Computer Arithmetic

Addition and Subtraction of signed numbers, Design of fast adders, Multiplication of positive numbers, signed-operand multiplication, Fast multiplication. Integer division, Floating point numbers and operations

A basic operation in all digital computers is the addition or subtraction of two numbers. Arithmetic operations occurs at the machine instruction level. They are implemented along with basic logic functions such as AND, OR, NOT and Exclusive-OR (XOR) in the arithmetic and logic unit (ALU) of the processor.

Addition and Subtraction of signed numbers :-

The following table shows the logic truth table for the sum and carry out functions for adding equally weighted bits x_i and y_i.

x_i	y_i	Carry in c_i	Sum S_i	Carryout c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

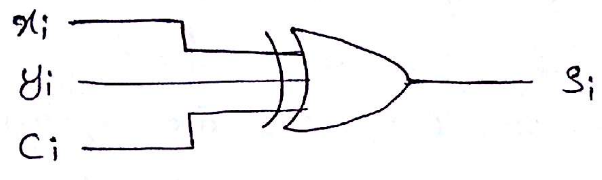
Scanned by CamScanner

where

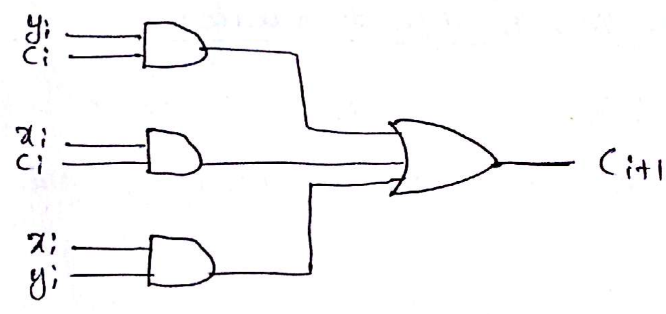
$$S_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

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

The sum expression can be implemented with a 3 input XOR gate like below.



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



The convenient symbol for the complete circuit for a single stage of addition called a "full adder" (FA).

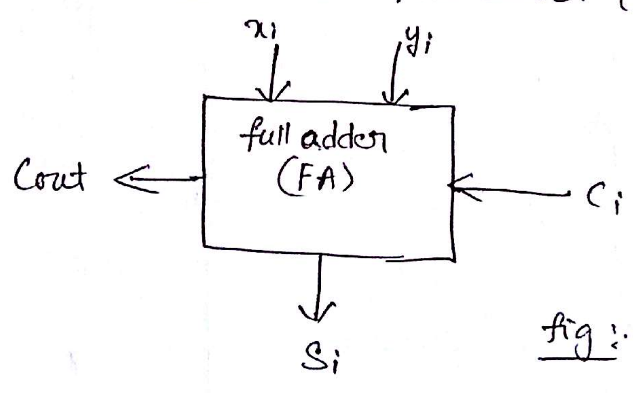


fig: logic for a single stage

A cascade connection of n full adder blocks 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.

