1/2/2020
Saturday

Unit - 2

Arithmetic

1. Introduction :-

→ A basic operation in all computers is the addition or subtraction of two numbers.

→ Arithmetic operations occur at the machine instruction level. They are implemented along with basic logic functions such as AND, OR, NOT, and EXCLUSIVE-OR (XOR), in the ALU subsystem of the processor.

→ But in this chapter, Logic circuits used to implement arithmetic operations.

→ Multiply & divide operations require more complex circuitary than either addition & subtraction operations, and also affect performance.

→ Compared with arithmetic operations, logic operations are simple to implement combinational circuitry. They require only independent Boolean operations whereas carry/borrow lateral signals are required in arithmetic operations.

→ To perform addition and subtraction operations 2's complement is the best representation.

→ A Logic circuit is designed to add unsigned binary numbers can also be used to add signed numbers in 2's complement.

Difference between Signed and Unsigned Numbers:

unsigned	signed
→ Valid only for positive nos	→ Valid for both +ve & -ve nos
→ NO sign bit concept.	→ sign bit concept is used.
→ It is the straight binary Representation	→ MSB of the entire Number is known as sign bit(S). S=0 ⇒ positive Number S=1 ⇒ Negative NO.
→ NO further categorization	→ Here there are 3 types of categorization. 1. Signed Magnitude Rep. 2. 1's complement 3. 2's complement.

TOPIC 1: Addition and Subtraction of Signed Numbers

x_i	y_i	Carry-in (c_i)	Sum (s_i)	Carry-out (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
0	1	1	0	1
1	1	1	1	1

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

$$= x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i.$$

→ The above figure shows the logic truth table for the sum (s_i) and carry out (c_{i+1}) functions for adding equally weighted bits x_i & y_i in two nos x & y.

Ex:

x	7	0	1	1	1		
+y	+6	0	1	1	0	0	
z	13	1	0	0	1		

Carry out c_{i+1} → □ y_i □ ← carry-in c_i ; x_i ; s_i

Legend for stage s_i.

fig: Logic specification for a stage of binary addition

→ The logic expression for s_i can be implemented with a 3-input XOR gate.

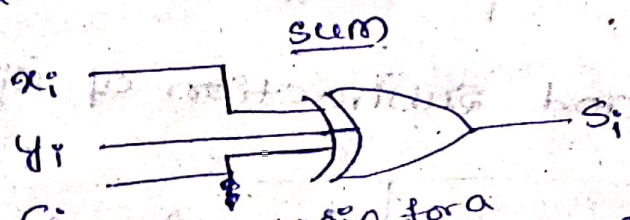


fig: Logic for a single stage of binary addition.

→ The carry-out function, c_{i+1}, is implemented with a two-level AND-OR logic circuit.

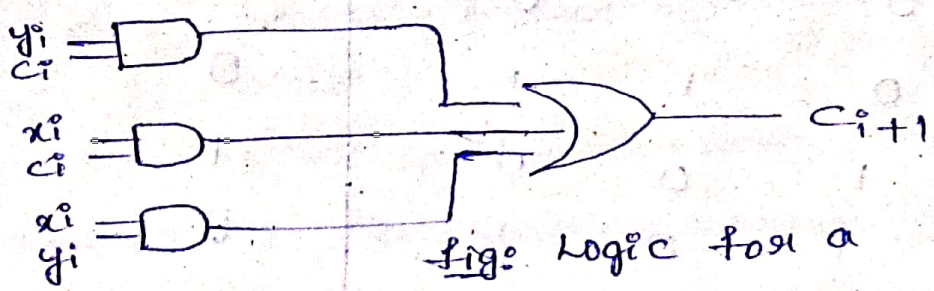
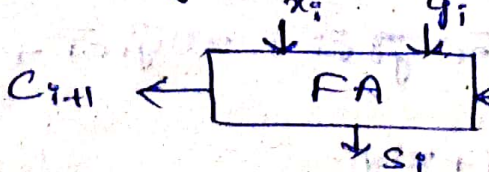
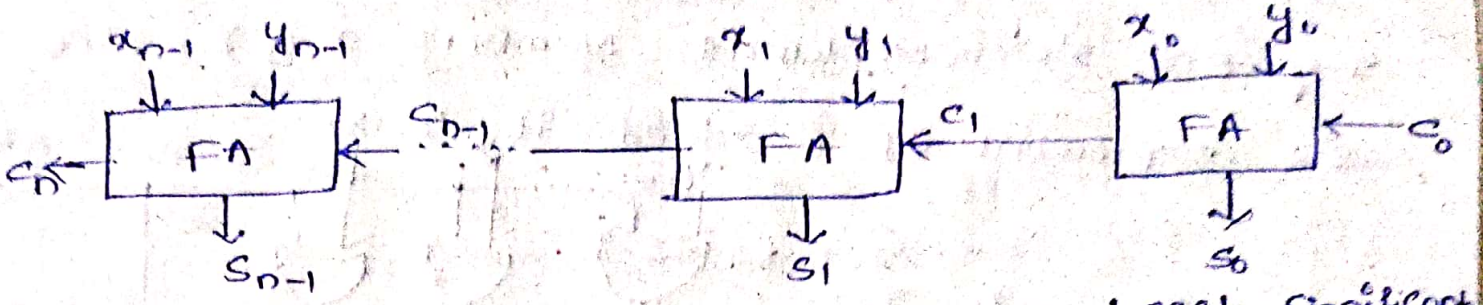


fig: Logic for a single stage

→ A convenient symbol for the complete circuit for a single stage of addition called a full Adder (FA)



→ The cascaded connection of n full adder blocks:



Most significant bit (MSB) position

Least significant bit (LSB) position

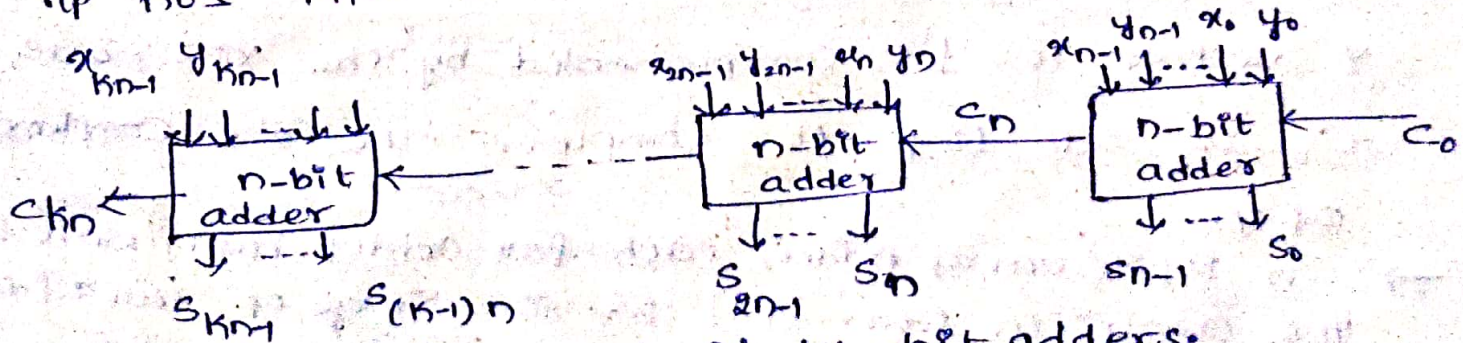
(b) fig: An n-bit ripple-carry adder.

→ It can be used to add two n-bit numbers. since the carries must propagate, (or) ripple, through this cascade, the configuration is called an n-bit ripple-carry adder.

→ The carry-in (c_0) into the least-significant-bit (LSB) position provides a convenient means of adding 1 to a number.

→ for instance, forming the 2's complement of a no involves adding 1 to the 1's complement of the number.

→ The carry-signals are also useful for interconnecting k-adders to form an adder capable of handling ip nd's that are kn bits long, as shown in fig.



(c) Cascade of k n-bit adders.

All(c) Total fig name :- Logic for Addition of binary vectors.

Addition / subtraction Logic unit :-

→ In order to perform the subtraction operation $x-y$ on 2's complement numbers X & Y, the 2's complement of Y added it to X.

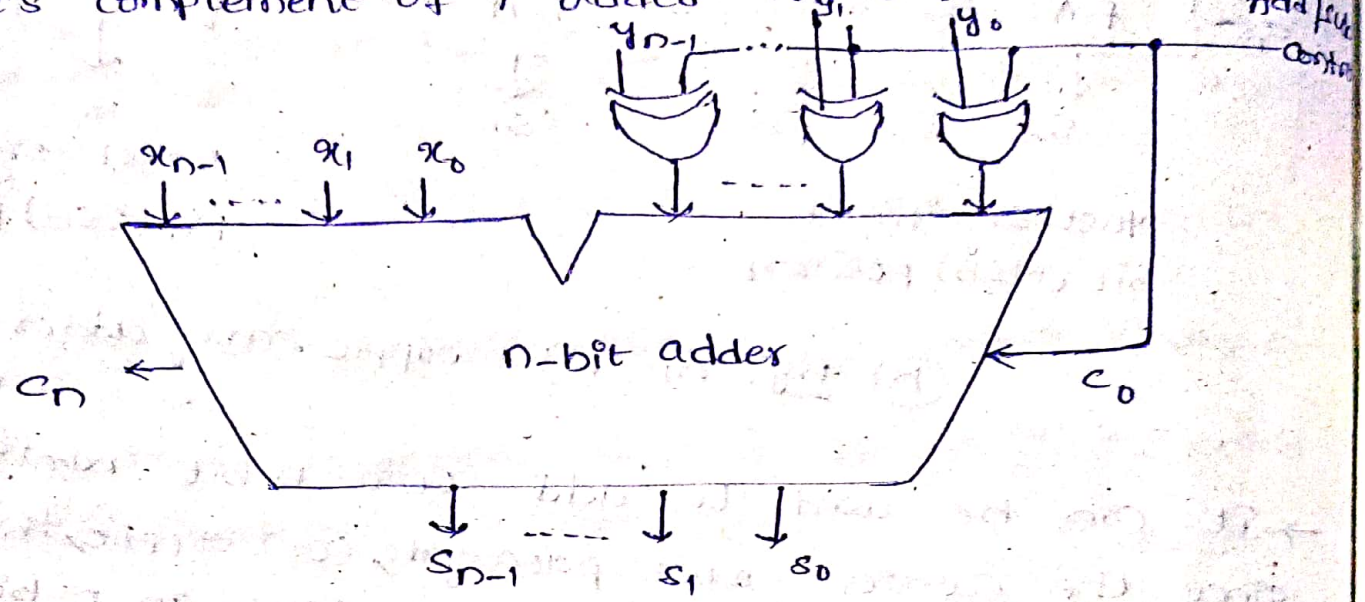


fig: Binary addition-subtraction logic network.

→ This add/sub Logic unit can be used perform either addition (or) subtraction based on the valued given to the add/sub control line.

→ This line is set to 0 for addition, applying the Y vector unchanged to one of the adder inputs along with a carry-in signal, c_0 of 0.

→ When the Add/sub control line is set to 1, the Y vector is 1's complemented by the XOR gates & c_0 is set to 1 to complete the 2's complementation of Y.

→ In Ripple carry Adder, each full adder has to wait for its carry in from its previous stage of full adder.

→ This causes a delay & make ripple carry adder extremely slow. To overcome this carry look-Ahead Adder

Topic 2: Design of Fast Adders

→ If an n-bit ripple-carry adder is used in the addition/subtraction unit, it may have too much delay in developing its outputs.

→ In order to reduce the delay the approach is to used the fastest possible electronic technology in implementing the Ripple-carry-logic design. (or) carry-look ahead addition.

Carry-Look Ahead Addition:-

→ Carry Look Ahead Adder is an improved version of the Ripple-carry adder

→ The Logic Expressions for full adder sum (S_i) & Carry out is (C_{i+1}) given as,

$$S_i = x_i \oplus y_i \oplus c_i \quad \text{—} \rightarrow (1)$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i \quad \text{—} \rightarrow (2)$$

Factoring the Eqn (2) we get,

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

we can write $\boxed{c_{i+1} = G_i + P_i C_i}$

$$G_i = x_i y_i \qquad P_i = (x_i + y_i).$$

→ The G_i and P_i are the called the Generate & Propagate functions.

