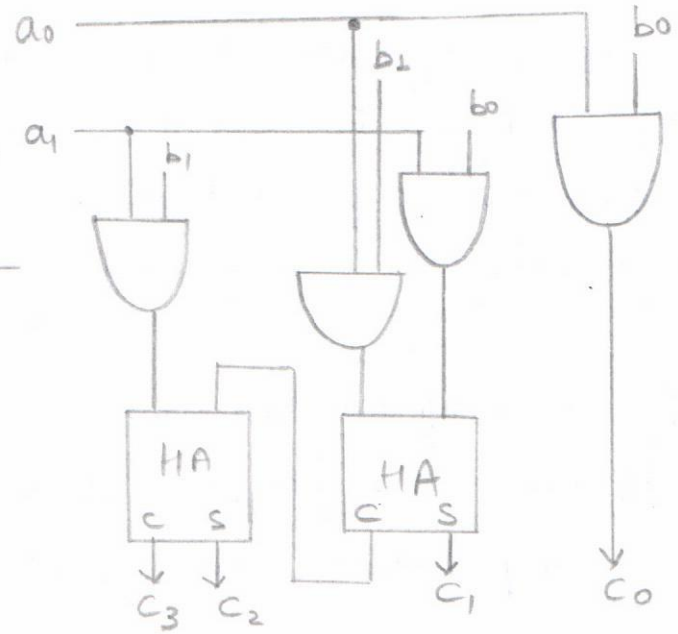
$A = 0,\ E = 0,\ SC = 101$

Multiplicand $B = 01001$	E	A	Q	SC
Multiplier in Q	0	00000	01101	101
$Q_n = 1$ add B	0	01001		
		01001		
shr EAQ	0	00100	10110	100
$Q_n = 0$, shr EAQ	0	00010	01011	011
$Q_n = 1$, add B	0	01001		
		01011		
shr EAQ	0	00101	10101	010
$Q_n = 1$, add B	0	01001		
		01110		
shr EAQ	0	00111	01101	001
$Q_n = 0$, shr EAQ	0	00011	10110	000