Scanned by CamScanner

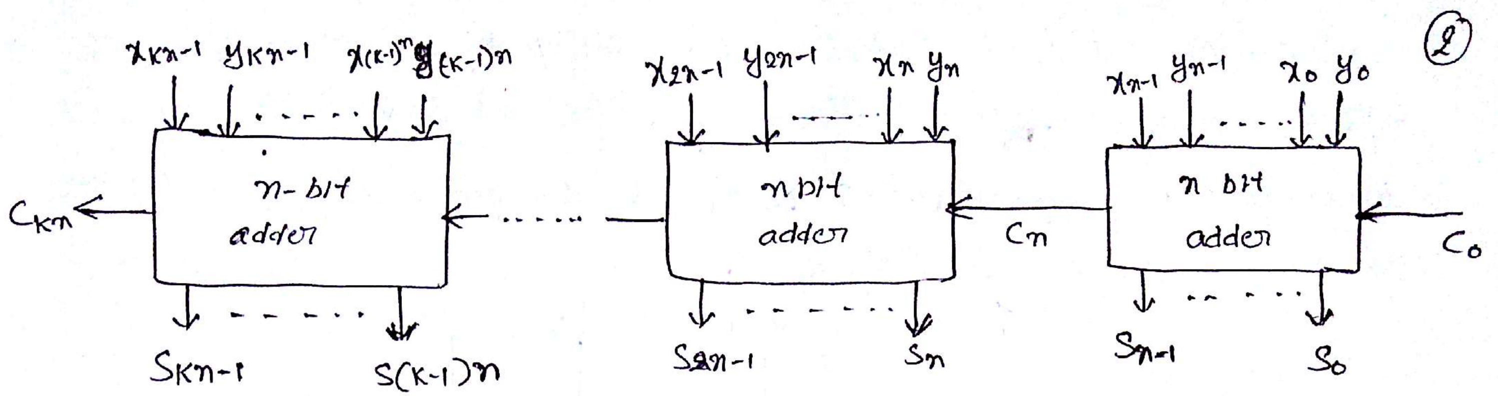


fig: Cascade of k n-bit adder

~~Addition/Subtraction Logic unit:~~

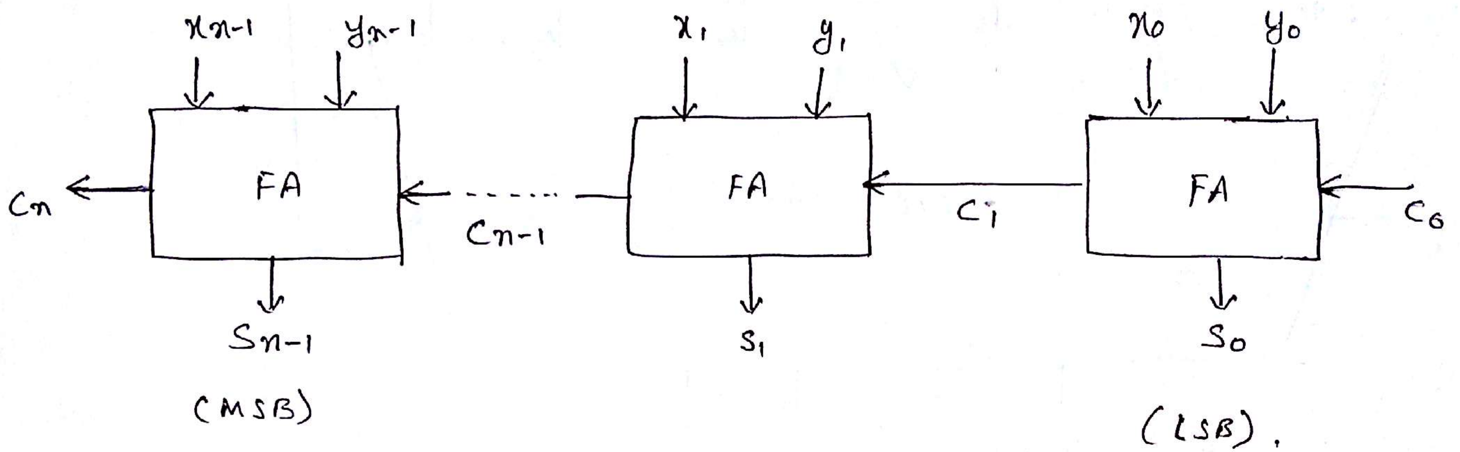


fig: An n-bit ripple-carry adder

Addition / Subtraction Logic Unit :-

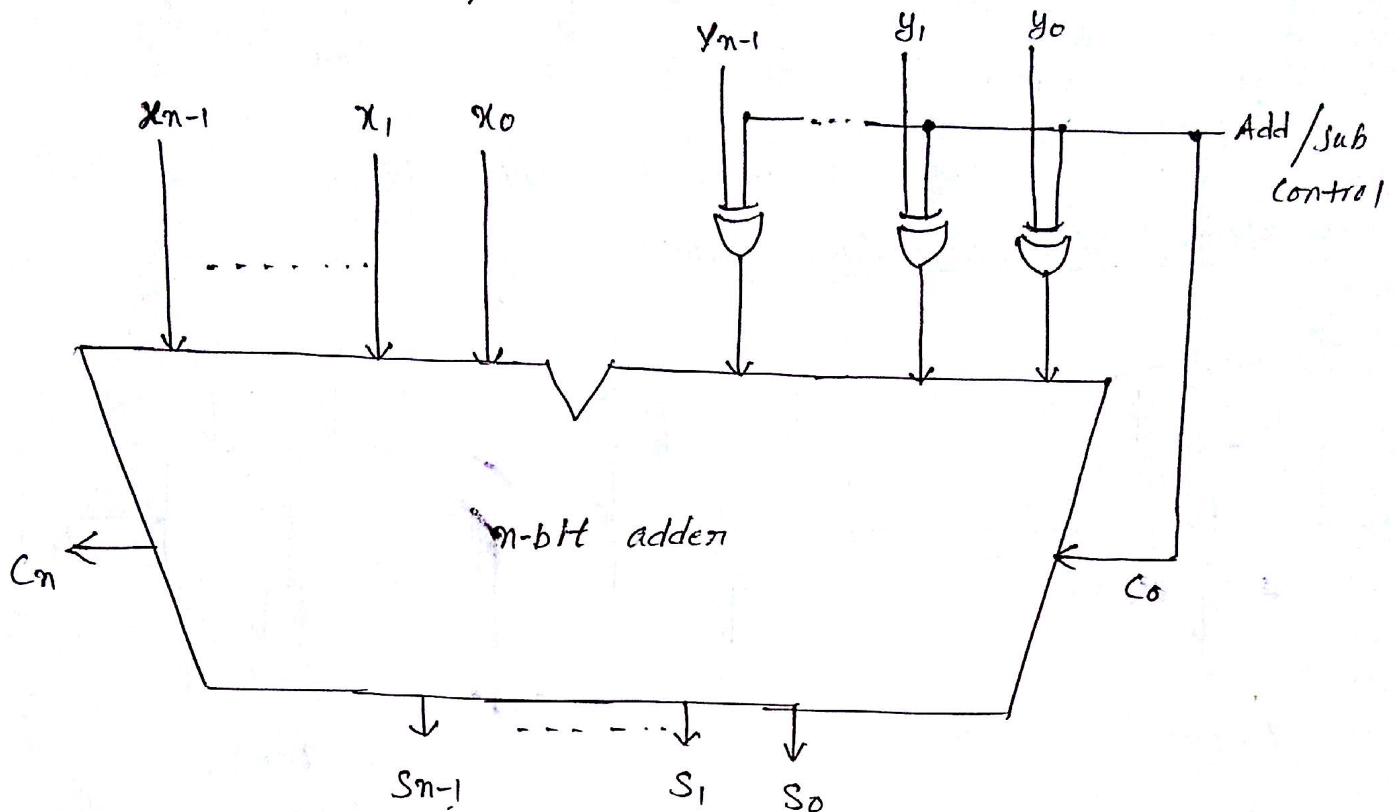
Overflow can only occur when the signs of the two operands are the same. An overflow may occur if the sign of the result is different. A circuit to detect overflow can be added to the n-bit adder by implementing the logic expression.

$$\text{Overflow} = x_{n-1} y_{n-1} \overline{S}_{n-1} + \overline{x}_{n-1} \overline{y}_{n-1} S_{n-1}$$

In order to perform subtraction operation X−Y, we takes the 2's complement of Y and add it to X. The following circuit diagram perform the either addition or subtraction based on the value of add/sub input control line.