→ The Propagate functions means that an input carry will produce an Output carry when either x_i is 1 (or) y_i is 1.

→ Each bit stage contains an AND gate to form G_i an OR gate to form P_i & a three i/p XOR gate to form S_i.

→ The adequate propagate functions can be realized as, $P_i = x_i \oplus y_i$ which differs from $P_i = x_i + y_i$ only when $x_i = y_i = 1$.

→ $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$

$= x_i y_i + (x_i + y_i) c_i$

$$\boxed{c_{i+1} = G_i + P_i c_i}$$

When $i = 0$,

$c_{0+1} = G_0 + P_0 c_0 \Rightarrow c_1 = G_0 + P_0 c_0$

$i = 1$, $c_{1+1} = G_1 + P_1 c_1 \Rightarrow c_2 = G_1 + P_1 c_1$

$= G_1 + P_1 (G_0 + P_0 c_0)$

$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$

$i = 2$, $c_{2+1} = G_2 + P_2 c_2$

$c_3 = G_2 + P_2 [G_1 + P_1 G_0 + P_1 P_0 c_0]$

$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$

$i = 3$, $c_{3+1} = G_3 + P_3 c_3$

$c_4 = G_3 + P_3 [G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0]$

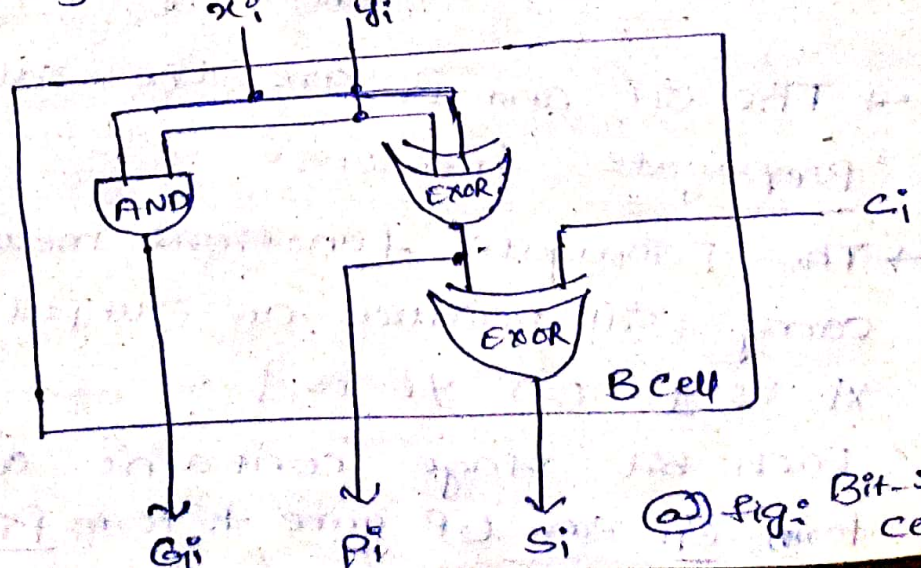
$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$

Here $c_{i+1} = G_i + P_i c_i$

$G_i = x_i y_i$

$P_i = x_i \oplus y_i$

$S_i = x_i \oplus y_i \oplus c_i$



(a) Fig: Bit-slice cell

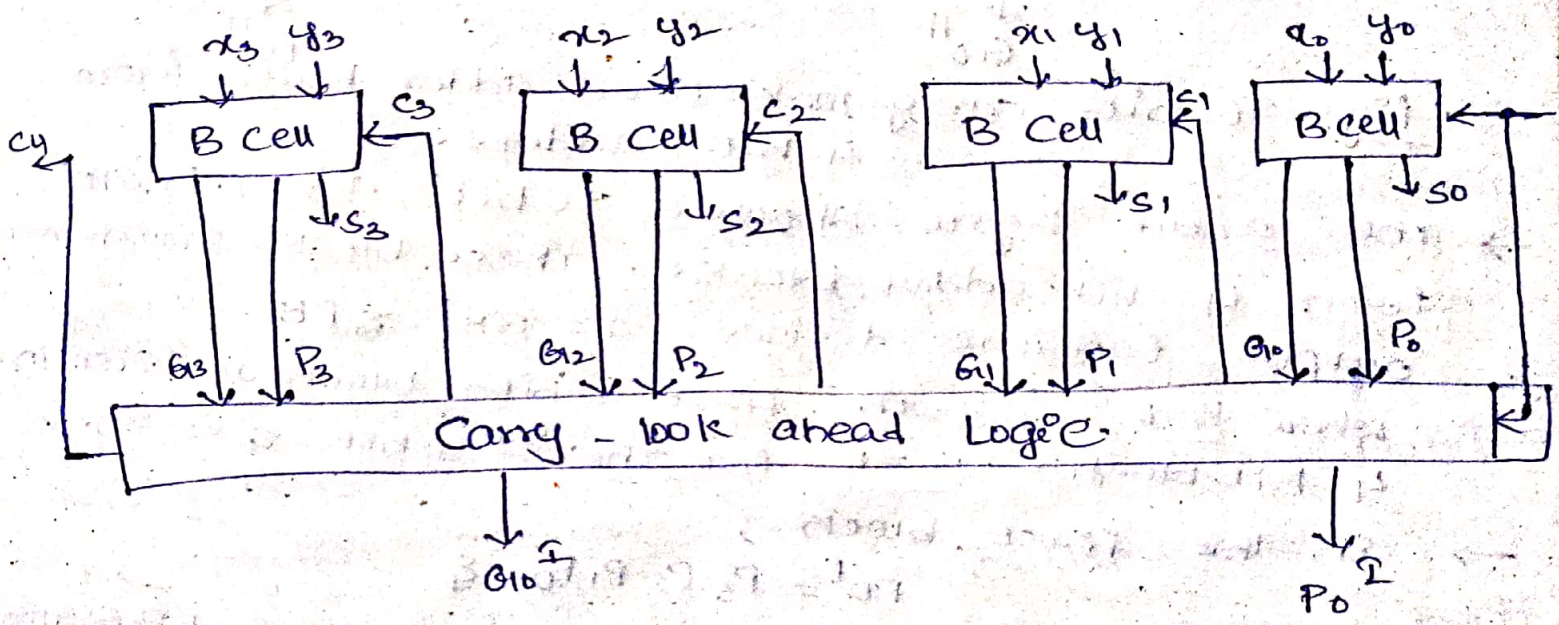
→ Advantages:-

1. Circuitary Reduce that's why it is better.
2. Only two pin (x,y) Gates instead of 3 pin gates only two inputs are allowed.
3. It reduces the propagation delay.

Drawbacks:

4-Gate Delay. [One AND Gate Delay & 1 OR Gate Delay & (ci, & 1 EXOR]

→ It gets more complicated as the no. of bits increase.



(b) 4-bit Adder

fig: 4-bit Carry-lookahead Adder.

→ The complete 4-bit adder, here the carries are implemented in the carry look ahead logic. An adder implemented in this form is called a carry-lookahead adder.

Drawbacks:

→ 8 gate Delay for carry bit & 4 gate delay for all sum bits.

Note: 4-bit ripple-carry Adder requires 7 gate delays for S_3 & 8 gate delays for C_4.

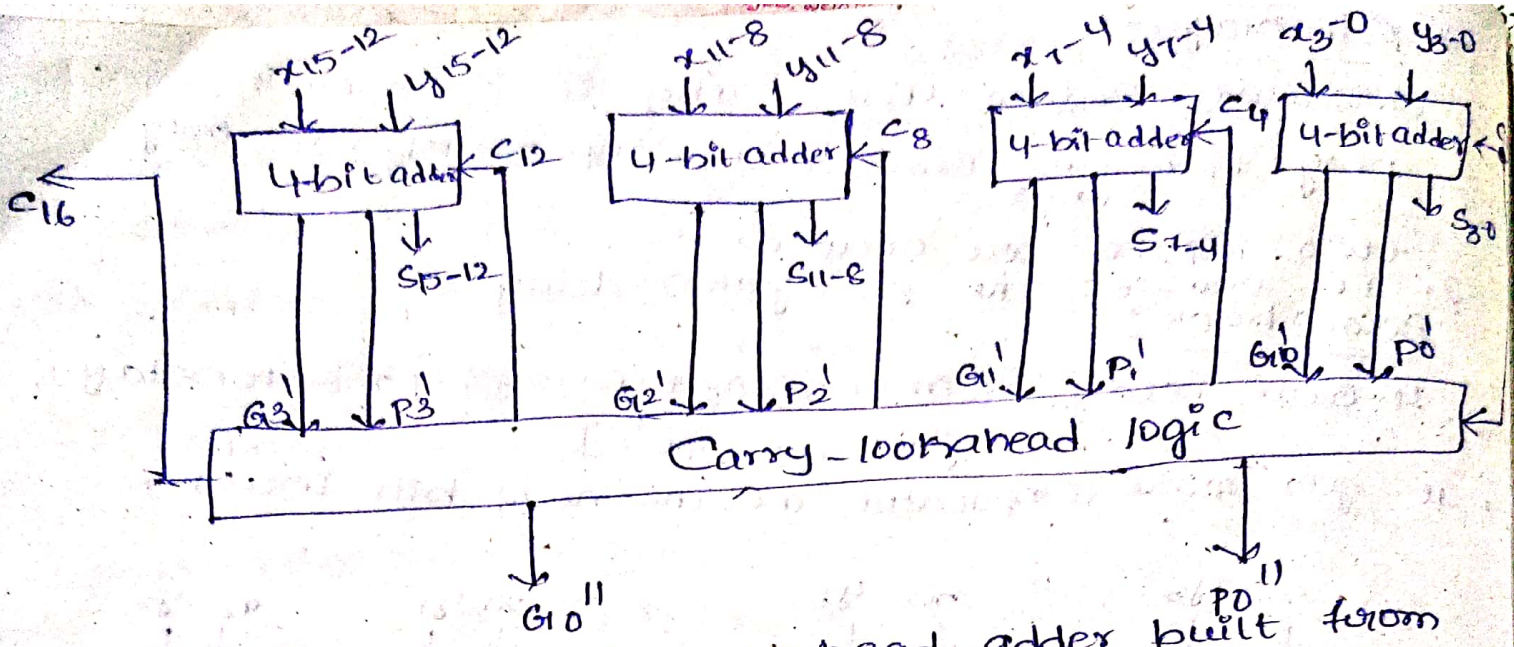


fig: 16-bit carry-lookahead adder built from 4-bit adders.

→ The above figure shows a 16 bit adder built from 4-bit adder blocks. These blocks provide new output functions defined as G_k' & P_k'.

→ where $k=0$ for the first 4-bit block as shown in 4 bit blocks. $k=1$ for the 2nd 4-bit & so-on.

→ In the first block,

$$P_0' = P_3 P_2 P_1 P_0 \text{ &}$$

$$G_0' = G_3 + P_3 G_2 + P_3 G_2 G_1 + P_3 G_2 G_1 G_0'$$

→ In the first level G_i & P_i functions determine whether bit stage i generates (or) propagates a carry. And

→ The 2nd level G_k' & P_k' functions determine whether blocks 'k' generates (or) propagates a carry.

→ The carry C_{16} is formed by one of the carry-look ahead circuits and given as,

$$C_{16} = G_3' + P_3' G_2' + P_3' P_2' G_1' + P_3' P_2' P_1' G_0' + P_3' P_2' P_1' P_0' C_0$$

Topic 3: Multiplication of Positive Numbers:

→ The Multiplication is a complex operation than addition and subtraction. It can be performed in H.W (or) S.W.

```
        1 1 0 1      (13)  Multiplicand (M)
      × 1 0 0 1      (9)   Multiplier (Q)
      ---------
        1 1 0 1
      0 0 0 0
    0 0 0 0
  1 1 0 1
  ---------------
  1 1 1 0 1 0 1    Final product.
```

} Partial product.

→ Multiplication process involves generation of partial products, one for each digit in the Multiplier.

→ In the binary system the partial products are easily defined when the Multiplier bit is 0, the partial product is 0, & when the Multiplier is 1, the partial product is Multiplicand.