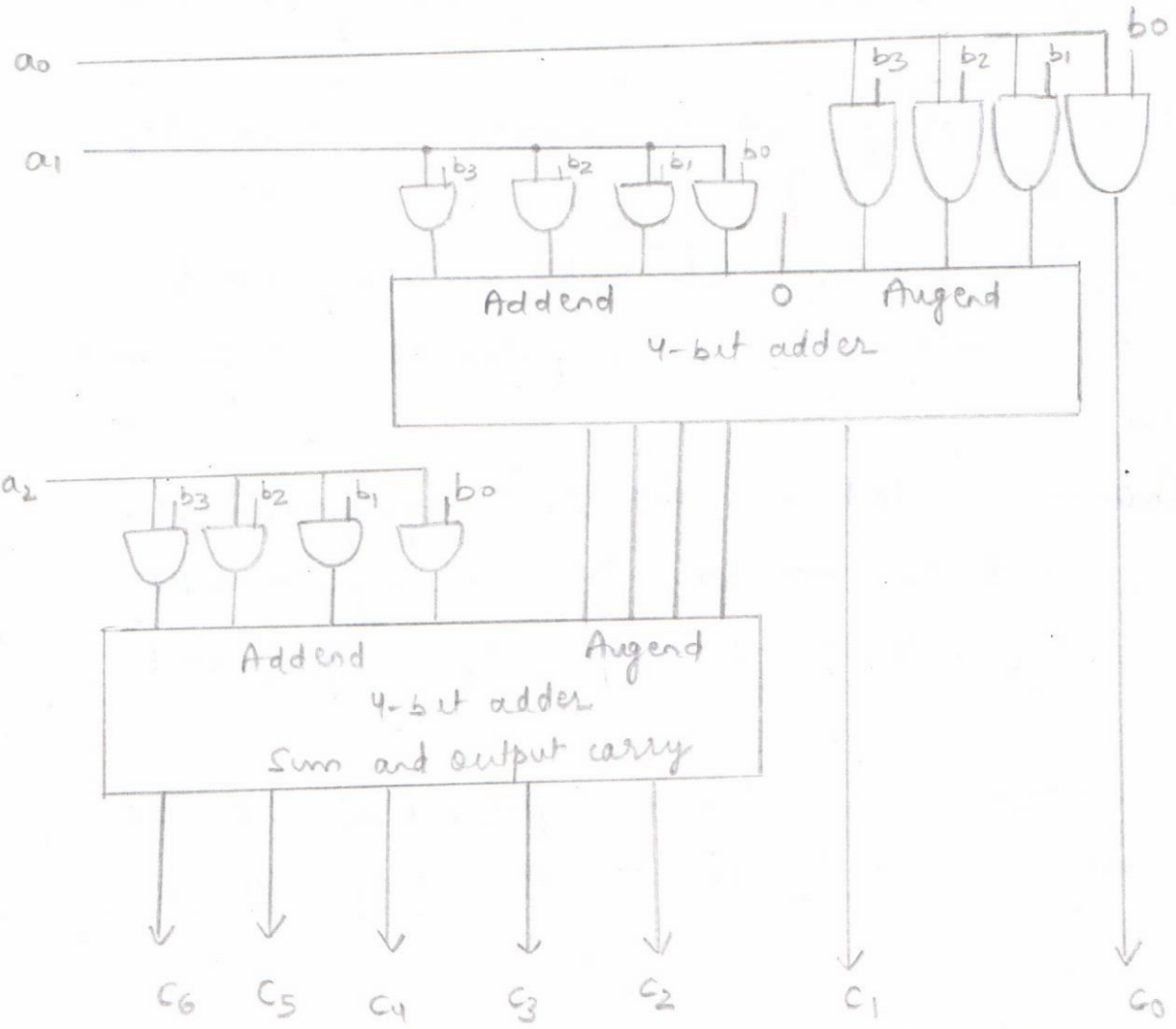
10.3. Array Multiplier

* Consider the multiplication of two 2-bit numbers.
* The multiplicand bits are b_1 and b_0 and the multiplier bits are a_1 and a_0 and the product is $c_3c_2c_1c_0$.
* The first partial product is formed by multiplying a_0 by b_1, b_0.
* The multiplication of two bits such as a_0 and b_0 produces a 1 if both bits are ~~implem~~ 1; otherwise it produces 0.
* This can be implemented with an AND gate.
* Similarly, the second partial product is formed by multiplying a_1 by b_1 b_0 and is shifted one position to the left.
* The two partial products are added with two half-adder circuits.
* Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum.
* It should be noted that the least significant bit of the product doesnot have to go through an adder since it is formed by the outputs of the first AND gate.
* For j multiplier bits and k multiplicand bits, we need $j \times k$ AND gates and $(j-1)k$-bit adders to produce a product of $j+k$ bits.

b_1 b_0
a_1 a_0
a_0b_1 a_0b_0
a_1b_1 a_1b_0
3 c_2 c_1 c_0



2-bit by 2-bit array multiplier



4-bit by 3-bit array multiplier

* The circuit connections can be understood as :-

		b_3	b_2	b_1	b_0	
			a_2	a_1	a_0	
		a_0b_3	a_0b_2	a_0b_1	a_0b_0	
	a_1b_3	a_1b_2	a_1b_1	a_1b_0		
a_2b_3	a_2b_2	a_2b_1	a_2b_0			
c_6	c_5	c_4	c_3	c_2	c_1	c_0

Advantages

* It is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array.

Disadvantage

* It requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.

11. Division algorithm for signed magnitude data

Manual approach:

Given divisor B = 10001
dividend A = 0111000000

```
            011010
       ___________________
 10001 ) 0111000000           Quotient Q
         00000                Dividend = A
         ------               5 bits of A < B, so Q = 0
         011100               6 bits of A ≥ B
          10001
         -------
         0010110              7 bits of remainder ≥ B
           10001
         --------
         00001010             Remainder < B; enter 0 in Q
             00000
         ---------
             10100            Remainder ≥ B
             10001
             ------
             000110           Remainder < B
                000
                ---
                110           final remainder
```

Hardware Implementation

* When division is implemented in a digital computer, it is convenient to change the process slightly.
* Instead of shifting the divisor to the right, the dividend or partial remainder, is shifted to the left.
* Subtraction can be achieved by adding A to the 2's complement of B.