Scanned by CamScanner

when the add/sub control line is set to 0 it will performs the addition operation. when the add/sub control line set to 1 it will takes the 1's complement of Y by the XOR gates and C_0 is set to 1 to get the 2's complement of Y.



DESIGN OF FAST ADDERS :-

If an n-bit ripple-carry adder is used in the addition/subtraction, it may take more time for producing outputs. weather to accept this delay or not depends upon the content of the speed of other processor components and the data transfer times of registers and cache memories. The delay through a network of logic gates depends on the integrated circuit electronic technology.

Using the logic implementation in logic for a single stage C_{n-1} is available in $2(n-1)$ gate delays, and S_{n-1} is correct one XOR gate delay later. The final carry-out C_n is available after $2n$ gate delays.

Scanned by CamScanner

Two approaches can be used to reduce delay in adders. The first approach is to use the fastest possible electronic technology in implementing the ripple-carry logic design. The second approach is to use an augmented logic gate network structure that is larger than parallel adder which has seen previously.

Carry-Lookahead Addition:-

One method of speeding up the process by eliminating inter stage carry delay is called "Lookahead-carry addition". It uses fast adder circuit to speed up. The logic expressions for S_i and C_{i+1} are

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

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

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

We can write this equation as

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

where $G_i = x_i y_i$ and $P_i = (x_i + y_i)$

Here the expressions G_i and P_i are called "generate" and "propagate" of carry. The generate function produces on carry when both x_i and y_i are one, regardless of the input carry. Each stage (bit stage) contains an AND gate to form G_i, an OR gate to form P_i and three input XOR gate to form S_i.

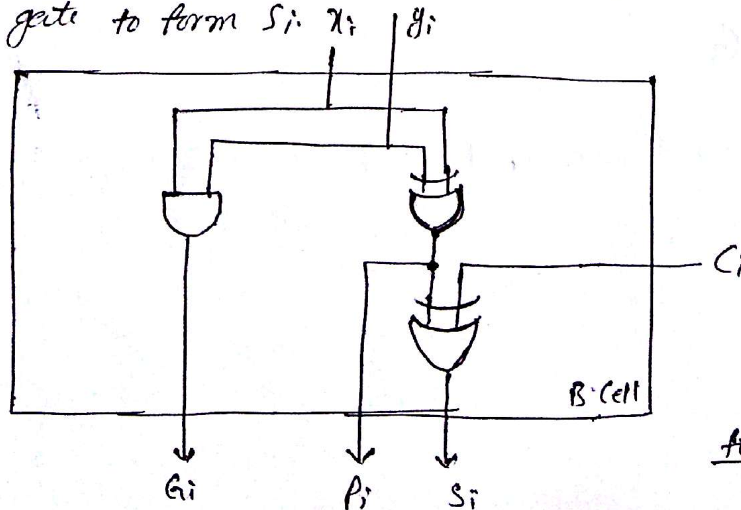


Fig: Bit-stage cell.

Now C_{i+1} can be expressed as a sum of products function of the P and G outputs of all the preceding stages.

$$C_1 = G_0 + P_0 C_{in}$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_1 P_2 G_0 + P_2 P_1 P_0 C_{in}$$