→ The partial product is shifted one position to the left.

Register Configuration (or) H.W Implementation of unsigned Binary Multiplication:-

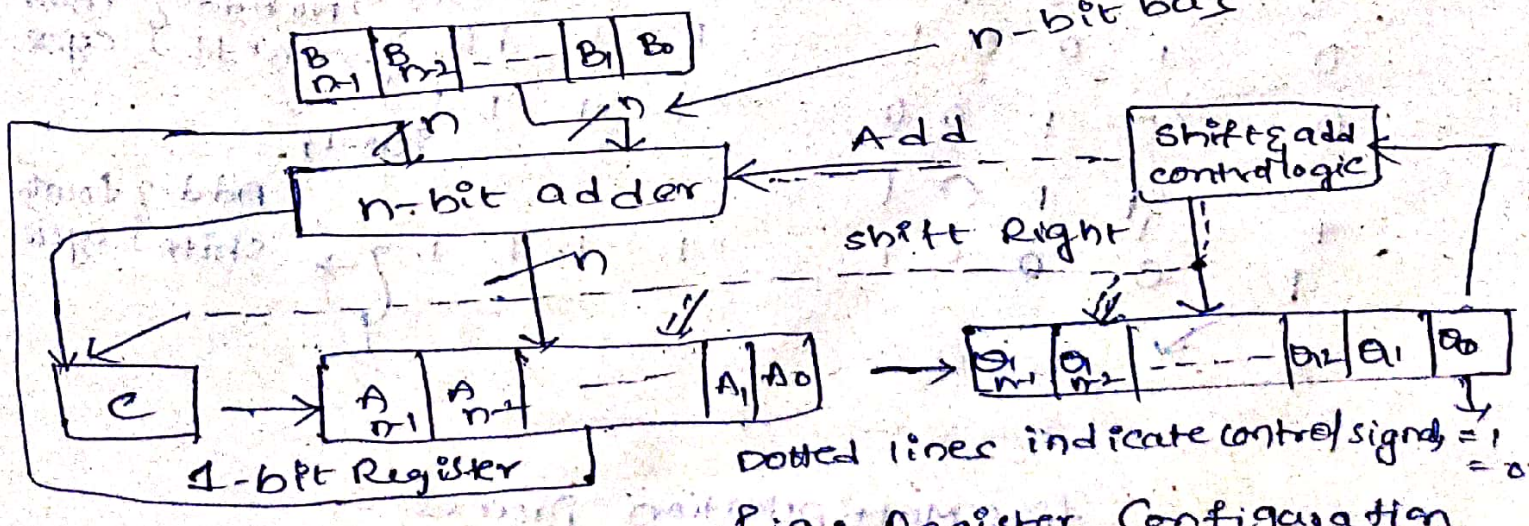


Fig: Register Configuration

→ Bit 0 of multiplier operand Q_0 of Q register is checked.

→ If Q_0 is one, then Multiplicand and partial Product are added & all bits C, A and Q Registers are shifted to the right one bit.

→ If Q_0 is zero, then no addition is performed only shift operation is carried out.

→ These steps are repeated until to get the desired results in the A and Q Registers.

Eg: Consider 4-bit Multiplier & Multiplicand

B → Multiplicand = 1101

Q → Multiplier = 1011

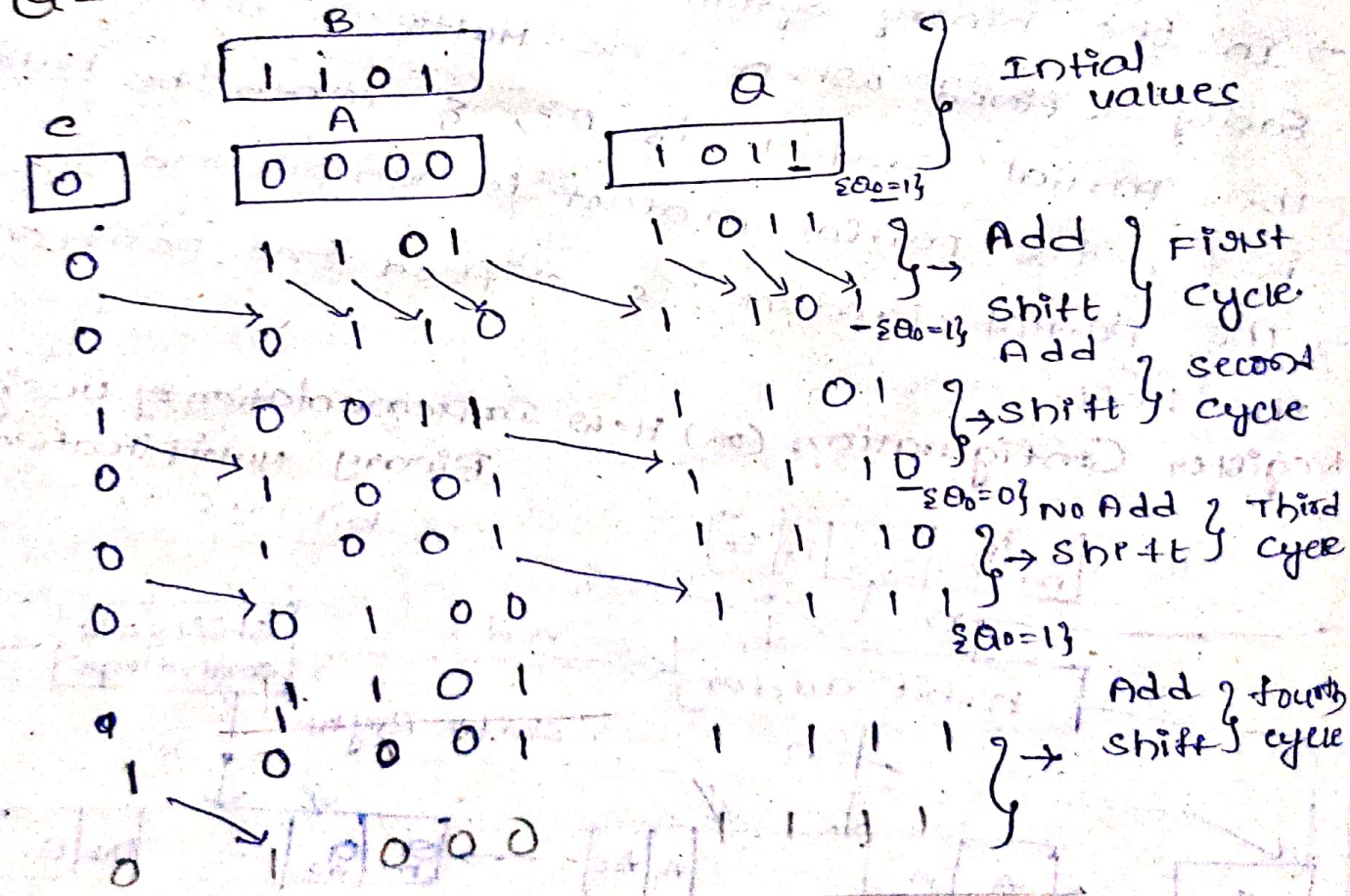


fig: Multiplication process.

