Algorithms for Addition & Subtraction:-

(1) When the signs of A & B are identical, add the two magnitude and attach the sign of A to result.

↳ (Addition ~~magnitude~~) Combined

(2) When signs of A & B are different and the two magnitude, and attach the sign of A to result.

↳ (Subtraction)

(3) In addition when signs of A & B are different Compare the magnitude and subtract smaller from larger. chose sign of result to be same as 'A'. if (A>B) (or) Complement of A-sign if (A<B).

If two magnitudes are Equal subtract B from A and make sign of result positive.

(4). In subtraction when sign of A & B are identical, Compare the magnitude and subtract smaller from larger.

→ chose the sign of result same as A if (A>B) (or) Complement of A-sign if (A<B)

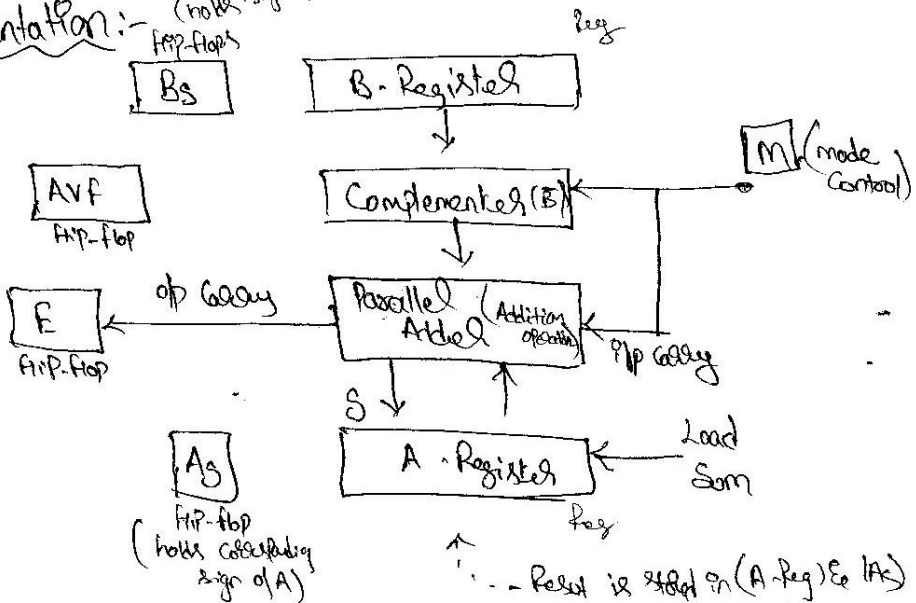
→ If the two magnitudes are Equal subtract 'B' from 'A' and make the sign of result positive.

note:-

→ the two algorithms are similar except for sign comparision.

→ The procedure to be followed for identical sign in addition algorithm is same as for different sign in subtraction algorithm. and vise-versa.

Hardware implementation:-



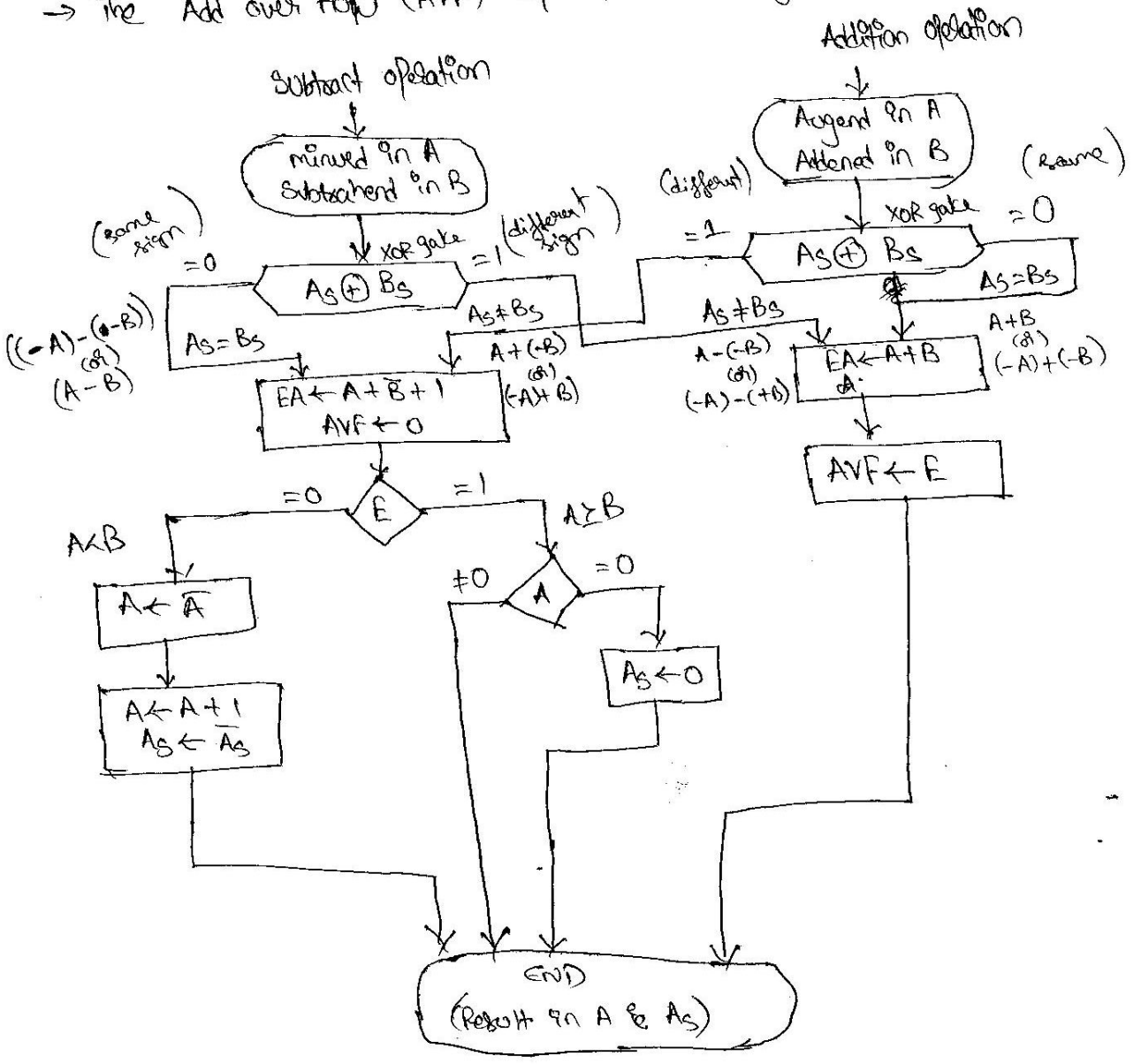
∴ - Result is stored in (A-Reg) & (AS) form Accumulator