11.1. Divide Overflow

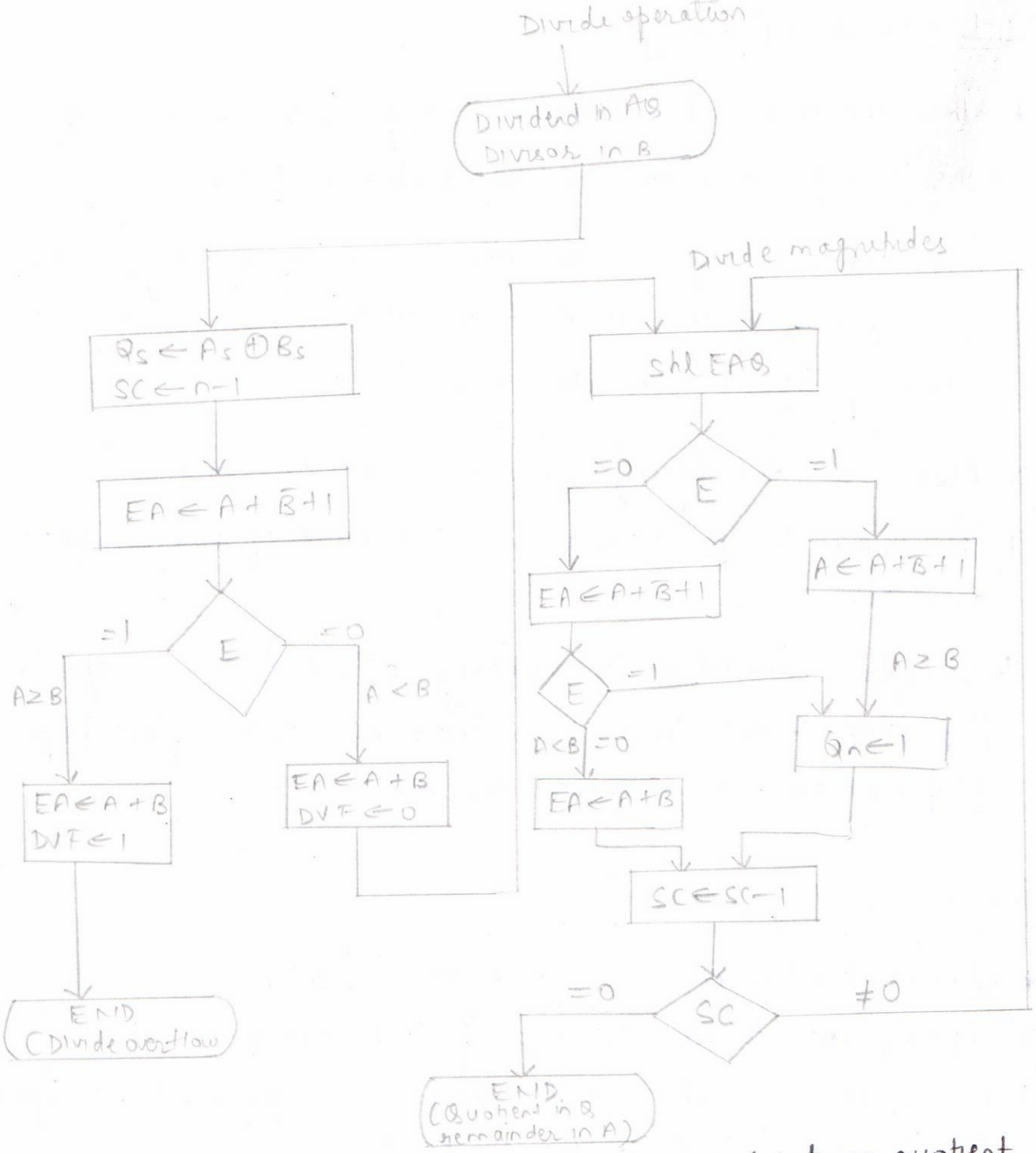
* When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows—

  "A divide overflow condition occurs if the high order half bits of the dividend constitute a number greater than or equal to the divisor."

* Moreover division by zero must be avoided.
* The divide overflow condition takes care of this condition as well.
* Overflow condition is usually detected when a special flip-flop is set. This is called divide-overflow flip-flop and can be labelled as DVF.

11.2. Hardware Algorithm

* Dividend is in A and Q and divisor in B.
* Sign of the result is transferred into Q.
* A constant is set into the sequence counter SC to specify the number of bits in the quotient.
* Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.