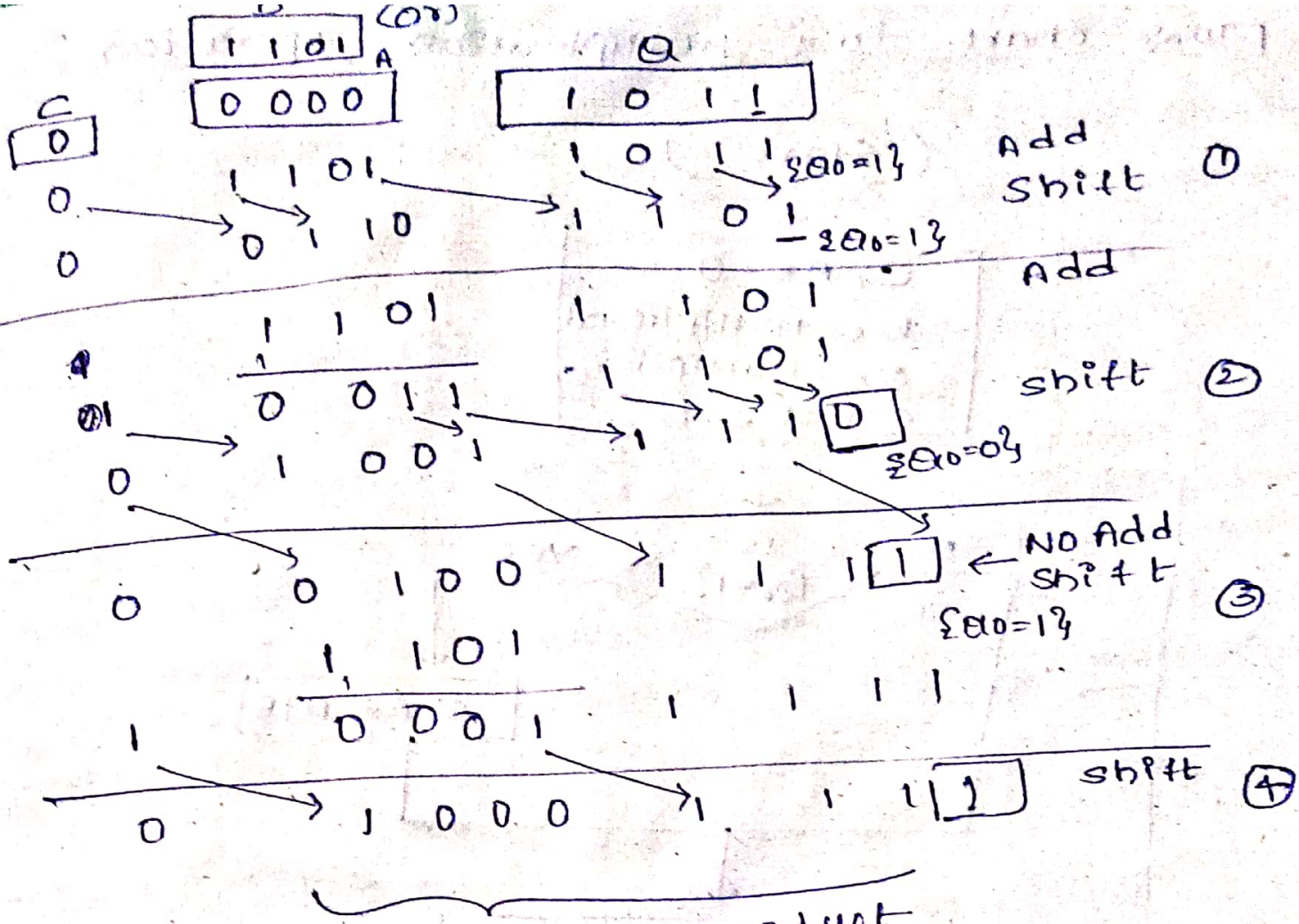


fig: Multiplication process

B - Multiplicand = $1101_{8421} = 13$

Q - Multiplier $= 1011 = 11$

$13 \times 11 = 143$

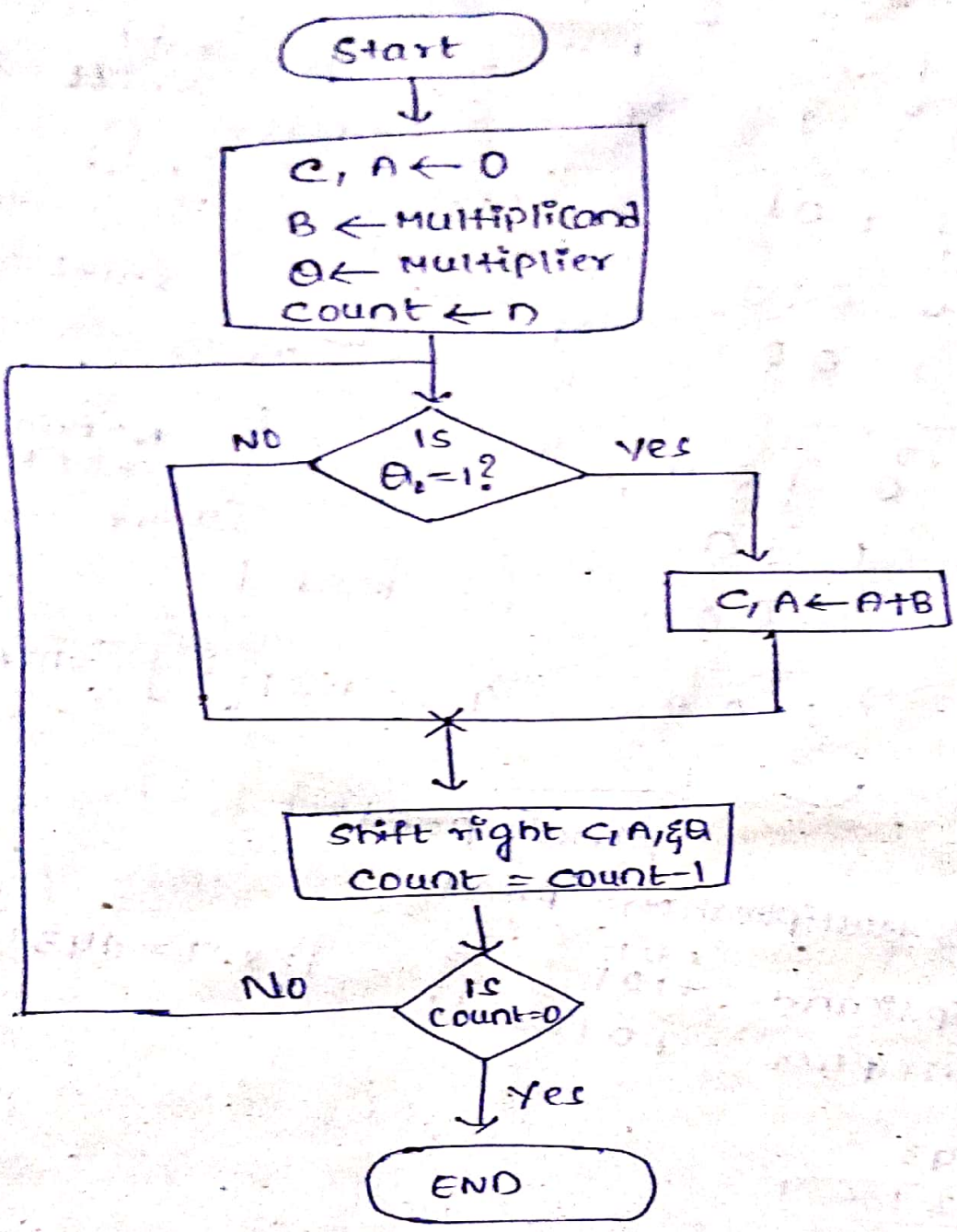
2	143
2	71 — 1
2	35 — 1
2	17 — 1
2	8 — 1
2	4 — 0
2	2 — 0
2	1 — 0
	0 — 1

1 0 0 0 1 1 1 1

128 64 32 16 8 4 2 1

$128 + 15 = 143$.

Flow chart to a Multiplication operation :-



Topic 4: Multiplication of Signed Numbers (or) Signed Operand Multiplication

→ In signed operand Multiplication, first consider the case of a positive Multiplier & Negative Multiplicand.

→ when we add a Negative Multiplicand to a Partial product, we must extend the sign-bit value of the Multiplicand.

→ For a Negative Multiplier is to form the 2's complement of both the Multiplier & the Multiplicand. And proceed as in the case of a positive Multiplier.

A technique that works equally well for both Negative & positive Multiplier called the Booth alg.

8 4 2 1

M → −13 → 1 1 0 1, 1's comp → 0 0 1 0,

B → +11 → 1 0 1 1

0 0 1 1

It is -ve Number so place '1' in front

(-13) → 1 0 0 1 1

(+11) → x 0 1 0 1 1 (+ve no so placed 0)

1 1 1 1 1 1 0 0 1 1

Sign 1 1 1 1 1 0 0 1 1

Extension 0 0 0 0 0 0 0 0 0

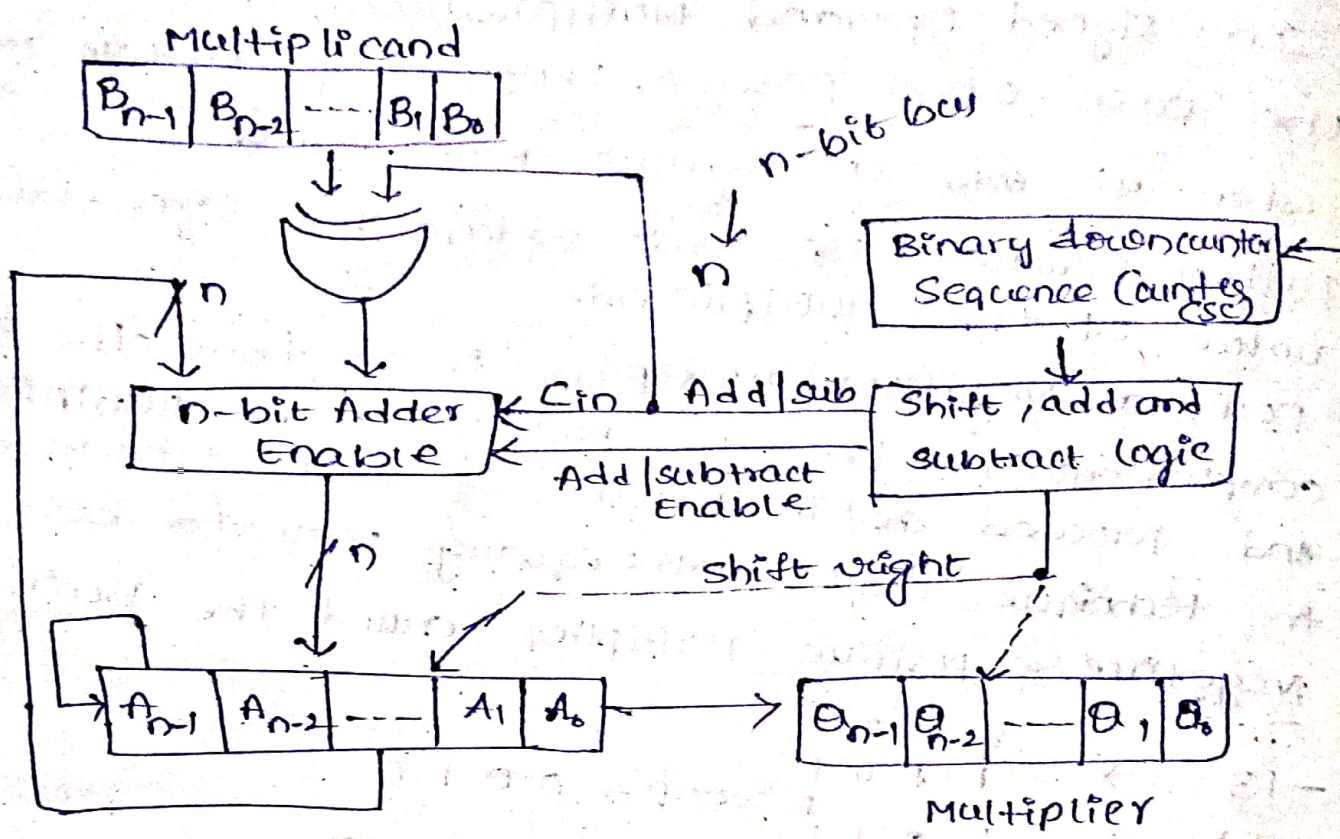
is shown. 1 1 1 0 0 1 1

0 0 0 0 0 0 1 1

1 1 0 1 1 0 0 0 1 (−143)

sign extension of Negative Multiplicand.

Sign Extension Of Negative Multiplicand (or)
How implementation of signed Binary Multiplication



Intially : $A \leftarrow 0$ & $Q_{-1} = 0$

→ The Booth's algorithm can be implemented as shown in above figure.

→ The circuit is similar to circuit for positive No Multiplication.

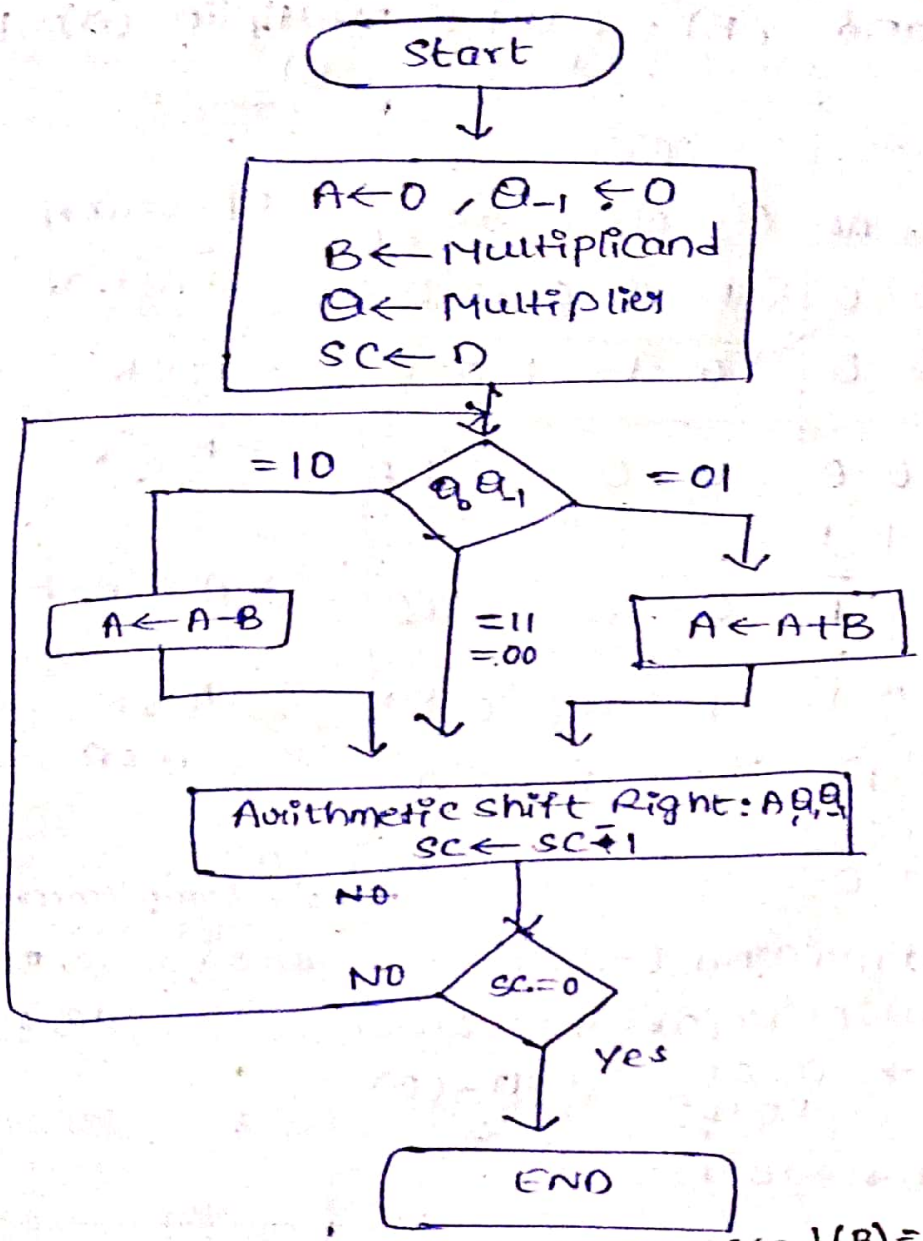
→ It consists of n-bit adder, shift, add subtract control Logic & four registers. A, B, Q & Q-1.

→ The Multiplier & Multiplicand are loaded into register Q and register B.

→ The register A and Q_{-1} are initially set to 0.

The sequence counter, SC is set to a Number n Equal to the Number of bits in the Multiplier.