→ A & B are two Registers hold magnitude of two numbers
→ A_s & B_s are flip-flops indicates sign.
→ Parallel Adder Perform Addition operation
→ (A & A_s) form an Accumulator where results stored.
→ Complemented Provided Complement for B-Reg based on mode i/p.

when M=0 ⇒ i/p Carry to Parallel Adder = 0 & o/p of B transfered to Parallel Adder.
i.e $S = A + B$. (Addition)

when M=1 ⇒ i/p Carry to parallel Adder = 1 & o/p of B is Complemented and transfered to parallel Adder.
$S = A + \bar{B} + 1$ ⇒ (Subtraction)

→ S is o/p of parallel Adder is applied to i/p of A-Reg. (Sum)
→ the o/p of Carry is transfered to flip-flop 'E' where it can be checked to determine the relative magnitude of two no:-
→ The Add over flop (AVF) flip-flop holds overflow bit when A & B are added.



To check relative magnitude of two no:- transfer to flip-flop 'E' is given as follow.

Case(i) → (A < B)

Eg: (1) A < B

let A = 2
B = 5

2 = 010
5 = 101 → 2's complete is

010 → is comple
1 + 1
011

Add A + B now:- 010
011
-3 ← 101

⇒ To define 'E' = 0 case.

here A = 010 is complemented give
A = 101 [A ← Ā] and then.

give A ← A + 1
A = 101
1
A ← 110 *

i.e A = 2's complent to define 'As'
110 ⇒ 001
1
010 = (2). Correct. and
[As ← Ās] Proved.

Case (ii) → (A ≥ B)

If E = 1

let A = 5 & B = 2.

5 = 101
2 = 010 → 2's complet ⇒ 101
1
110 (of 'B')

Subtraction operation

now A - B is

101
(+) 110
3 = 1011 here (A = 011 & E = 1)

Case (iii) If A = 2 ; B = 2.

A - B ⇒ 010
⇒ 110 (i.e As ← 0)
1 000 ⇒ A. (E = 1)

E = 1

Addition operation

Eg:- A = 2 ; B = 5
= 010
101
no Carry 111

Eg: A = 3 ; B = 5
= 011
= 1 101
Carry ← 1 1000
over flow. ↓
8 value

here over flow bit of 8 can not be stored so Carry out of E is copied in to Add over flow (AVF)

[AVF ← E] ↑.

Multiplication of two fixed point binary no's in signed representation

Multiplication of two fixed point binary numbers in signed representation is done with paper and pencil by a process of successive shift and add operations as shown below:

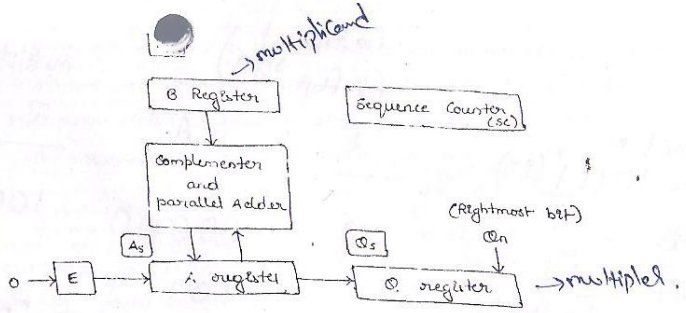
```
  23        10111      Multiplicand
 x19       x 10011     Multiplier
           ------
           10111
          10111
          00000
         00000
        10111
 437    ----------     Product
        110110101
```

→ The process consists of looking at successive bits of the multiplier, least significant bit first.
→ If multiplier is 1, multiplicand is copied down; otherwise zeros are copied down.
→ Numbers copied in successive lines are shifted by 1 position to the left from the previous number.
→ Finally, numbers are added & final product is formed.
→ Sign of the product is determined by the signs of the multiplicand and the multiplier.
  - If they are same ⇒ sign is +ve
  - If they are different ⇒ sign is -ve.