e.g. of overflow-

```
        1 0 1 1 1 1   ← 6 bits in quotient thus overflow
01000 ) 0 1 1 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 ↓
            ---------
            0 0 0 0 0
              0 1 0 0 0
              ---------
              1 0 0 0 0
                ⋮
```

e.g. 1 :- 10100011 ÷ 1011

A_Q = 10100011 B = 1011

$\overline{B}$ = 0100
+1
0101

$Q_S = A_S \oplus B_S$
= 0

SC = 4

EA = A + $\overline{B}$ + 1 : 1010
0101
1111

⇒ E = 0 ⇒ A >= B
DVF = 0

EA = A + B : 1111
1011
discard ↙ 11010

	E	A	Q	SC
		0 1 0 1	0 0 1 1	4
Shl EAQ	1	0 1 0 0	0 1 1 0	
add $\overline{B}$+1		0 1 0 1		
Q_n = 1		1 0 0 1	0 1 1 1	3
Shl EAQ	1	0 0 1 0	1 1 1 0	
add $\overline{B}$+1		0 1 0 1		
Q_n = 1		0 1 1 1	1 1 1 1	2
Shl EAQ	0	1 1 1 1	1 1 1 0	
add $\overline{B}$+1		0 1 0 1		
E = 1, Q_n = 1	1	0 1 0 0	1 1 1 1	1
Shl EAQ	0	1 0 0 1	1 1 1 0	
add $\overline{B}$+1		0 1 0 1		
E = 0		1 1 1 0		
Restore remainder		1 0 1 1		
A ← A+B	1	1 0 0 1	1 1 1 0	0
Q_n = 0		↓ Remainder	↳ Quotient	

e.g.2 :- 0000 1111 ÷ 0011

AQ = 0000 1111 B = 0011

$\bar{B}$ = 1100 + 1

$\bar{B}+1$ = 1101

E = A + $\bar{B}+1$: 0000 + 1101 = 1101 ⇒ E = 0 ⇒ A < B ⇒ DVF = 0

So, E = A + B ⇒ 1 1 1 1 0 + 0 0 1 1 = 1 0000 (discard)

	E	A	Q	SC
		0000	1111	4
Shl EAQ	0	0001	1110	
add $\bar{B}+1$		1101		
E=0, q_n=0		1110		3
Restore remainder	1	0001	1110	
Shl EAQ	0	0011	1100	
add $\bar{B}+1$		1101		
E=1, q_n=1	1	0000	1101	2
Shl EAQ	0	0001	1010	
add $\bar{B}+1$		1101	1010	
E=0, q_n=0		1110		
Restore remainder	1	0001	1010	1
Shl EAQ	0	0011	0100	
add $\bar{B}+1$		1101		
q_n=1	1	0000	0101	0

Remainder ← A = 0000; Quotient ← Q = 0101

11.3 Division Techniques

There are three methods to perform division in digital computer :-

a) Restoring method :- Here the partial remainder is restored by adding the divisor to the negative difference.

b) Comparison method :- In this method, A and B are compared prior to the subtraction operation. Then if $A \geq B$, B is subtracted from A.
If $A < B$, nothing is done. The partial remainder is shifted left and the numbers are compared again. The comparison can be determined prior to the subtraction by inspecting the end carry out of the parallel adder prior to its transfer to register E.

c) Non-restoring method :- In this method, B is not added if the difference is negative but instead, the negative difference is shifted left and then B is added.

Floating point Representation

* A floating point number have a decimal point.
  e.g. 3.12, 5.46, -12.14, 56.125
* These numbers represent wide variety of range.
* A decimal point in floating point numbers can float
  e.g. 1.2345 can also be represented as 12.345
* A floating point number representation has following parts—
  a) Significand :- containing digits of a number
     A -ve significand represents -ve numbers
  b) Exponent :- placement of decimal point relative to the beginning of significand
     A -ve exponent represents small numbers
     e.g 5.321 can be represented as —
     5321×10^{-3} (-3 → exponent; 10 → base; 5321 ↑ Significand)
* Any floating point number can be represented as —
  $S \times B^E$
* In 1985, IEEE 754 for Floating point Arithmetic was established, which is a technical standard for floating point computation.

IEEE 754 has three components —

a) Sign (S) :- 0 means +ve number
1 means -ve number

b) Biased/Modified Exponent (E') :- It can be -ve or +ve. A bias is added to actual exponent in order to get stored exponent.

c) Normalized Mantissa (M) :- Mantissa is that part of number which consists of significant digits. So, a normalised mantissa is one with only one 1 to the left of decimal

IEEE 754 Floating point

Single precision

* 32 bits
* Largest number :- all 32 bits as 1
* Smallest number :- all 32 bits as 1 with sign bit -ve