Booth algorithm for signed Multiplication :-



$-B = \overline{1010} + 1 = 1011$ $(Q) = [7]$

case i) : Both positive Multiplicand (B) = 0101 [5] & Multiplier = 0100

SC	A	Q	Q-1	operation
	A_3 A_2 A_1 A_0	Q_3 Q_2 Q_1 Q_0		
	0 0 0 0	0 1 0 0	0	Initial
100	0 0 0 0	0 0 1 0	0	Arithmetic shift Right
011	0 0 0 0	0 0 0 1	0	ASR
010	0 0 0 0	0 0 0 1	0	ASR
	1 0 1 1			
	1 0 1 1	0 0 0 1	0	A ← A-B
001	1 1 0 1	1 0 0 0	1	ASR
				A ← A+B
				Res: 0001 0100

2. Negative Multiplier (5 × −4)

Multiplicand (B) = 0101 & Multiplier (Q) = 1100 (−4)

1010, 1011 (−B)

SC	A				Q					Operation
	A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
100	0	0	0	0	1	1	0	0	0	Initial
011	0	0	0	0	0	1	1	0	0	ASR
010	0	0	0	0	0	0	1	1	0	ASR
	⊕ 1	0	1	1						A ← A−B
	1	0	1	1	0	0	1	1	0	
001	1	1	0	1	1	0	0	1	1	ASR
000	1	1	1	0	1	1	0	0	1	ASR

Result = 1 1 1 0 1 1 0 0 ⇒ −20

(2's complement of 20)

16 8 4 2 1

00010, 01011

11101, 01100

3. Negative Multiplicand (−5) → B

Multiplier (Q) = 4 ⇒ (0100)

Multiplicand: B → 0101

1010

−B → 1011

A − (B)

0

SC	A				Q					Operation
	A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
100	0	0	0	0	0	1	0	0	0	Initial
011	0	0	0	0	0	0	1	0	0	ASR
010	0	0	0	0	0	0	0	1	0	ASR
	⊕ 0	1	0	1						A ← A−B
001	0	1	0	1	0	0	0	1	0	ASR
	0	0	1	0	1	0	0	0	1	
	1	0	1	1						A ← A+B
000	1	1	0	1	1	0	0	0	1	ASR
	1	1	1	0	1	1	0	0	0	−8

4. Both Negative (−5 × −4)

Multiplicand (B) = 1011 (−5)

Multiplier (Q) = 1100 (−4)

SC	A	Q	Q_{-1}	operation
	A_3 A_2 A_1 A_0	Q_3 Q_2 Q_1 Q_0		
	0 0 0 0	1 1 0 0	0	Initial
100	0 0 0 0	0 1 1 0	0	ASR
	0 0 0 0	0 0 1 1	0	ASR
011	0 1 0 1	0 0 1 1	0	A ← A − B
	0 1 0 1	1 0 0 1	1	ASR
010	0 0 1 0	0 1 0 0	1	ASR
001	0 0 0 1			

Result = 0 0 0 1 0 1 0 0 = +20

Topic 5: Fast Multiplication.

→ There are two techniques for speeding up Multiplication process.

→ In first technique guarantees that the maximum number of summands (version of the multiplicand) are must be added is n/2 for n-bit operands.

→ The second technique reduces the time needed to add the summands.

1. Bit-pair recoding of Multipliers
2. Carry-save Addition of summands.

1. Bit-pair recoding of Multipliers:

→ 1. for Fast Multiplication &
2. Speed the Multiplication process.

Bit-pair recoding Table

Multiplier bit-pair		Multiplier bit on the 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

(b) Table of multiplicand selection decision

Eg:- B = M = (13) = 0 1 1 0 1 [positive so first 0]

Multiplier - Q = (-6) = 1 1 0 1 0 [-ve 0110
so 1001
1 first] 11
1010

Take Multiplier value 1 1 1 0 1 0 [0 [Add '0' in LSB position]

MSB Position: 0 -1 -2

Place 1: 0 1 1 0 1. → Multiplicand

0 -1 -2

1 1 1 1 1 0 0 0 1 1 0 [2's complement of Multiplier and add 1]

0 1 1 0 1 → Multiplicand

0 −1 −2

+ 0 0 1 XXXB [2's complement of Multiplicand and Multiply 2[10]

1 1 1 1 1 0 0 1 1 0 → (2's)

1 1 1 1 0 0 1 1 → (2's)

0 0 0 0 0 0 ↓

1 0 0 ① ① ①

Mul 2's comp → 1 0 0 1 0

1

1 0 0 1 1

1 1 1 0 1 1 0 0 1 0

Ignore carry.

Cross verify. 13 × −6 = (−78)

64 32 16 8 4 2 1

1 0 0 1 1 1 0

0 1 1 0 0 0 1

① ①

1 1 1 0 1 0 0 1 0

Add 3 ones

Fig: Multiplication requiring only 1/2 summands.

Eg: Multiply B(Multiplicand) = −11 = 2's comp of 11 ⇒ (1)0 1 0 1 1

Multiplier → A = 27 = 0 1 1 0 1 1

1 1 0 1 0 1

5+1=

Take Multiplier value for Recoding

0 1 1 0 1 1 0

i+1 i i-1

+2 −1 −1

Multiplicand value is → 1 1 0 1 0 1 X

+2 −1 −1 (leave one bit)

(extend 6 more bit) 0 0 0 0 0 0, 0 0 1 0 1 0 (2's comp of B)

(last bit is 1 so Extending) 0 0 0 0 0 0, 1 0 1 1

1 1 1 0 1 0 1 0 B × (10)

1 1 1 0 1 0 1 0 1 1

2. Carry-save Addition of summands

→ Multiplication requires the addition of several summands.

→ A technique called carry-save addition (CSA) speeds up the addition process.

→ Consider the array for 4×4 multiplication shown in fig. with the first row consisting of just the AND gates that implement the bit products m_{30}, m_{20}, m_{10}, and m_{00}.

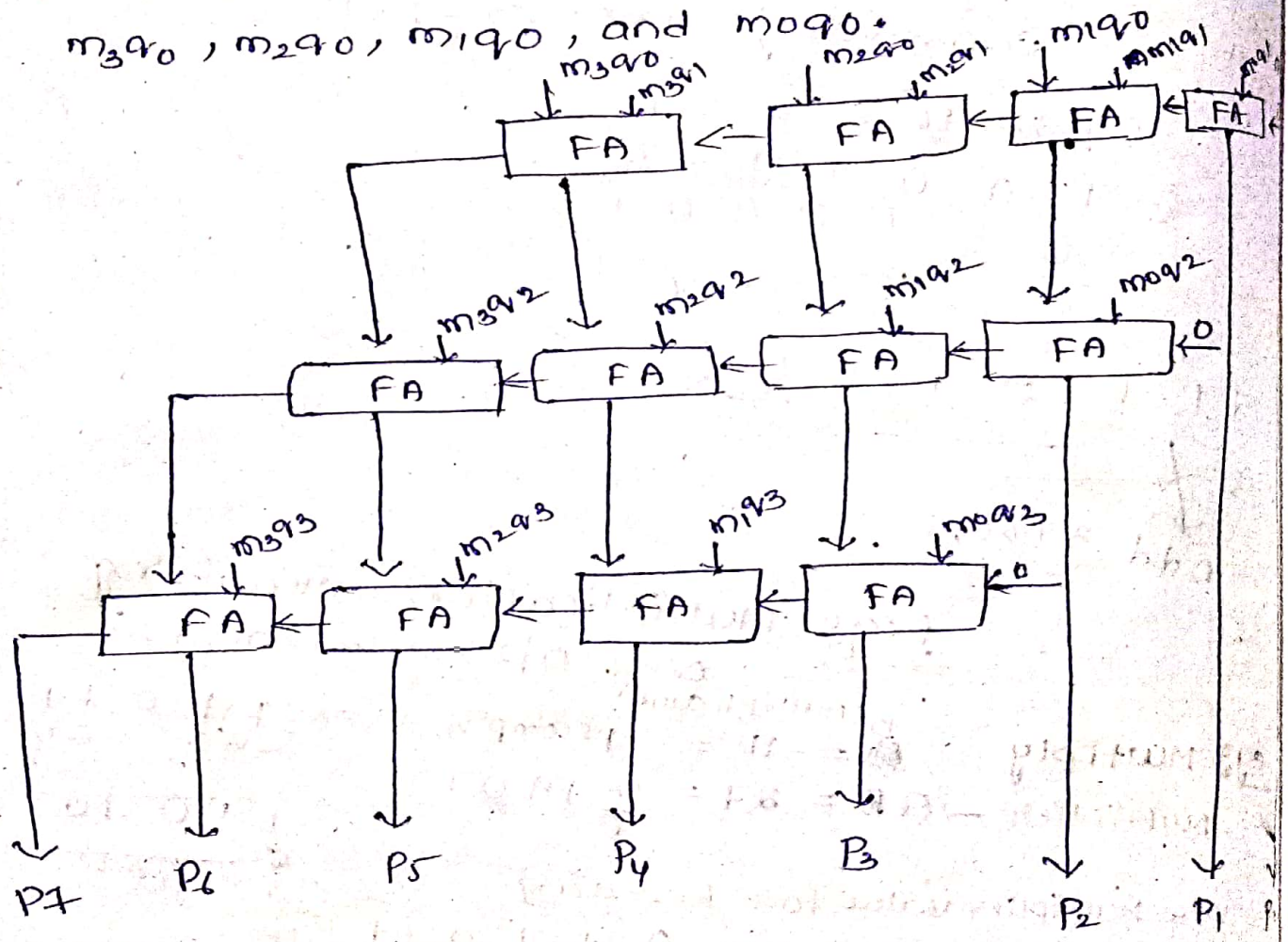


Fig:(a) Ripple Carry Array.

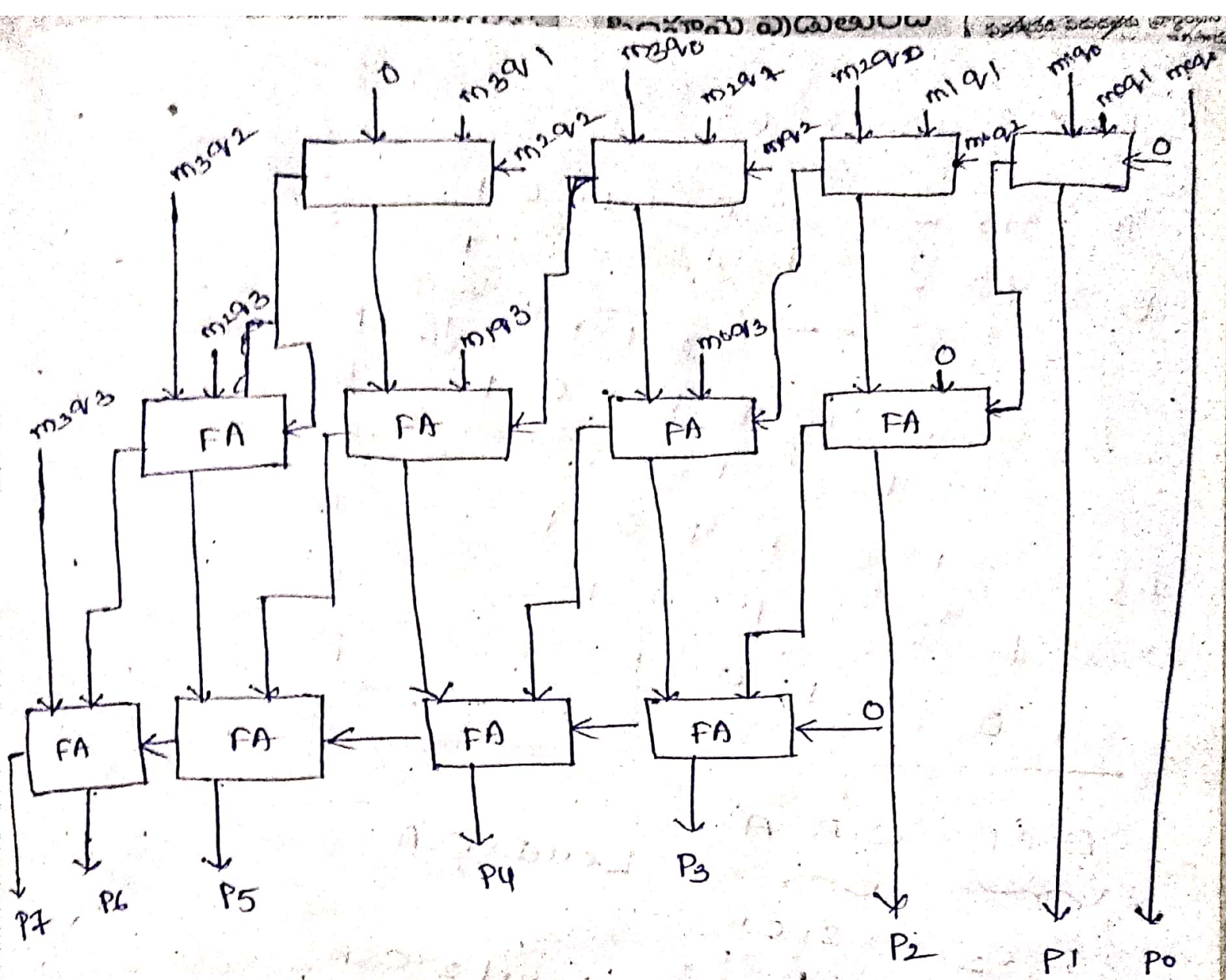


fig: (b) Carry-save array

The above fig (a) And (b) are Ripple-carry and Carry-save arrays for the multiplication operands, MX Q = P for 4-bit operands.