→ Hardware Implementation

Multiplication is implemented in digital computer with a few changes with respect to normal process.

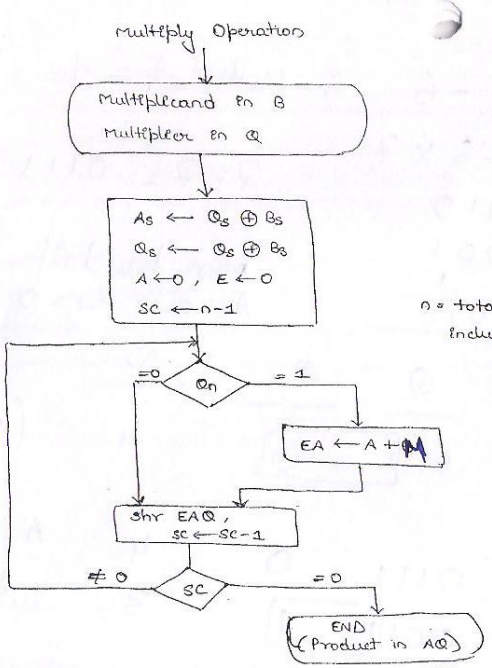
1) Instead of providing registers to store intermediate binary numbers (which is equal to no. of bits of multiplier) and then adding to obtain the product, it is convenient to provide an adder for the summation of only binary nos. & successively accumulate the partial products in a register.
2) Instead of shifting the multiplicand to the left, the partial product is shifted right, which results in leaving the partial product and the multiplicand in the required relative positions.
3) When corresponding bit of multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.



→ Hardware requires two more registers SC and Q when compared to addition & subtraction.
→ Initially, E = 0 and A = 0
→ Multiplicand is in B. Multiplier is in Q and its sign in Q_S
→ SC is initially set to number equal to no. of bits in multiplier.
→ SC is decremented by 1 after forming each partial product.
→ Least significant bit is denoted by Q_n which is tested.
→ When SC = 0, product is formed & process stops.
→ If multiplier bit is 0, nothing is done.
→ If multiplier bit is 1, sum of A and B form partial product which is transferred to EA register.
→ Then both partial product and multiplier are shifted right (E → A → Q) by 1 bit in both the cases.
→ Q_n will now hold the bit of the multiplier which must be inspected next.
→ Sign of result is stored in Q_S which is XOR of Q_S and B_S.
→ Initial value of SC = n-1 where word has n-bits. Since an operand must be stored with its sign, 1 bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

2 x 3 = 6

```
2 = 010  → multiplicand
3 = 011  → multiplier
   010
  010
 000
------
00110  = 6 → product
```



multiplier n = no. of bits of Q

Multiplicand (23) M = 10111	E	A	Q (19)	SC n-1
Multiplier in Q	0	00000	10011	101 5
Qn = 1 ; add M		10111	Qn	
1st partial product	0	10111		
Shift Right EAQ	0	01011	11001	100 4
Qn = 1 ; add M		10111		
2nd partial product	1	00010		
Shift Right EAQ	0	10001	01100	011 3
Qn = 0 ; shr EAQ	0	01000	10110	010 2
Qn = 0 ; shr EAQ	0	00100	01011	001 1
Qn = 1 ; add B		10111		
5th partial product	0	11011		
Shift right EAQ	0	01101	10101	000 0

Final product = AQ = 01101 10101

Q = 10011 = $(19)_{10}$, B = 10111 = $(23)_{10}$

AQ = Result = 0110110101 = $(437)_{10}$.

(5.12) of two binary numbers in signed-magnitude representation, 11111 (multiplicand) and 10101 (multiplier). The signs are not included.

Eg: Multiplicand B = 11111 = $(31)_{10}$, Multiplier Q = 10101 = $(21)_{10}$.

⇒ AQ = B × Q = 31 × 21 = $(651)_{10}$.

Multiplicand B = 11111.	E	A	Q	SC
Multiplier in Q	0	00000	10101	101
Qn = 1 ; add B		11111		
shr EAQ	0	11111		
shr EAQ	0	01111	11010	100
Qn = 0; shr EAQ	0	00111	11101	011
Qn = 1; add B		11111		
	1	00110		
shr EAQ	0	10011	01110	010
Qn = 0; shr EAQ	0	01001	10111	001
Qn = 1 add B		11111		
	1	01000		
shr EAQ	0	10100	01011	000

Final product = AQ = 10100 01011 = $(651)_{10}$.

d) Multiplication with signed-2's complement data (Booth's Algorithm)

Principle:

String's of 0's in the multiplier does not require any addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$

Eg:- 011110 (+14) has string of 1's from 2^3 to 2^1 ⇒ k = 3 and m = 1.