$$\vdots$$

The following diagram shows the general form of a carry-lookahead adder circuit

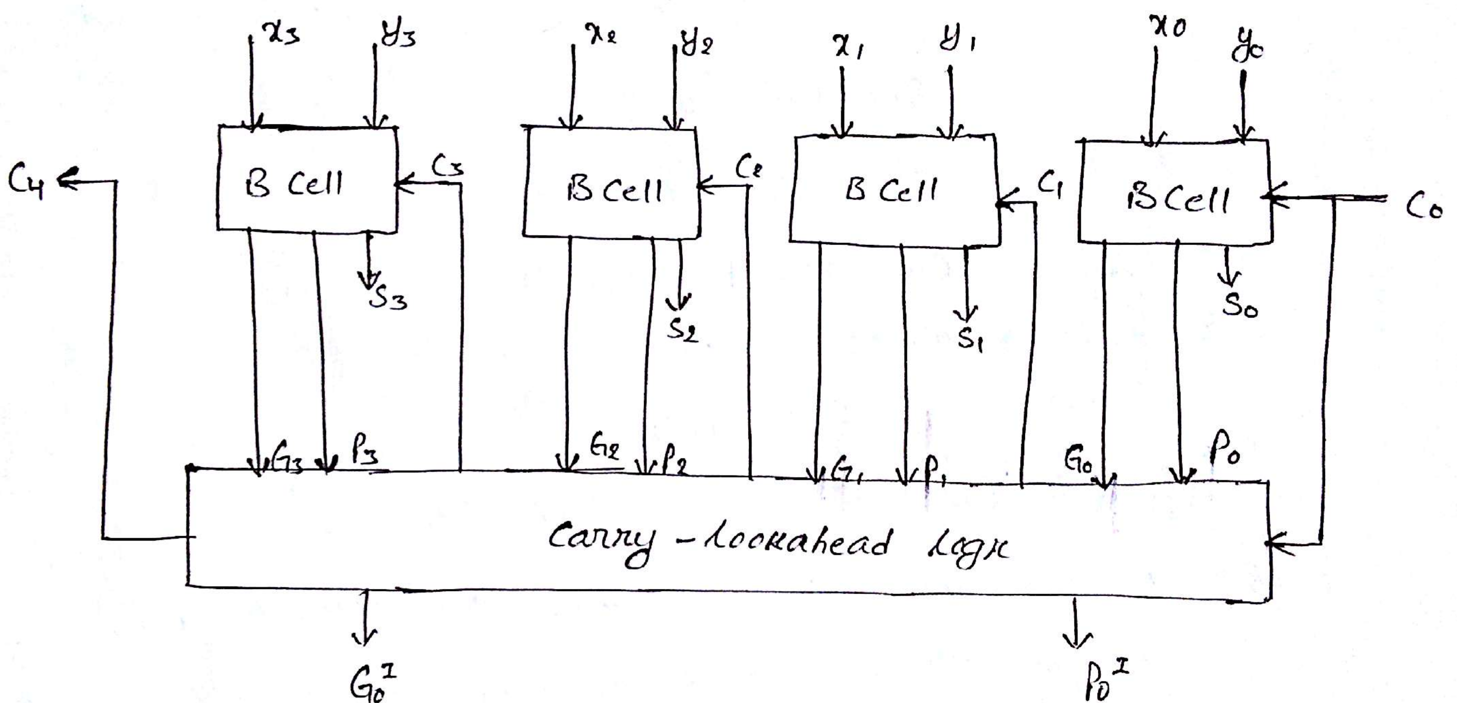


Fig: 4-bit carry-lookahead adder

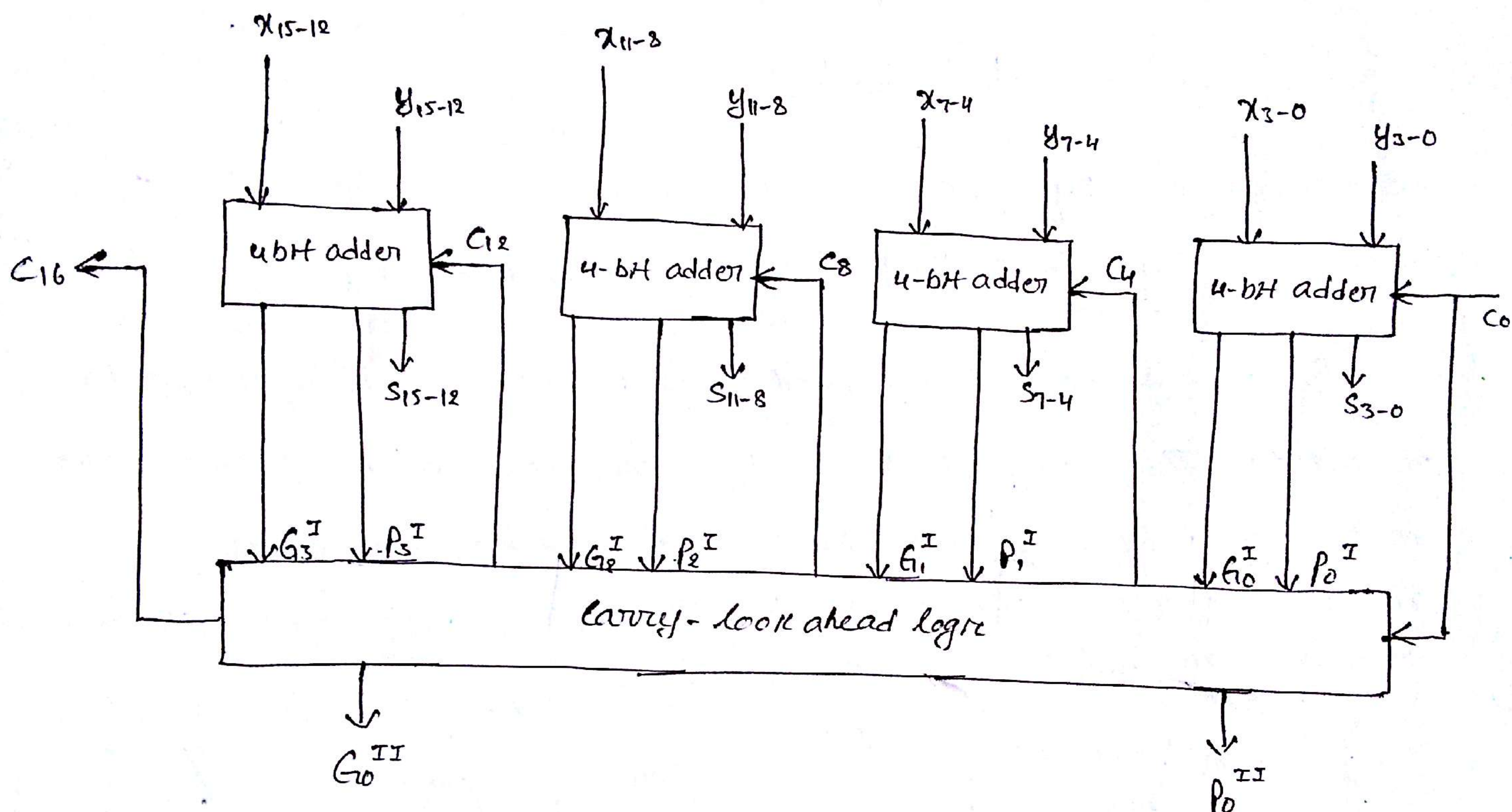
$$C_4 = G_3 + P_3 C_3$$

$$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$$

Scanned by CamScanner

High level Generate and Propagate function:-

Previously we have seen only 1 bit adder block in circuit. Same circuit we can implement for 4-bit adder which is high level. By using high-level block generate and propagate functions, it is possible to use the lookahead approach to develop the carries C_4, C_8, C_{12}.... in parallel.