Ex: 45 × 63

B 45 — 1 0 1 1 0 1

Q 63 — 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	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	1			

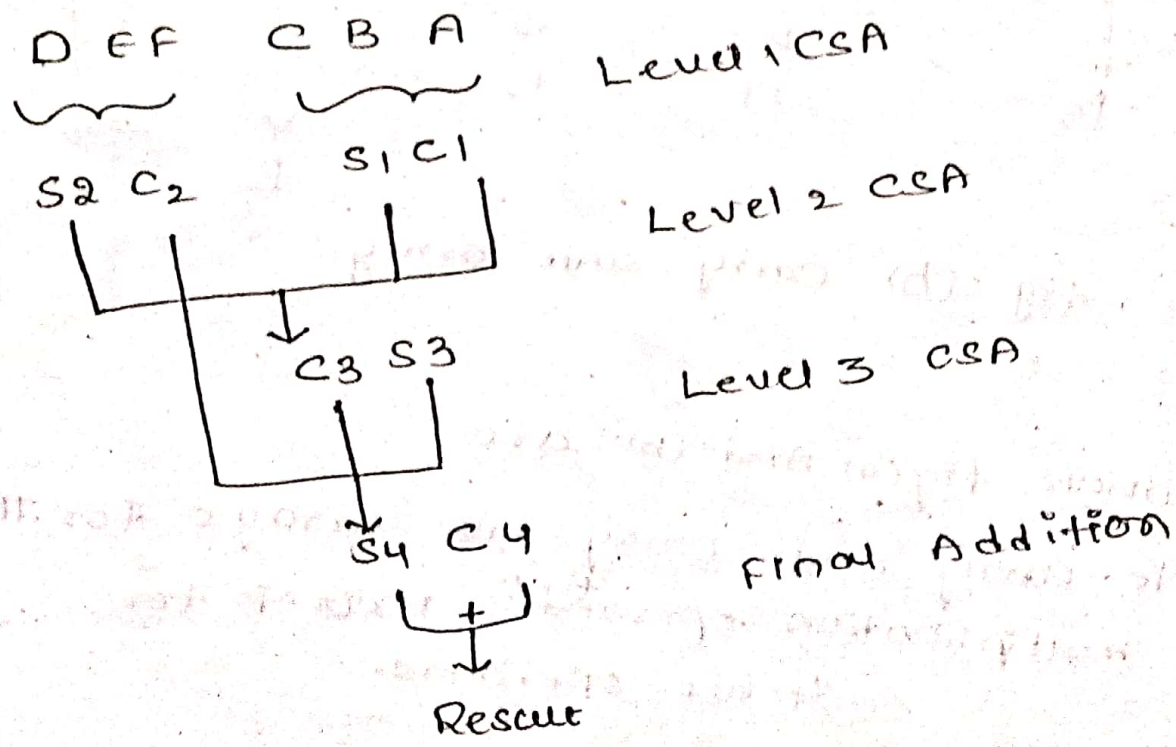


fig: Schematic Representation of the carry Save Addition.

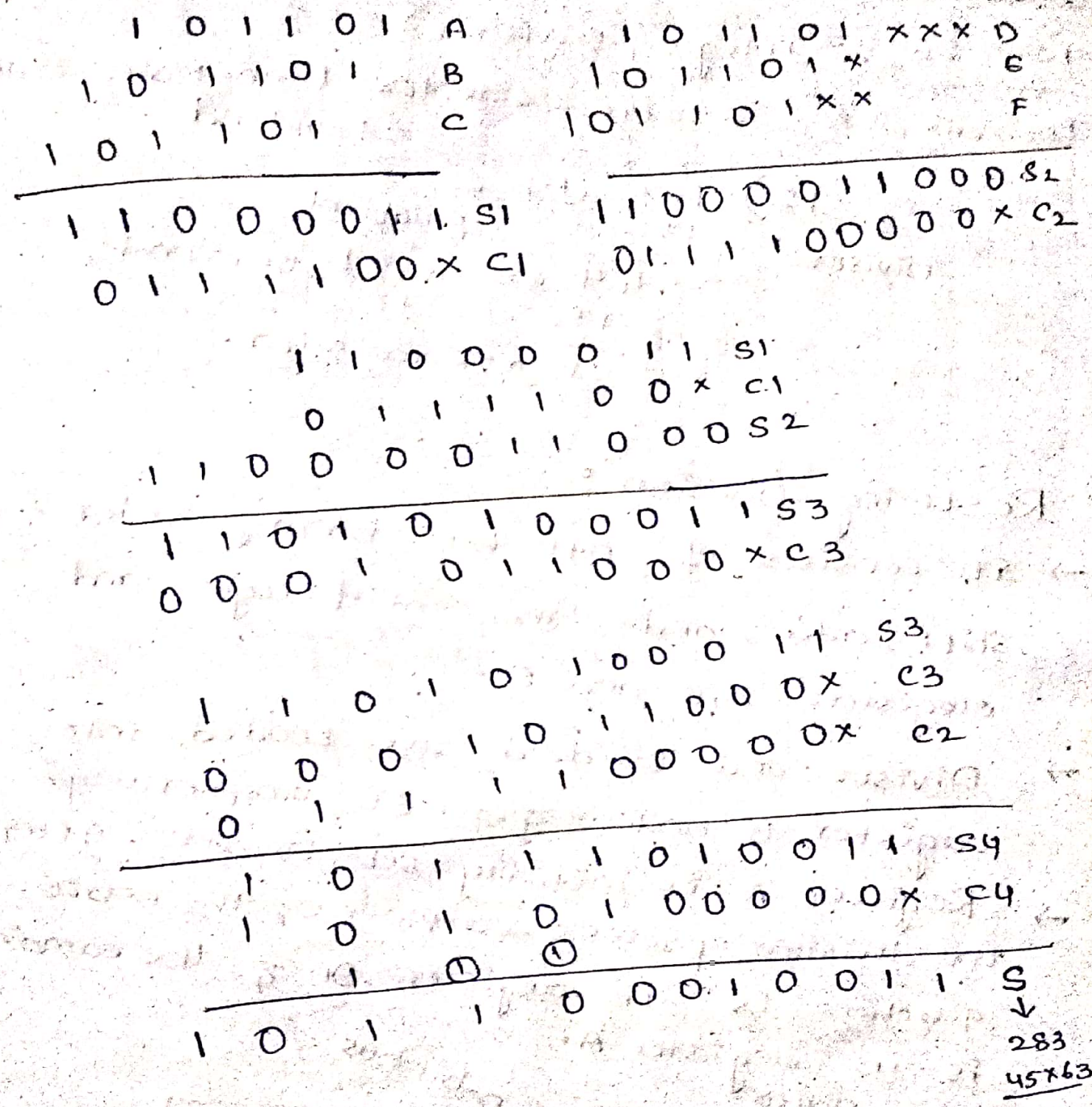


fig: The multiplication example ~~for~~ 45x63 performed using carry save Addition

Topic : 6 :- Integer Division

The division is more complex than Multiplication

Dividend ←

12) 169 (14 ← Quotient

→ Divisor

12

49 ← Partial remainder

48

1 ← Remainder.

Restoring Division :-

→ It consists of n+1 bit binary adder, shift, add, and sub control logic and registers A, B and Q.

→ Divisor and Dividend are loaded into register B and register Q respectively.

→ Register A is initially set to zero. After the division process completes, the n-bit quotient is in register Q & the remainder is in register A.

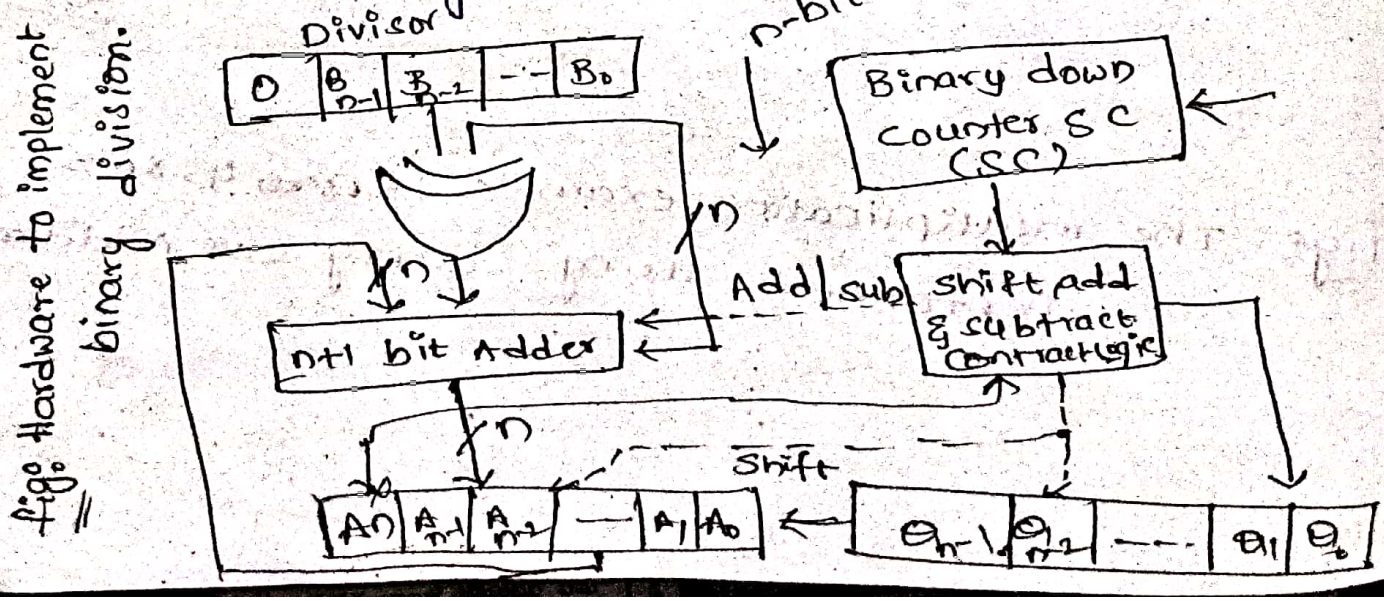


fig: Hardware to implement binary division.

Division Operation steps :- (Restoring division):

1. Shift A and Q left one binary position.
2. Subtract divisor from A & place answer back.
3. If the sign bit of A is '1', set Q_0 to 0 and add divisor back to A. Otherwise, set Q_0 to 1.
4. Repeat steps 1, 2, 3 n times.