It is -2147483647 to 2147483647

* $E' = e + 127$
  ↑ bias
  ↑ Modified Exponent

Double precision

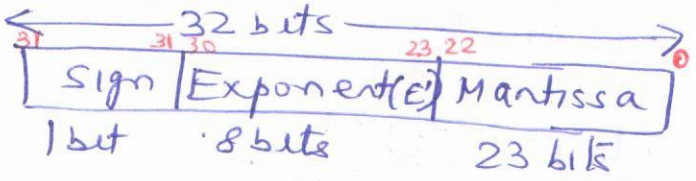
* 64 bits
* Largest number :- $2^{64}-1$
* Smallest number :- $-2^{64}-1$

Range :-
-9223372036854775807 to 9223372036854775807

* $E' = e + 1023$
  ↑ bias

Single precision Floating point Number

Format :-



e.g $(1460.125)_{10}$

S1 :- Conversion into binary form

So, $(1460.125)_{10} = (10110110100.001)_2$

S2 :- Normalizing binary number

$10110110100.001 = 1.0110110100001 * 2^{10}$

Now, $S = 0$

$e = 10$

$M = 01101101000001$

S3 :- Computing E'

$E' = e + 127$

$= 10 + 127$

$= 137$

$= (10001001)_2$

So,

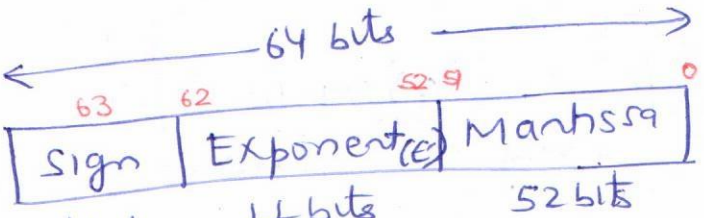
S	E'	M
0	10001001	01101101000001 00000000000

↳ padding of 0's to make it of 23 bits

is single precision floating point format

Double precision floating point number

Format:



e.g $(1460.125)_{10}$

$= (10110110100.001)_2$

$= 1.0110110100001 * 2^{10}$ // Normalized form

Now $S = 0$

$E = 10$

$M = 0110110100001$

$E' = e + 1023$

$= 10 + 1023$

$= (1033)_{10}$

$= (10000001001)_2$

So,

S	E'	M
0	10000001001	0110110100001.....0

↓ padding to make M of 52 bits

Note :- A floating point number allows the radix point to move.
If it is kept fixed then it is called fixed point number.

Floating point Addition & Subtraction

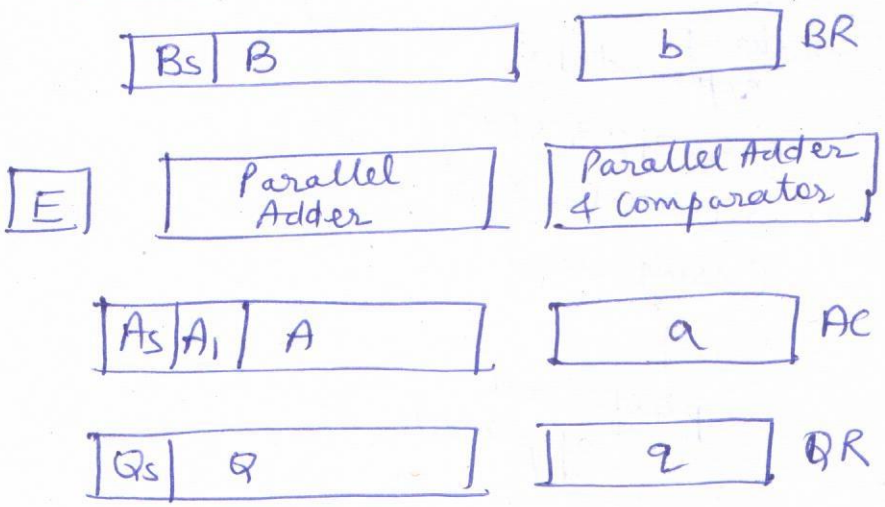
* Arithmetic operations on FP numbers consists of addition, subtraction, multiplication & division.

* A FP number is normalized if the most significant digit of mantissa is non zero.

* FP addition and subtraction requires an alignment of radix point since exponent parts should be equal.

* But FP multiplication and division donot require an alignment of mantissa —
  * Product can be formed by multiplying two mantissas and adding the exponent
  * Division is accomplished by dividing mantissas & subtracting exponents.

Hardware Implementation



Uppercase letters → mantissa

Lowercase letters → exponent

A_1 : MSB and it should be 1 for normalized

* During addition or subtraction, floating point operands are kept in AC & BR, the sum or difference is formed in AC.