In first block $P_0^I = P_3 P_2 P_1 P_0$.

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

C_{16} will be formed like below by using carry-look ahead logic

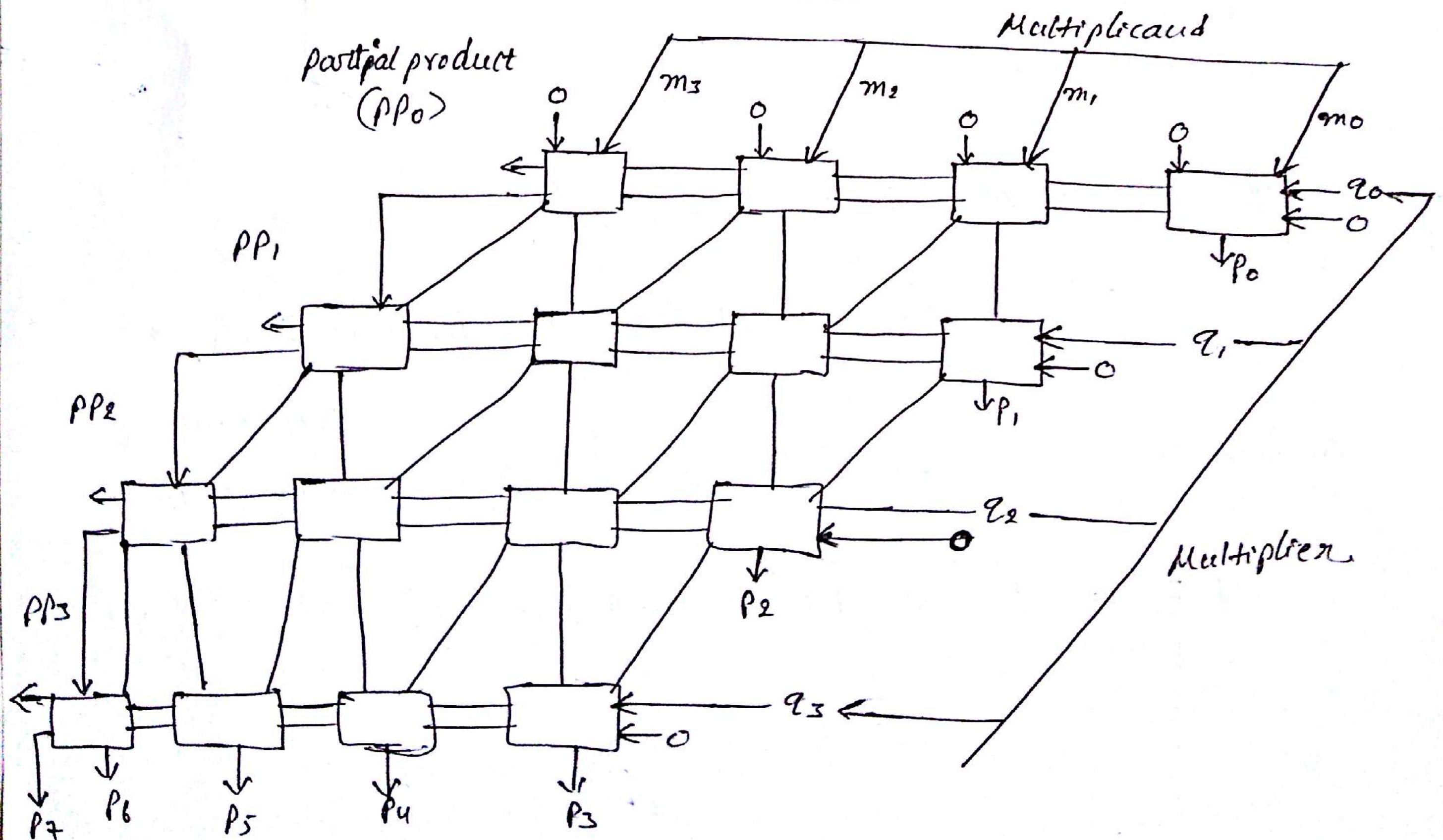
$$C_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^I G_0^I + P_3^I P_2^I P_1^I P_0^I C_0.$$

Multiplication of Positive Numbers:

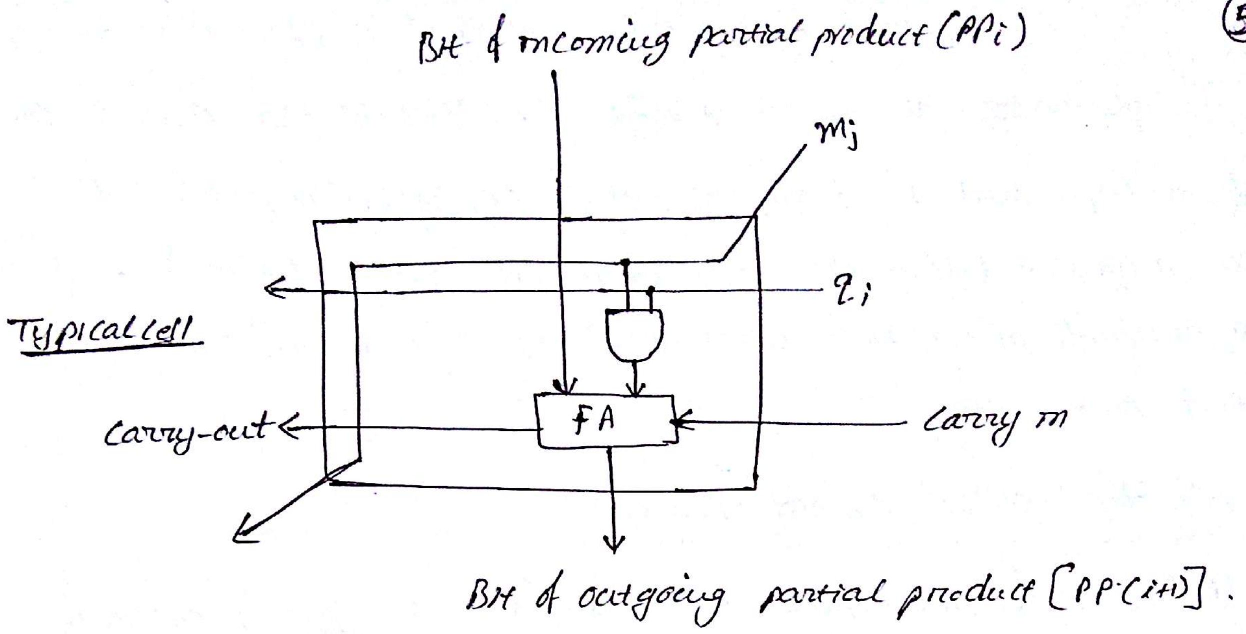
The usual algorithm for multiplying integers by hand is like below.

```
        1 1 0 1      (13) Multiplicand M
        1 0 1 1      (11) Multiplier Q
       ---------
        1 1 0 1
      1 1 0 1
    0 0 0 0
  1 1 0 1
 ---------------
 1 0 0 0 1 1 1 1     (143) Product P.
```

This algorithm can applied for unsigned numbers and positive numbers. The product of two n-digit numbers can be accommodated in 2n digits. Binary multiplication of positive operands can be implemented in a combinational two-dimensional logic array. The main component in each cell is a full adder FA.



Scanned by CamScanner



The simplest way to perform multiplication is to use the adder circuitry in the ALU for a number of sequential steps. The following diagram shows the hardware arrangement for sequential multiplication.

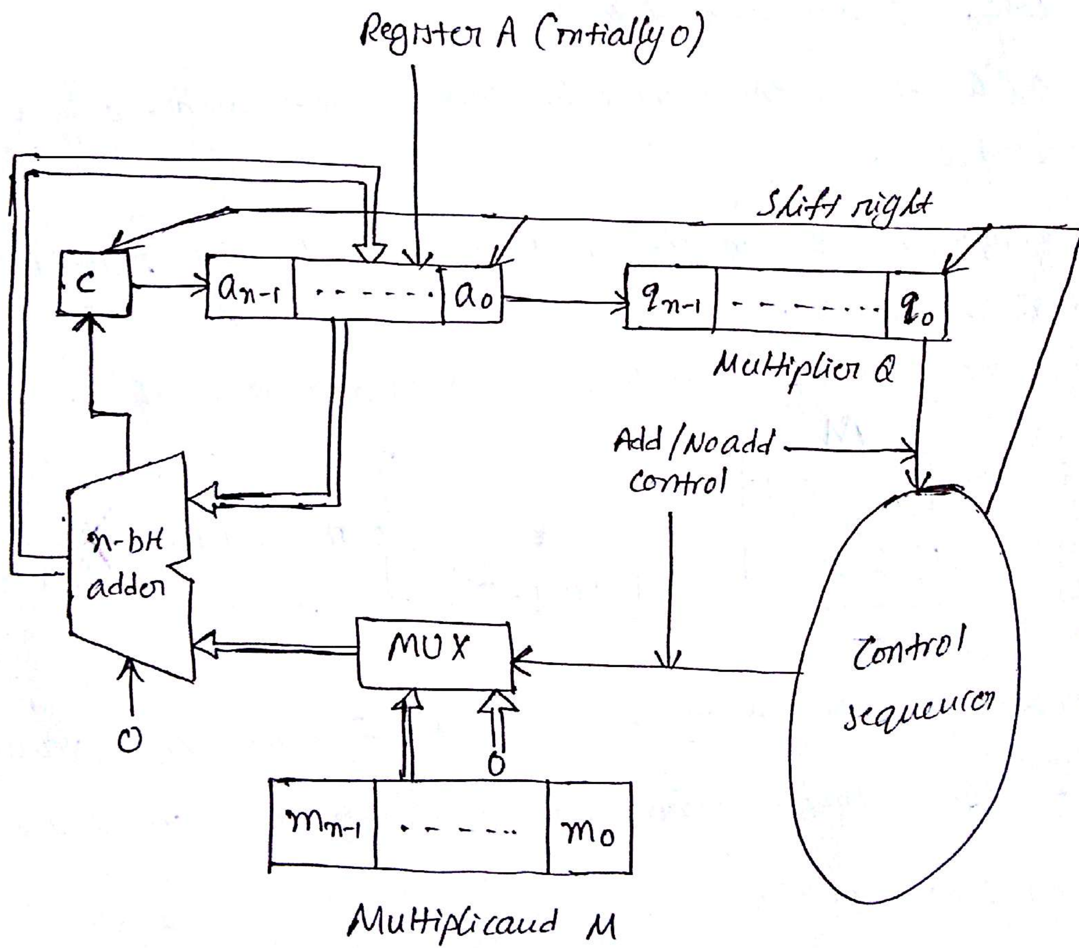


fig: Register configuration

Scanned by CamScanner

Registers A and Q combined hold PP_i while multiplier q_i generates the signal Add/NoAdd. This signal controls the addition of multiplicand M to PP_i to generate $PP(i+1)$. The product is computed in n cycles. Using the sequential hardware structure it is clear that a multiplication instruction takes much more time to execute than an Add instruction.