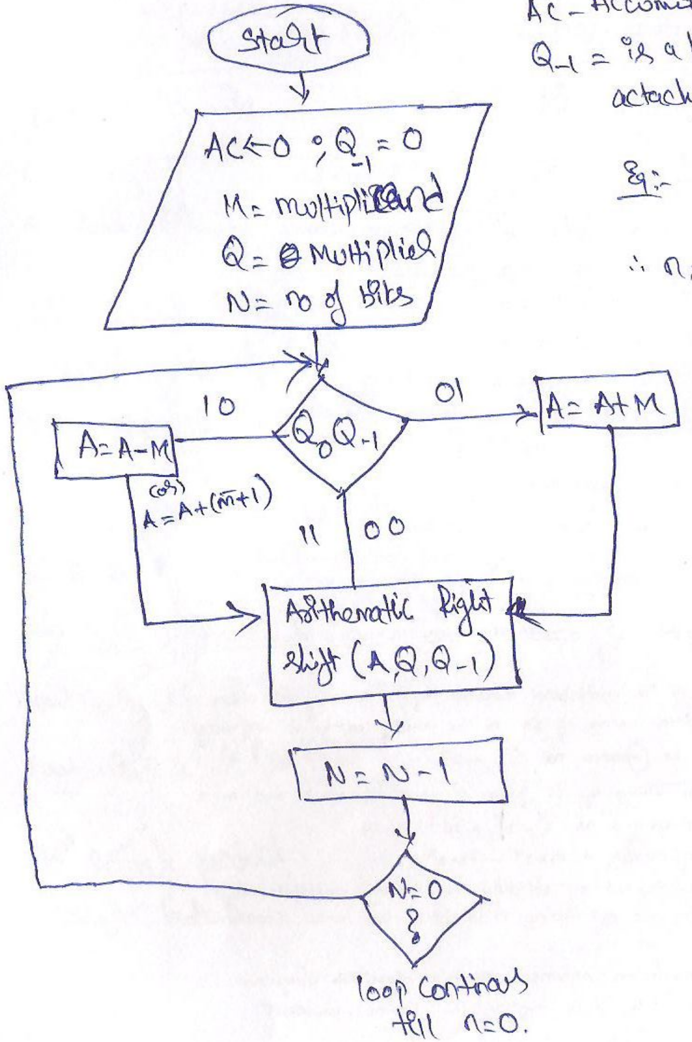
⇒ no. can be represented as $2^4 - 2^1 = 16 - 2 = 14$

⇒ Multiplication of M × 14 = $M \times 2^4 - M \times 2^1$

Product can be obtained by shifting the binary multiplicand M by M times to the left and subtracting M shifted left once (arithmetic shift).

As in all multiplication schemes, Booth's algorithm requires examination of multiplier bits and shifting the partial product.

Booth's Algorithm



AC – Accumulator

Q_{-1} = is a bit which comes actually after Q_0

Eg:- 0110 0

$Q_3Q_2Q_1Q_0$ Q_{-1}

∴ n = 5. with Q_{-1} also

00 } shift right
11 }

01 → A ← A+M (Add) & shift Right

10 → A ← A−M (Sub) & shift Right

Eg:- Multiplicand −6 & multiplier = 7

−6 × 7

Q = 7 = 0111.

+6 = 0110

1's = 1001

M = 2's ← 1010 = M

M = (−6)

From flowchart below table given.

AC = 0 ; Q_{-1} = 0

	A	Q	Q_{-1}	N	Operation
Initialize		0111	0	4	(A ← A−M & shift Right & N ← N−1)
Step − 1	0000				A ← A+(M̄+1)
(A−(−6) 2's)	− 1010			4	
	0110	0111	0	3	~~Shift Right & N←N−1~~
	0011	0011	1		Shift Right & N←N−1
Step − 2	0001	1001	1	2	
Step − 3	0000	1100	1	1	shift Right & N←N−1
Step − 4	0000 + 1010				A←A+M & shift Right & N←N−1
	1010	1100	1	1	
	1101	0110	0	0 (N=0 loop stop)	

Product = AQ is

= 1101.0110 ⇒ (sign −ve)

Prefer 2's complement for exact Answer.

1's compl: 0010100 1

+ 1

0101010 = 42.

32 8 2

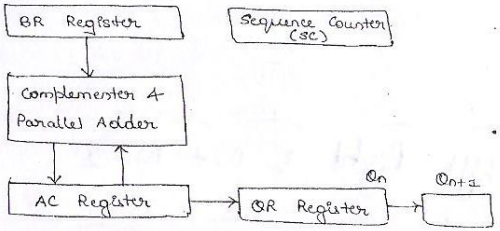
(i.e Ans = −42).

product, subtracted from partial product & left changed according to following rules.

1. Multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. Multiplicand is added to partial product upon encountering the 1st 0 (provided that there was a previous 1) in a string of 0s in the multiplier.
3. Partial product doesnot change when the multiplier bit is identical to the previous multiplier bit.