* Algorithm can be divided into four steps —

a) Check for zeros
b) Align the mantissa
c) Add or subtract the mantissas
d) Normalize the result

(a) Case 1 : AC = 0, BR = 0.156 — AC + BR = 0.156 (AC)

Case 2 :- AC = 0.156, BR = 0 — AC + BR = 0.156 (AC)

Case 3 :- AC = 0.156, BR = 0 — AC − BR = 0.156 (AC)

Case 4 :- AC = 0, BR = 0.156 — AC − BR = − BR = −0.156

(b) Observe the value of exponents as these must be equal for add/subtract oper^n

e.g: 0.583123×10^3
0.123000×10^{-1}

Making exponent $= 10^{-1}$
0.230000×10^{-1}
0.123000×10^{-1}
(using shl)
↓
not preferred because some bits are lost

Making exponent $= 10^3$
0.583123×10^3
0.000012×10^3
(using shr)
↓
preferred approach

Floating point multiplication

* Floating point multiplication does not require an alignment of ~~mtissas~~; mantissas; product can be formed by multiplying two mantissas & adding the exponents.

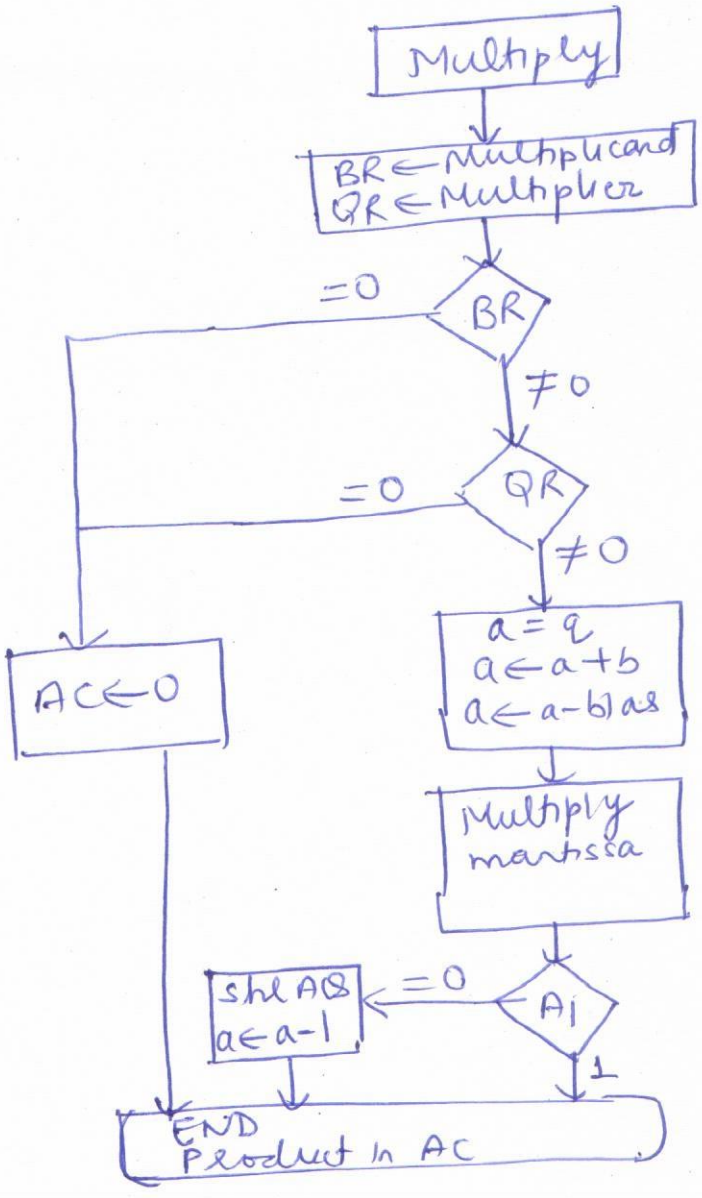
* The algorithm can be divided as —

S1 :- Check for zeros

S2 :- Add the exponents

S3 :- Multiply mantissas

S4 :- Normalize the result



(c) Add/subtract the mantissa and Normalizing
(d)†

e.g 0.534×10^3
$+ \; 0.712 \times 10^3$
1.246×10^3

Registers cannot ~~perform~~ store overflow bit so shr operation will be performed
0.124×10^4

0.534×10^3
0.521×10^3
0.013×10^3

is not floating point number
If MSB is zero, it is underflow condition & shl operation will be performed
0.130×10^2