Multiplication operation steps :-

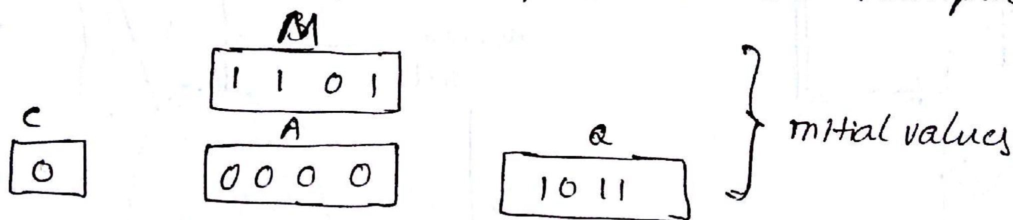
① Bit 0 of multiplier operand (Q_0 of Q register) checked.

② If bit 0 (Q_0) is 1 then multiplicand and partial product are added and all bits of C, A and Q register are shifted to the right one bit position. So the C bit goes into A_{n-1}, A_0 goes into Q_{n-1} and Q_0 is lost. C bit cleared to zero

③ If bit 0 (Q_0) is 0, then no addition is performed, only shift operation is carried out.

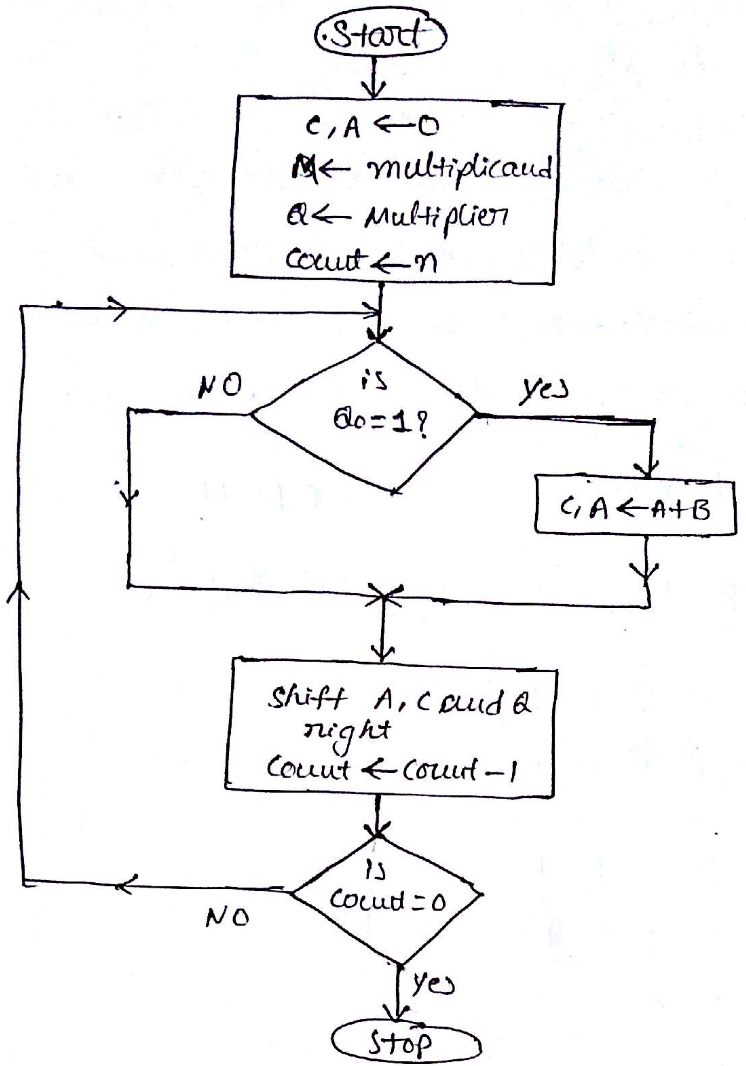
④ Repeat step 1, 2, 3 n times to get the desired result in the A and Q registers.