(If we encounter a "0" without previous 1, donot change partial product because it is assumed to have a previous "0" for LSB bit of multiplier):

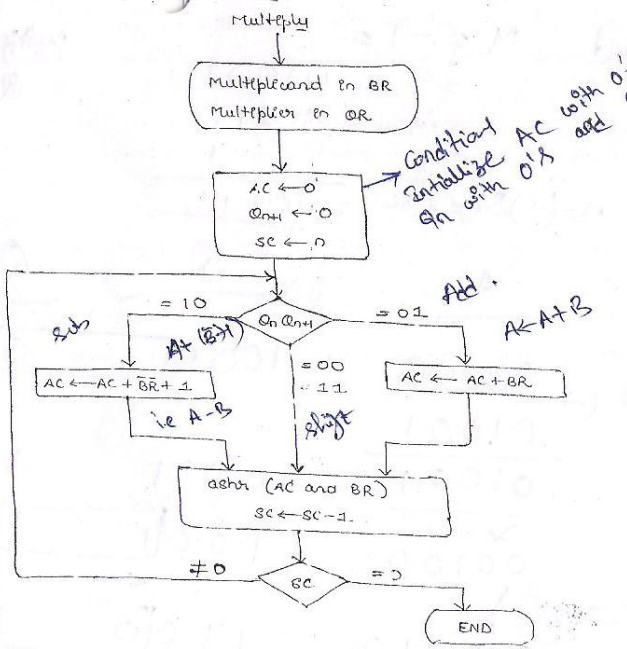
→ Hardware for Booth's Algorithm



Qn designates the LSB of the multiplier in register QR. An extra bit Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier.

Here, sign bits are not seperated from the rest of the registers. To show the difference, we rename the registers A, B and Q, as AC, BR and QR, respectively.

Q_0	Q_{-1}	
0	0	} shifting Right
1	1	
1	0	– $(A \leftarrow A-M$ & shifting$) = A \leftarrow (A+(\bar{M}+1))$
0	1	→ $A \leftarrow A+M$ & shifting. ✓



Eg:- $(-9) \times (-13) = +117$, Assume $n = 5$

QR = −13 = 10011, BR = −9 = 10111.

2's complement of BR = 9 = 01001

BR = 10111

BR'+1 = 01001.

Qn Qn+1		AC	QR	Qn	SC
	Initial	00000	10011	0	101
10	Subtract BR	01001			
	⇒ AC + (BR'+1)	01001			
	BR+1 = 01001				
	ashr	00100	11001	1	100
11	ashr	00010	01100	1	011
01	Add BR	10111			
	⇒ AC + BR	11001			
	BR = 10111				
	ashr	11100	10110	0	010
00	ashr	11110	01011	0	001
10	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

Product = AC QR = 0001110101 = 117

Booth algorithm when the following binary nos. a multiplier
Assume 5-bit registers that hold signed numbers. The multiplicand
in both cases is (+15).

a) (+15) × (+13)

b) (+15) × (−13)

Sol: a) (+15) × (+13) = +195 = $(0\,0\,1\,1\,0\,0\,0\,0\,1\,1)_2$

BR = 01111 (+15) ⇒ $\overline{BR}+1$ = 10001 (−15)

QR = 01101 (+13).

Q_n	Q_{n+1}		AC	QR	Q_{n+1}	SC
		Initial	00000	01101	0	101
1	0	Subtract BR	10001			
			10001			
		ashr	11000	10110	1	100
0	1	Add BR	01111			
			00111			
		ashr	00011	11011	0	011
1	0	subtract BR	10001			
			10100			
		ashr	11010	01101	1	010
1	1	ashr	11101	00110	1	001
0	1	Add ~~subtract~~ BR	01111			
			01100			
		ashr	00110	00011	0	000

+195.

b) (+15) × (−13) = −195 = $(1\,1\,0\,0\,1\,1\,1\,1\,0\,1)_2$

BR = 01111 (+15), $\overline{BR}+1$ = 10001 (−15), QR = 10011 (−13)

Q_n	Q_{n+1}		AC	QR	Q_{n+1}	SC
		Initial	00000	10011	0	101
1	0	Subtract BR	10001			
			10001			
		ashr	11000	11001	1	100
1	1	ashr	11100	01100	1	011
0	1	Add BR	01111			
			01011			
		ashr	00101	10110	0	010
0	0	ashr	00010	11011	0	001
1	0	Subtract BR	10001			

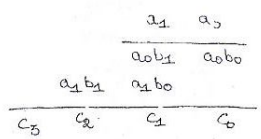
checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add & shift micro-operations.

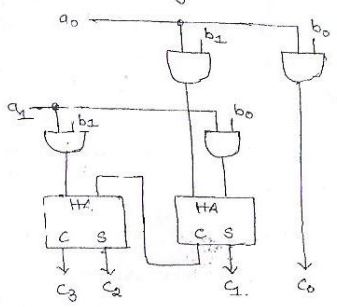
The multiplication of two binary numbers can be done with 1 micro-operation by means of a combinational circuit that forms the product bits all at once.