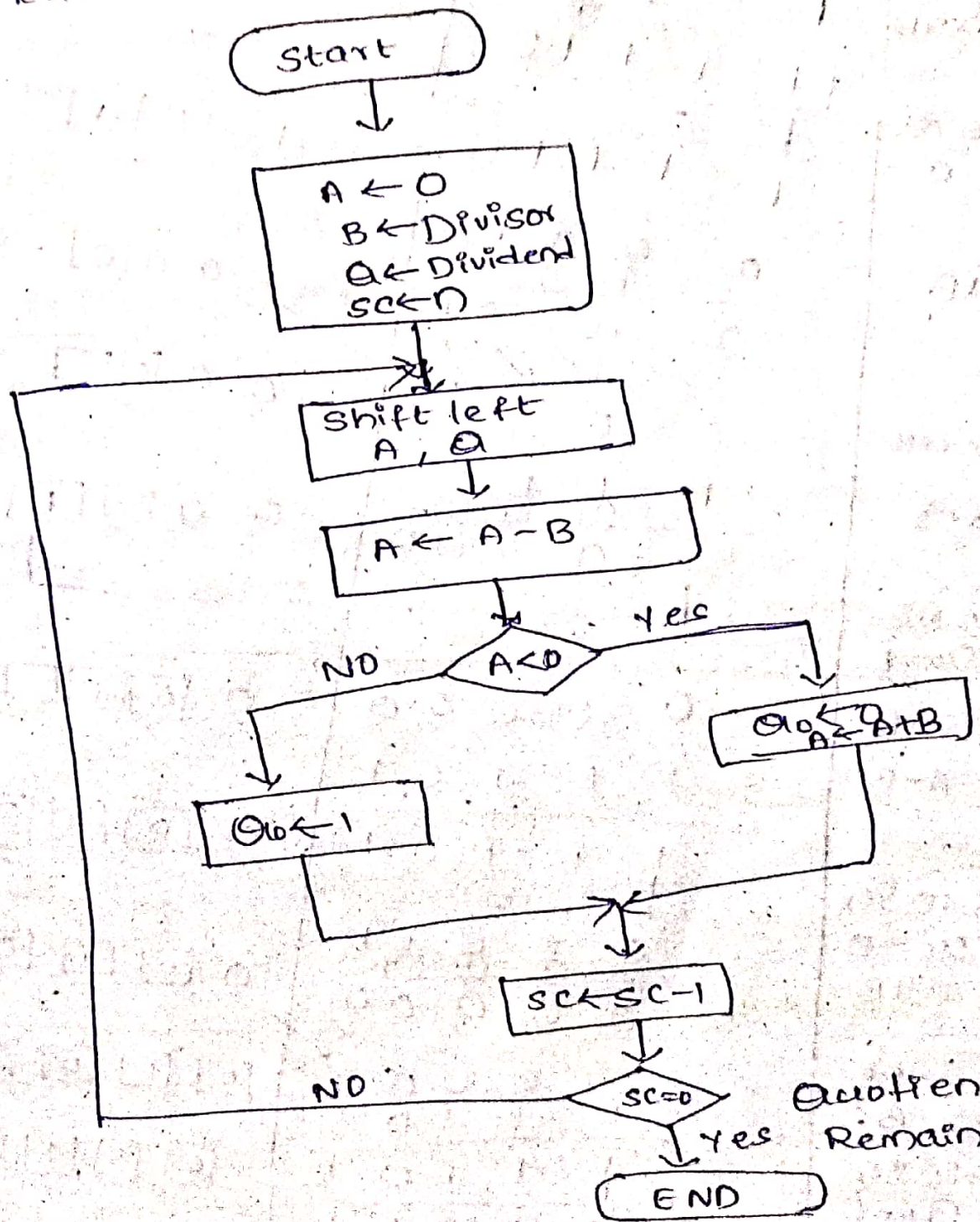


fig:- Flowchart for restoring division operation.

eg: Dividend = 1100 (A) (12)

Divisor = 0011 (B)(3)

1100
110

Q (dividend) SC

Steps	A	Q
Initial	0 0 0 0 0	1 1 0 0
Shift Left	0 0 0 0 1	1 0 0 □
A ← A − B, (Sub)	1 1 1 1 0	
A < 0 Yes, so $Q_0 = 0$	1 1 1 1 0	1 0 0 0
A ← A + B	0 0 0 1 1	
	0 0 0 0 1	1 0 0 0
Shift Left	0 0 0 1 1	0 0 0 □
A ← A − B	1 1 1 0 0	0 0 0 1
A < 0 NO, so $Q_0 = 1$	0 0 0 0 0	
Shift Left	0 0 0 0 0	0 0 1 □
A ← A − B	1 1 1 0 1	
A < 0 Yes, set $Q_0 = 0$	1 1 1 0 1	0 0 1 0
A ← A + B	0 0 0 0 0	0 0 1 0
Shift Left	0 0 0 0 0	0 1 0 □
A ← A − B	1 1 1 0 1	
A < 0 Yes, so $Q_0 = 0$	1 1 1 0 1	0 1 0 0
A ← A + B	0 0 0 0 0	0 1 0 0

Remainder in A.

A Restoring Division example. Quotient

Non-Restoring Division :-

1. If the sign of A is '0', shift A and Q left bit position & subtract divisor from A, otherwise shift A & Q Left and add divisor to A, if the sign of A is 0, set Q_0 to '1'; otherwise set Q_0 to '0'.
2. Repeat steps 1 & 2 for 'n' times.
3. If the sign of A is 1, add divisor to A.

A=0 shift, sub
A=1 shift, add
Note:

Non-Restoring division example :-

eg: Dividend = 1010 (10) → (Q)
Divisor = 0011 (3) → (B)

	sign bit A	A register	Q Register	SC
		0 0 0 0	1 0 1 0	1 0 0
Initially		0 0 0 0	1 0 1 0	
Shift left		0 0 0 0 1	0 1 0 □	→ 1st cycle
A ← A-B		1 1 1 0 1		
A<0 Yes		1 1 1 1 0	0 1 0 0	
Set Q0=0				
Shift		1 1 1 1 0	1 0 0 □	0 1 1
Sign<0				→ 2nd cycle
So Add B+A		0 0 0 1 1		
A<0 Yes		1 1 1 1 1	1 0 0 0	
Set Q0=0				
Shift		1 1 1 1 1	0 0 0 □	0 1 0
Sign<0				→ 3rd cycle
So Add B+A		0 0 0 1 1		
A<0 So Q0=0		0 0 0 1 0	0 0 0 1	
Shift left		0 0 1 0 0	0 0 1 □	0 0 0 4th cycle
Sign bit<0 Not		0 1 0 1 0 1		
So Subtract		0 0 0 0 1	0 0 1 1	

Dividend 10 (3 → Quotient
Divisor 3) 9 1
R

Remainder

Quotient

Differences b/w Restoring & Non-Restoring Division:-

Restoring Division	Non-Restoring Division
1. Needs restoring of register A if the result of the subtraction is negative.	1. Does not need restoring.
2. In each cycle content of Register A is first shifted Left and then divisor is subtracted from it.	2. In each cycle content of Register A is first shifted Left & then, divisor is added (or) subtracted with the content of register A depending on the sign of A.
3. Does not need restoring of Remainder.	3. Needs Restoring of Remainder if remainder is Negitive.
4. Slower Algorithm.	4. Faster Algorithm.

Topic 7: Floating point Nos & Arithmetic Operations:-

→ until now, we have discussed numbers with out any decimal point, fixed point numbers.

→ The decimal digit is always assumed to be the right of the Least significant digit.

→ To represent fractional binary numbers it is necessary to consider binary point.

→ If binary point is assumed to the right of the sign bit we can represent fractional binary number as given below.

$$B = b_0 \times 2^0 + b_{-1} \times 2^{-1} + \cdots + b_{-(n-1)} \times 2^{-(n-1)}$$

→ With this fractional number system we can represent the fractional numbers in the following range:

$$2^{-(n-1)} \leq F \leq 1 - 2^{-(n-1)}$$

→ If $n=32$, then the value range is approximately

$$2^{(-31)} \leq F \leq 1 - 2^{-(31)}$$

(i) $2^{-31} = -2.3283 \times 10^{-10}$

→ But this range is not sufficient to represent fractional numbers.

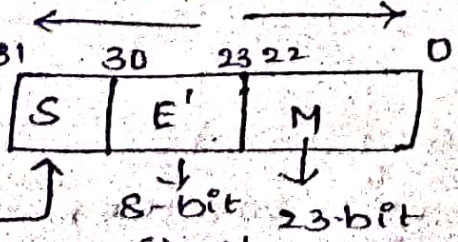
→ To accommodate very large integers and very small fractions, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted.

→ In this case, the binary point is said to float, and the numbers are called "floating-point numbers."

Floating point Representation:-

Floating point can be represented in two forms

(a) single-precision
↓
32 bits



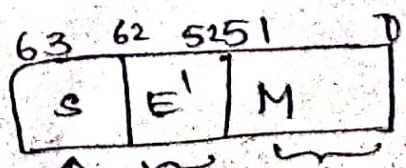
Sign
0 signifies +
1 signifies -

8-bit Signed exponent

23-bit mantissa fraction

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

(b) Double precision.
↓
64-bits



sign

11 bit excess

52 bit Mantissa fraction.

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

Eg: $(1460.125)_{10}$

$0.125 \times 2 \rightarrow 0.25 \rightarrow 0$

$0.25 \times 2 \rightarrow 0.5 \rightarrow 0$

$0.5 \times 2 \rightarrow 1.0 \rightarrow 1$

Top to bottom → 001

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

↓ Normalize

$1.0110110100001 \times 2^{10}$ → Normalized Result

Single precision :-

S = 0 → for positive

E → Exponent = 10

M → Mantissa (after decimal)

M = 01101101 00001

$E^1 = E + bias$

$E^1 = 10 + 127 = 137$

$E^1 = 137$

$E^1 = (137)_{10} = (10001001)_2$

S	E^1	M
0	10001001	01101101 00001

Double precision :-

S = 0

$E^1 = E + bias$

$= 10 + 1023$

$E^1 = (1033)_{10}$

$M = (10000001001)_2$

M = (10000001001)

S	E^1	M
0	10000001001	01101101 00001

Floating point Addition and subtraction:-

Consider two floating point numbers:

$X = m1 \cdot r^{e1}$ and

$Y = m2 \cdot r^{e2}$ Assume : $e1 > e2$.

Rules for Addition and subtraction:-

1. select the number with smaller exponent and shift its mantissa right, a no. of steps equal to the difference in exponents $|e2-e1|$.
2. set the exponent of the result equal to the larger exponent.
3. perform addition/subtraction on the mantissa and determine the sign of the result.
4. Normalize the result, if necessary.

Eg: Add single precision floating point numbers A and B, where A = 44900000H and B = 42A00000H.

→ In the first step, convert Hexa Decimal number to Binary.

Sol: Represent numbers in single precision format

	S	E	M
A =	0	1000 1001	001 0 0 0 ... 0
B =	0	1000 0101	010 0 0 0 ... 0

Exponent for A = 1000 1001 ⇒ 137

Actual exponent = $e2-e1$ = 137 − 127 (Bias)

$E' \neq E - 127$

= 10.

Exponent for B = 1000 0101

= 133

Actual Exponent = 133 − 127 (Bias) = 6.

∴ Number B has smaller exponent with difference 4. Hence its mantissa is shifted right by 4 bits as shown below.

Step 2: Shift Mantissa

Mantissa of B = 0 1 0 0 . 0 0 0 0 ---0

1st shift Mantissa of B = 0 0 1 0 0 0 0 0 --0

2nd " " B = 0 0 0 1 0 0 0 0 ---0

3rd shift Right " " = 0 0 0 0 1 0 0 0 --0

4th shift Right " " = 0 0 0 0 0 1 0 0 ---0

Shifted Mantissa of B = 0 0 0 0 0 1 0 0 ---0

Step 3: Add Mantissas

Mantissa of A = 0 0 1 0 0 0 0 0 ----0

Mantissa of B = 0 0 0 0 0 1 0 0 ---0

Mantissa of Result = 0 0 1 0 0 1 0 0 --0

As both numbers are positive, sign of the result

∴ Result = 0 1000 1001 00100100 --0

= 44920000H

→ We have seen addition process for floating point numbers. Similar process is used for subtraction of floating point numbers.