Adv:- fast way of multiplication two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array.

Disadv:- Requires a large number of gates, and ⇒ not economical.

Figure below shows a 2-bit by 2-bit array multiplier





Multiplicand = $b_1\,b_0$

Multiplier = $a_1\,a_0$

Product = $C_3\,C_2\,C_1\,C_0$.

— The first partial product is obtained by multiplying a_0 by b_1b_0.
— Multiplication of a_0 with b_0 produces 1 if both are 1's and produces 0 if one of them is 0. This is same as AND gate
— 1st partial product is obtained by means of 2 AND gates.
— 2nd partial product is formed by multiplying a_1 by b_1b_0 and is shifted 1 position to left.
— The two partial products are added with two Half-adders (HA) circuits.
— Least significant bit of the product need not pass through an adder since it is formed by the output of the first AND gate.

Principle:- A combinational circuit binary multiplier with more bits can be constructed. A bit of the multiplier is ~~added~~ ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a

→ Division of two fixed-point binary nos. in signed-magnitude representation is done with paper & pencil by a process of successive compare, shift & subtract operations

→ Binary division is simpler than decimal division because the quotient digits are either 1 or 0 and there is no need to estimate how many times the dividend or partial ~~remainder~~ remainder fits into the divisor.

→ Eg:- 0111000000 ÷ 10001

```
Divisor                  11010     Quotient = Q
B = 10001 ) 0111000000            Dividend = A
            10001                 here 5 bits of A < B
            -----
            011100                here 6 bits of A > B
             10001                shift right B & subtract;
             -----                enter 1 in Q.
              01011 0             7 bits of A > B
               10001              shift right B & subtract;
               -----              enter 1 in Q
                00101 0           Remainder less than B; enter 0 in Q
                 01010            Remainder ≥ B
                 10001            shift right B & subtract;
                 -----            enter 1 in Q
                 00011 0          Remainder < B; enter 0 in Q

⇒ (00110) — final remainder
```

— Divisor B consists of 5 bits and the dividend A consists of 10 bits
— 5 MSB bits of A are compared with the divisor.
— Since, 5 bit number < B, we try again by taking 6 bits of A and compare with B.
— 6 bit no. > B ⇒ place 1 in Quotient bit
— Divisor is then shifted right once and subtracted from the dividend. The difference is partial remainder because division could have stopped here to obtain a quotient 1 and a remainder equal to partial remainder
— The process is continued by comparing a partial remainder with the divisor.

Rule:- 1) If partial remainder > B, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder
2) If partial remainder < divisor B, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case.
3) Result gives both Quotient & Remainder

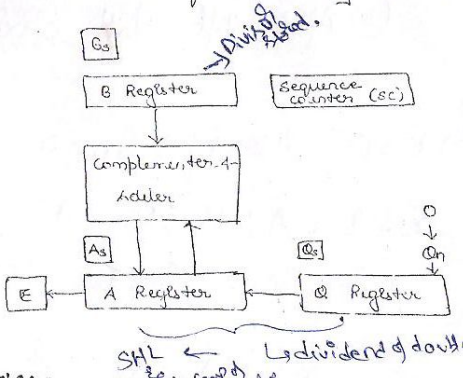
Division Algorithm.

(7.20)

When division is implemented in digital computer, it is convenient to change the process slightly.

1) Instead of shifting the divisor to the right, the dividend or partial remainder, is shifted to the left. This leaving the two numbers in the required relative positions.
2) Subtraction can be achieved by adding A to 2's complement of B. The information about the relative magnitudes is given by end-carry E.

Hardware for implementing division is same as multiplication



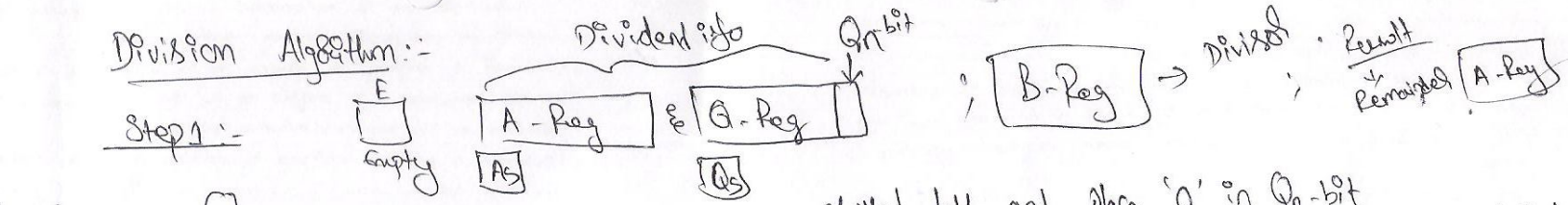
✓ Here, EAQ is shifted to the left with 0 inserted into Qn and the previous value of E is lost.

↳ dividend of double length.

Steps:-

— Divisor is stored in B register and the double length dividend is stored in register A and Q.
✓ Dividend is shifted to left and the divisor is subtracted by adding the 2's complement value.
— The information about the relative magnitude is available in E
✓ If E = 1, ⇒ A ≥ B ⇒ Quotient bit = 1 and is inserted in Qn and partial remainder is shifted to left to repeat the process.
✓ If E = 0, ⇒ A < B ⇒ Qn remains as 0 and value of B is then added to restore the partial remainder in A to its previous value.
partial remainder is shifted left and the process is repeated again until all 5 Quotient bits are formed.
✓ Sign of quotient is determined by signs of the dividend and the divisor.
If the 2 signs are alike, quotient sign = +ve
If the 2 signs are different, quotient sign = -ve

Division Algorithm :-



Step 1 :-

① Initially E = Empty A & Q are shifted left and place '0' in Qn-bit

② After left shift operation if E = 0 then A < B ; no DVF occurs & value divided is to by adding B to A.

E = 1 then A ≥ B ; Divide overflow (DVF) is set to '1' and terminated

In E = 0 ; A < B ; Qn = 0 value B-added to A. and defines (Remainder is restored in A-Reg)

In E = 1 ; A ≥ B ; Qn = 1 A ← A + B̄ + 1 is done and E is copied into Qn and continous until Sequence Counter is '0' //

① Initial E = ? A Q & Qn bit '0'

→ Shift left

→ Add A ← A + B̄ + 1 After adding if result is E = 1 then copy to Qn location. and (Sc ← Sc - 1)

" " " " if " " E = 0 then. " " " " and A ← A + B if the result is '1'

then defines restore remainder in A-Reg

Divided A = 01110 00000

Divisor B = 10001 $\quad \overline{B}+1 = 01111$

	E	A	Q	Qn	SC	
Dividend	0	01110	00000		5	→ initial value of E = 0
Shl EAQ	0	11100	00000			→ shift left
Add $\overline{B}+1$		01111				→ Add 2's comp of B
E = 1	1	01011				→ Copy E to Qn-bit
Set Qn = 1	1	01011	00001		4	→ N ← N − 1
Shl EAQ	0	10110	00010			when E = 1 shift left
Add $\overline{B}+1$		01111				Add 2's comp of B
E = 1	1	00101				copy E to Qn
Set Qn = 1	1	00101	00011		3	N ← N − 1
Shl EAQ	0	01010	00110			E = 1; shift left
Add $\overline{B}+1$		01111				Add 2's complement of B
E = 0, leave Qn = 0	0	11001	00110			copy E = 0 to Qn = 0
Add B		10001				Add B
Restore Remainder	01	01010	00110		2	Restore Remainder; shift left
Shl EAQ	0	10100	01100			N ← N − 1
Add $\overline{B}+1$		01111				Add B's 2's Complement
E = 1	1	00011				E → Qn copied
Set Qn = 1	1	00011	01101		1	N ← N − 1
Shl EAQ	0	00110	11010			shift left
Add $\overline{B}+1$		01111				Add 2's of B
E = 0, leave Qn = 0	0	10101	11010			E = copy to Qn
Add B		10001				E = 0 then Qn = 0; Add B
Restore Remainder	1	00110	11010		0	Restore Remainder.
Neglect E.						N ← N − 1
Remainder in A :		00110				neglect E.
Quotient in Q :			11010			

(55)

division of a) 10100011 by 1011

b) 00001111 by 0011 (use 8 bit dividend).

Sol: a) 10100011 by 1011. $\Rightarrow \frac{163}{11} = 14 + \frac{9}{11}$.

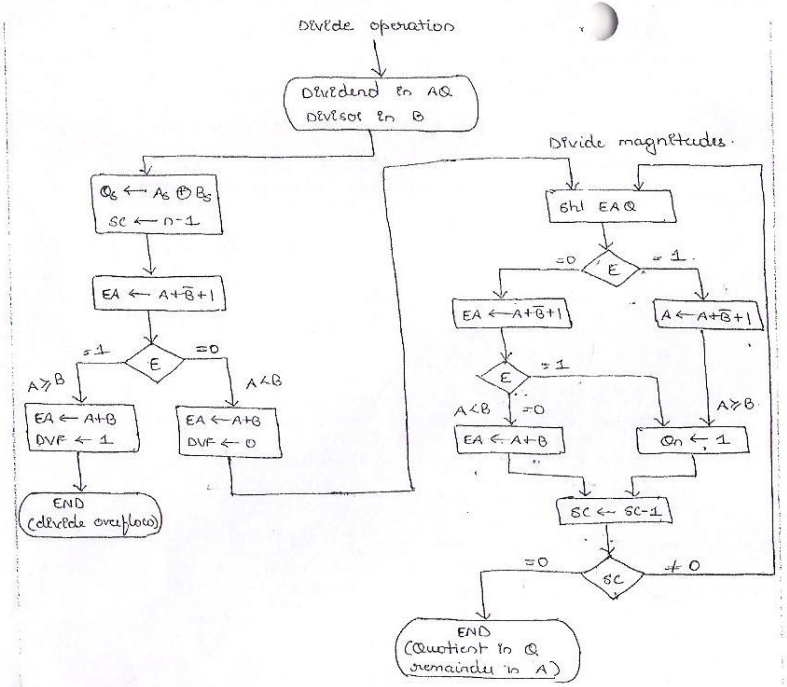
$= \frac{10100011}{1011} = 1110 + \frac{1001}{1011}$.