→ In subtraction, two mantissas are subtracted instead of addition and the sign of greater mantissa is assigned to the result.

eg: subtract single point precision floating point numbers A and B (A-B), where A = 44900000 H
B = 42A00000 H.

Sol:

Step 1: Represent numbers in single precision format

A = 0 1000 1001 0010 0000 ----0

B = 0 1000 0101 0100 0000 ----0

Exponent for A = 1000 1001
= 137

Actual exponent = 137 - 127 (Bias)
= 10

Exponent for B = 1000 0101
= 133

∴ Actual exponent = 133 - 127 (Bias)
= 6

∴ Number B has smaller exponent with difference 4. Hence its mantissa is shifted right by 4 bits as shown below.

Step 2: shift Mantissa

Shifted Mantissa of B = 0000 0100 ----0

Step 3: subtract Mantissa

Mantissa of A = 0010 0000 ----0

Mantissa of B = 0000 0100 ----0

2's → -B → 1's: 1111 1011 → 1111 1100

1's B → 1111 1100

0 0 010 0100 ----0

Mantissa for A is greater than mantissa for B. therefore sign of result is sign of A.

Result = 0 100,0100,1 000,1110,0 ----0.

= 448E0000 H.

Problems in Floating point Arithmetic :-

Mantissa Overflow :- The addition of two mantissas of the same sign may result in a carry-out of the MSB bit. If so, the mantissa is shifted right and the exponent is incremented. [E = EMAX]

Mantissa Underflow :- If the process of aligning mantissas, digits may flow off the right end of the mantissa. In such case truncation methods such as chopping, rounding are used. [M > Min]

Exponent Overflow :- Exponent Overflow occurs when a positive exponent exceeds the maximum possible exponent value. In some systems this may be designated as $+\infty$ or $-\infty$. [E < EMAX]

Exponent Underflow :- Exponent underflow occurs when a negative exponent exceeds the maximum possible exponent value. In such cases, the number is designed as zero. [E > Emin]

Flowchart & Algorithm for floating point Addition & Subtraction :-

Algorithm :- To write algorithm for floating point addition as follows :

Declare registers : AM [0 : M-1] ; For mantissa of A
BM [0 : M-1] ; For mantissa of B.
AE [0 : E-1] ; For exponent of A
BE [0 : E-1] ; For exponent of B
E [0 : E-1] ; For temporary storage Exponent.

step 1: Read Numbers: AM ← Mantissa of A
BM ← Mantissa of B
AE ← exponent of A
BE ← exponent of B

step 2: compare the two exponents: E = AE - BE

step 3: Equalize
if E<0 then
right shift AM;
E = E+1 and go to step3;
if E>0 then
right shift BM;
E = E-1 go to step3;

step 4: Add Mantissas: AM = AM + BM and
E = Max (AE, BE)

step 5: check exponent overflow
if overflow occurs during mantissa addition
then if E = Emax then Report overflow
else
right shift AM
E = E+1 go to step.

step 6: Adjust for zero result
if AM = 0 then E = 0; go to stop

step 7: Normalize result
if AM normalized then go to stop.

step 8: check exponent underflow
if E > Emin then
left shift AM;
E = E-1 go to step7

step 9: Stop.

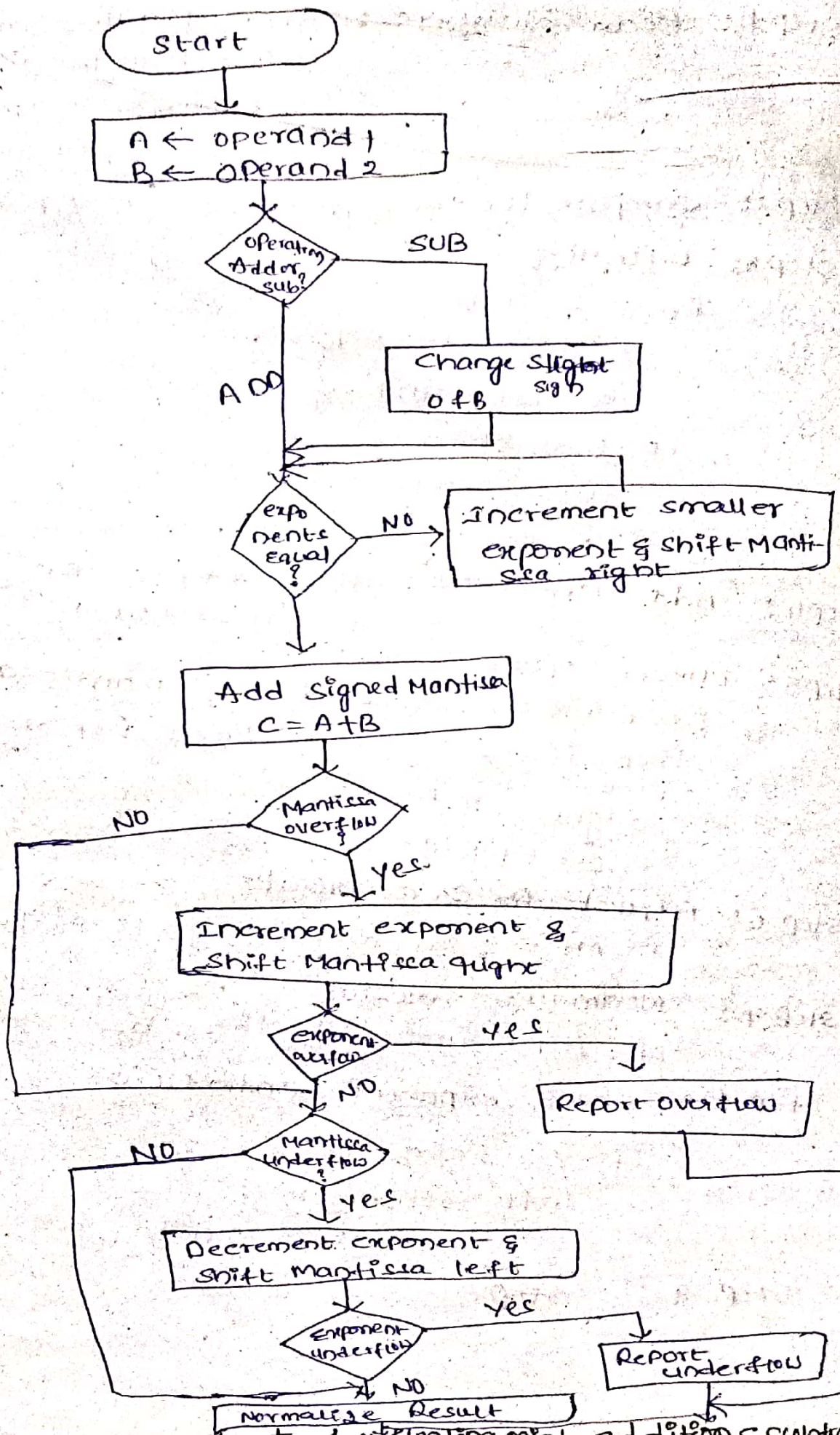
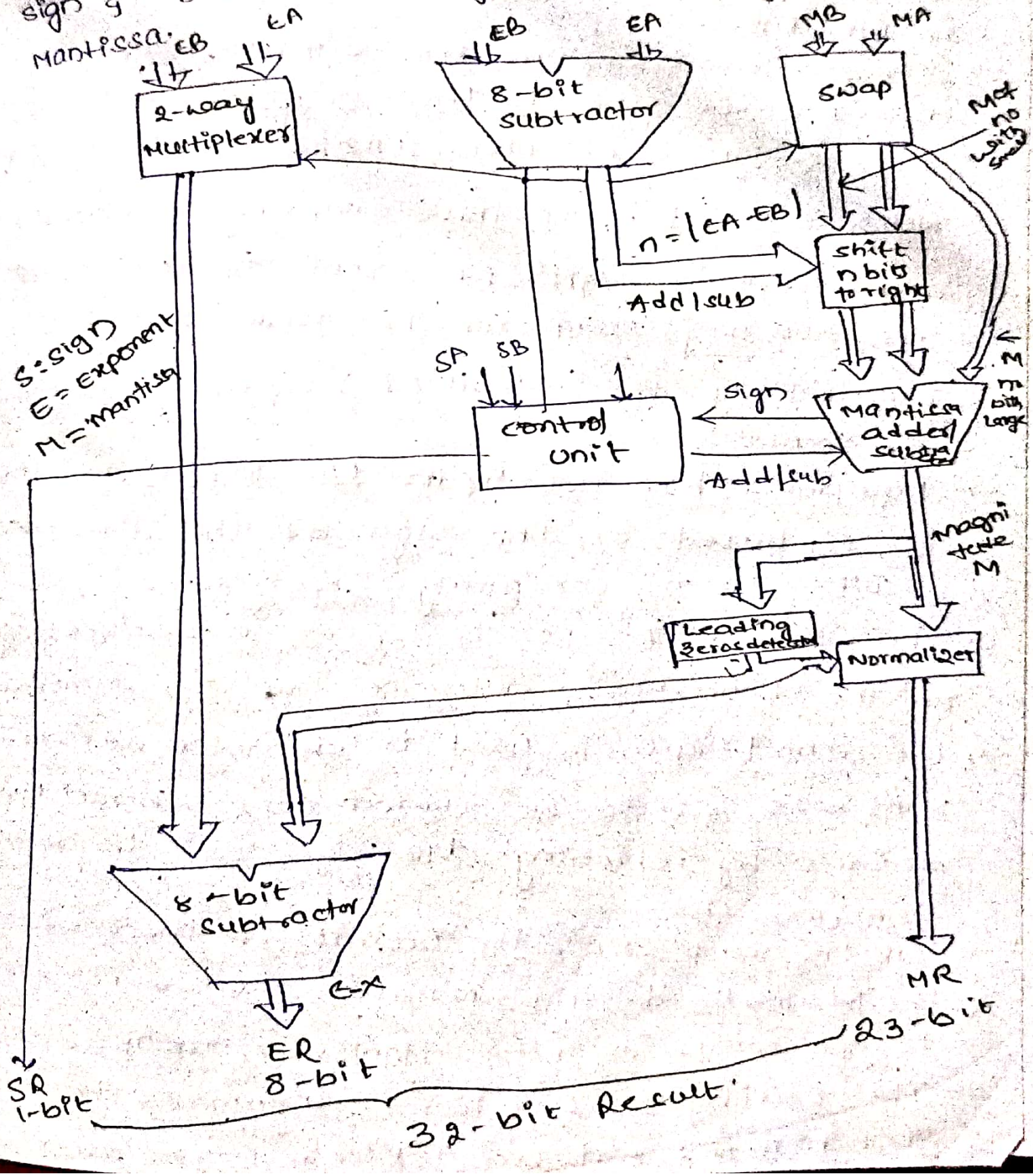


fig: flow chart for floating point addition & subtraction

Implementing Floating point operations (Add & sub) :-

→ It shows the hardware implementation for the addition & subtraction of 32-bit floating point operands.

→ It has the single precision format i.e. 1 bit for sign & 8-bits for signed exponent & 23-bit for Mantissa.



→ To compare the exponents of Numbers to determine a Number with a smaller exponent & then determine to shift the Mantissa of that Number with, so that its exponent Matches with the other Number.
→ The i/ps for the 8-bit subtractor are Exponent values of two Number & the operation performed is EA − EB.
→ The subtraction gives the difference 'd'. The difference is sent to the SHIFTER unit.
→ If the sign is 0, then EA ≥ EB and i/p to swap Network is 1. In this swapping is enabled and Mantissa MA is sent to the SHIFTER.
→ The two way Multiplexer is used to set the exponent of the result (E) Equal to the Larger Exponent.
→ Multiplexer has two inputs EA and EB and the o/p is based on the sign of the difference results from comparing Exponents.
→ The Next step is to perform addition & subtraction on the Mantissas & determine the sign of the Result.
→ The control logic is used to determine whether the Mantissas are to be added (or) subtracted, it decides by checking the signs of the operands (SA and SB)
→ In the Last step, the result of the Mantissa(M) is Normalized. The Normalized value is truncated to generate the 23-bit Mantissa, MR of the result
→ The Leading zeros detector determines the number of bit shifts, x to be applied to the Mantissa(M)