B = 1011, $\overline{B}+1 = 0101$

	E	A	Q	SC	
Dividend in AQ	0	1010	0011	100	
Shl EAQ	1	0100	0110		
Add $\overline{B}+1$		0101			
E = 1, Set Qn = 1	1	1001	0111	011	
Shl EAQ	1	0010	1110		
Add $\overline{B}+1$		0101			
E = 1, Set Qn = 1	1	0111	1111	010	
Shl EAQ	0	1111	1110		
Add $\overline{B}+1$		0101			
E = 1, Set Qn = 1	1	0100	1111	001	
Shl EAQ	0	1001	1110		
Add $\overline{B}+1$		0101			
E = 0, leave Qn = 0	0	1110	1110		
Add B		1011			
Restore remainder	1	1001 (Remainder)	1110 (Quotient)	000	

$\Rightarrow \frac{10100011}{1011} = 1110 + \frac{1001}{1011}$.

(56)



- Dividend in A and Q
- Divisor is in B
- sign of result is transferred in Q_s ($Q_s \leftarrow A_s \oplus B_s$)
- Sequence counter is initialized with a constant = no. of bits in Quotient ($SC \leftarrow n-1$)
- Divide overflow condition is checked by subtracting the divisor in B from half of bits of dividend stored in A. ($EA \leftarrow A+\bar{B}+1$)
  - If $A \geq B$, DVF is set & operation is terminated
  - If $A < B$, no divide overflow occurs so the value of dividend is restored by adding B to A.
- Division of magnitudes starts by shifting the dividend in AQ bit to left with the higher-order bit shifted into E.
- If $E = 1 \Rightarrow EA > B$ because EA consists of 1 followed by n-1 bits while B consists of only n-1 bits & B must be subtracted from EA & 1 is inserted into Q_n for quotient bit

in E), its value is $EA - 2^{n-1}$

$\Rightarrow$ Adding 2's complement of B to A

$= A + \bar{B} + 1 = EA - 2^{n-1} + 2^{n-1} - B$

$= EA - B$

The carry from this addition is not transferred to E if we want E to remain a 1.

- If the shifted bit inserts a 0 into E, the divisor is subtracted by adding its 2's complement value & the carry is transferred into E.

  Now if $E = 1 \Rightarrow A \geq B \Rightarrow Q_n = 1$

  if $E = 0 \Rightarrow A < B \Rightarrow$ original no. is restored by adding B to A and $Q_n = 0$.

- The process is repeated again with register A holding the partial remainder.
- After n-1 times, the quotient magnitude is formed in register Q and the remainder is found in register A.
- Quotient sign in Q_n, sign of remainder in A_s = same as sign of dividend.

b) Other Algorithms of division

- Division algorithm discussed till now is called Restoring Method, because the partial remainder is restored by adding the divisor to the negative difference.
- Two other methods are

a) Comparison Method:

Here, A and B are compared prior to subtraction operation.

Then, if $A \geq B$, B is subtracted from A

if $A < B$, nothing is done

The partial remainder is then shifted left & nos. are compared again.

b) Non-Restoring Method:

Here, B is not added if difference is negative, but instead, the negative difference is shifted left & then B is added.

- when $A < B$,

  In restoring method,

  B is subtracted from A & then added to A to restore. The no. is shifted left (i.e multiplied by 2) & B is subtracted again.

  $\Rightarrow (A-B+B) \times 2 - B = 2A - B$.

*) ALU Design:-

4-Basic Arithmetic operations → Addition, Subtraction, Multiplication, Division.

→ An arithmetic instruction may specify binary (or) Decimal data.
→ In each case data may be fixed point (or) floating point form.

→ 3-ways of representing negative fixed point
  (a) Signed - 1's complement
  (b) Signed - 2's " → (most computers use) Performing arithmetic operations on integers.
  (c) signed - Magnitude.

→ for floating point operations most computer used Signed Magnitude representation for the Mantissa.

→ The 4-basic Arithmetic operations considered for following 4 types:-
  (1) fixed-point binary data in signed-magnitude representation.
  (2) " " " " " " " - 2's complement "
  (3) floating-point binary data
  (4) Binary coded decimal (BCD) data.

Addition & Subtraction of signed-magnitude numbers:-

Rule	operations	Add magnitudes	Subtract magnitudes: when A>B	A<B	A=B
1)	(1) (+A) + (+B)	+(A+B)	+(A-B)	-(B-A)	+(A-B)
2)	(3) (+A) + (-B)		-(A-B)	+(B-A)	+(A-B)
3)	(3) (-A) + (+B)	-(A+B)	+(A-B)	-(B-A)	+(A-B)
4)	(1) (-A) + (-B)				
5)	(4) (+A) - (+B)	+(A+B)			
6)	(2) (+A) - (-B)				
7)	(2) (-A) - (+B)	-(A+B)	-(A-B)	+(B-A)	±(A-B)
8)	(4) (-A) - (-B)				