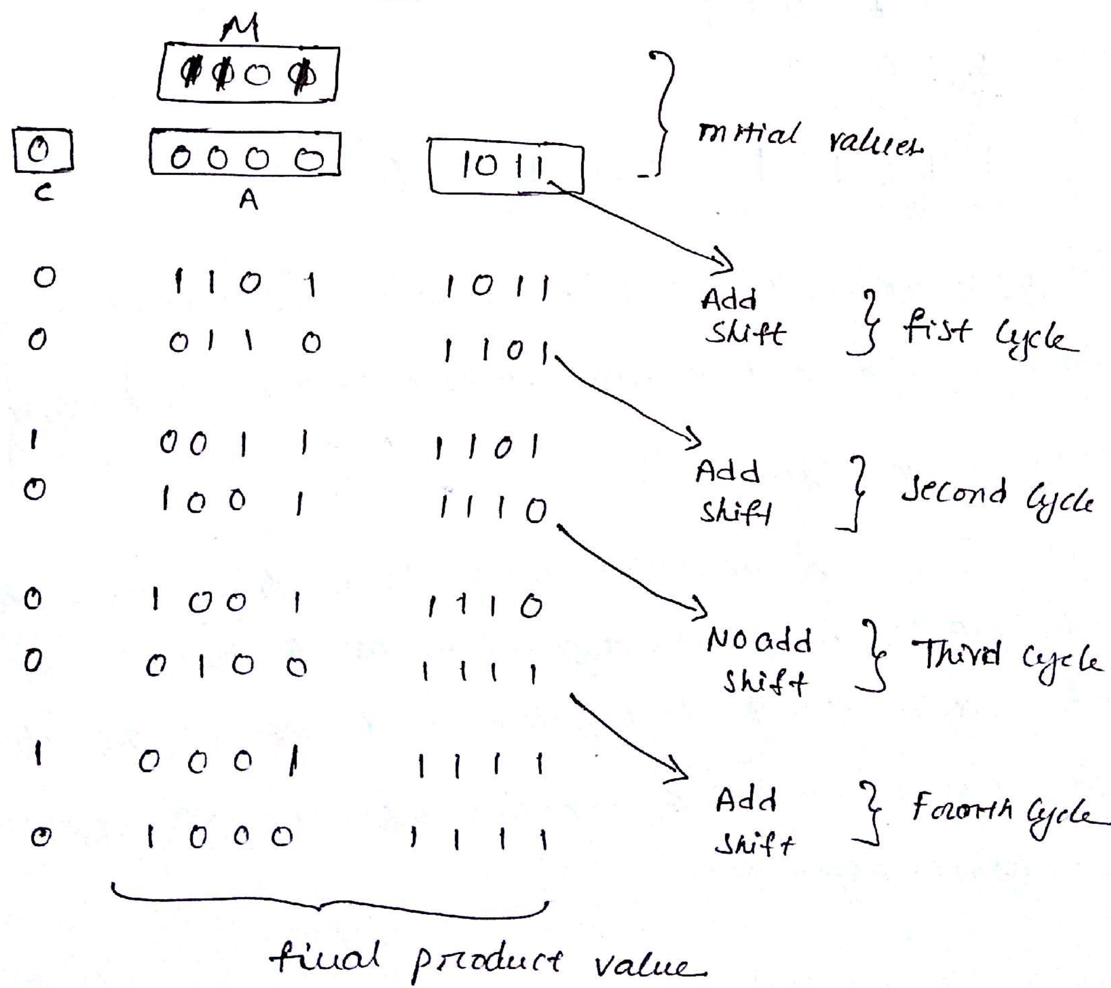
⇒ Consider the example Multiplicand = 1101 Multiplier = 1011



Initially the A register and C (carry bit) values sets to zero. The following flowchart indicates the sequential multiplication operation

Scanned by CamScanner





Scanned by CamScanner

SIGNED-OPERAND MULTIPLICATION :-

Consider the case of a positive multiplier and a negative multiplicand. When we add a negative multiplicand to a particular partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend. Consider the following example −13 X 11

11 = 01011

−13 = 10011 (2's complement of 13).

```
          1 0 0 1 1    (-13)
          0 1 0 1 1    (11)
      ---------------
 1 1 1 1 1 1 0 0 1 1
 1 1 1 1 1 0 0 1 1
 0 0 0 0 0 0 0 0
 1 1 0 0 0 1 1
 0 0 0 0 0 0
---------------------
 1 1 0 1 1 1 0 0 0 1   (-143)
---------------------
```

Now we will discuss a technique which works equally well for both negative and positive multiplier called Booth algorithm

Booth Algorithm :-

A powerful algorithm for signed-number multiplication is Booth's algorithm, which generates a 2n-bit product and treats both positive and negative numbers uniformly. We can use a concept of recoding when we gets negative numbers in Booth Algorithm

Scanned by CamScanner

the general Booth's algorithm recoding scheme can be given as

~~Multiplier~~ -1 times the shifted multiplicand is selected when moving from 0 to 1, +1 times the shifted multiplicand is selected when moving from 1 to 0, and 0 times the shifted multiplicand is selected for none of the above case as multiplier is scanned from right to left.

Multiplier		version of multiplicand selected by bit i
Bit i	Bit i-1	
0	0	0 X M
0	1	+1 X M
1	0	-1 X M
1	1	0 X M

fig: Booth multiplier recoding table

EX:- ① Recode the multiplier 101100

Sol:

1 0 1 1 0 0 [0] ← Implied Zero Multiplier

-1 +1 0 -1 0 0 Recoded Multiplier

→ 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
+1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 } worst-case multiplier.

→ 1 1 0 0 0 1 0 1 1 0 1 1 1 1 0 0
0 -1 0 0 +1 -1 +1 0 -1 +1 0 0 0 -1 0 0 } ordinary multiplier.

→ 0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1
0 0 0 +1 0 0 0 0 -1 0 0 0 +1 0 0 -1 } Good multiplier.

Scanned by CamScanner

⇒ Multiply 01110 (+14) and 11011 (−5).

Sol:

recode the multiplier −5

```
 1  1  0  1  1
0 -1  1  0 -1   → Recoded multiplier.
```

```
                     0   1   1   1   0
                     0  -1  +1   0  -1
--------------------------------------
 1   1   1   1   1   1   0   0   1   0
 0   0   0   0   0   0   0   0   0
 0   0   0   0   1   1   1   0
 1   1   1   0   0   1   0
 0   0   0   0   0   0
--------------------------------------
 1   1   1   0   1   1   1   0   1   0   (−70).
--------------------------------------
```

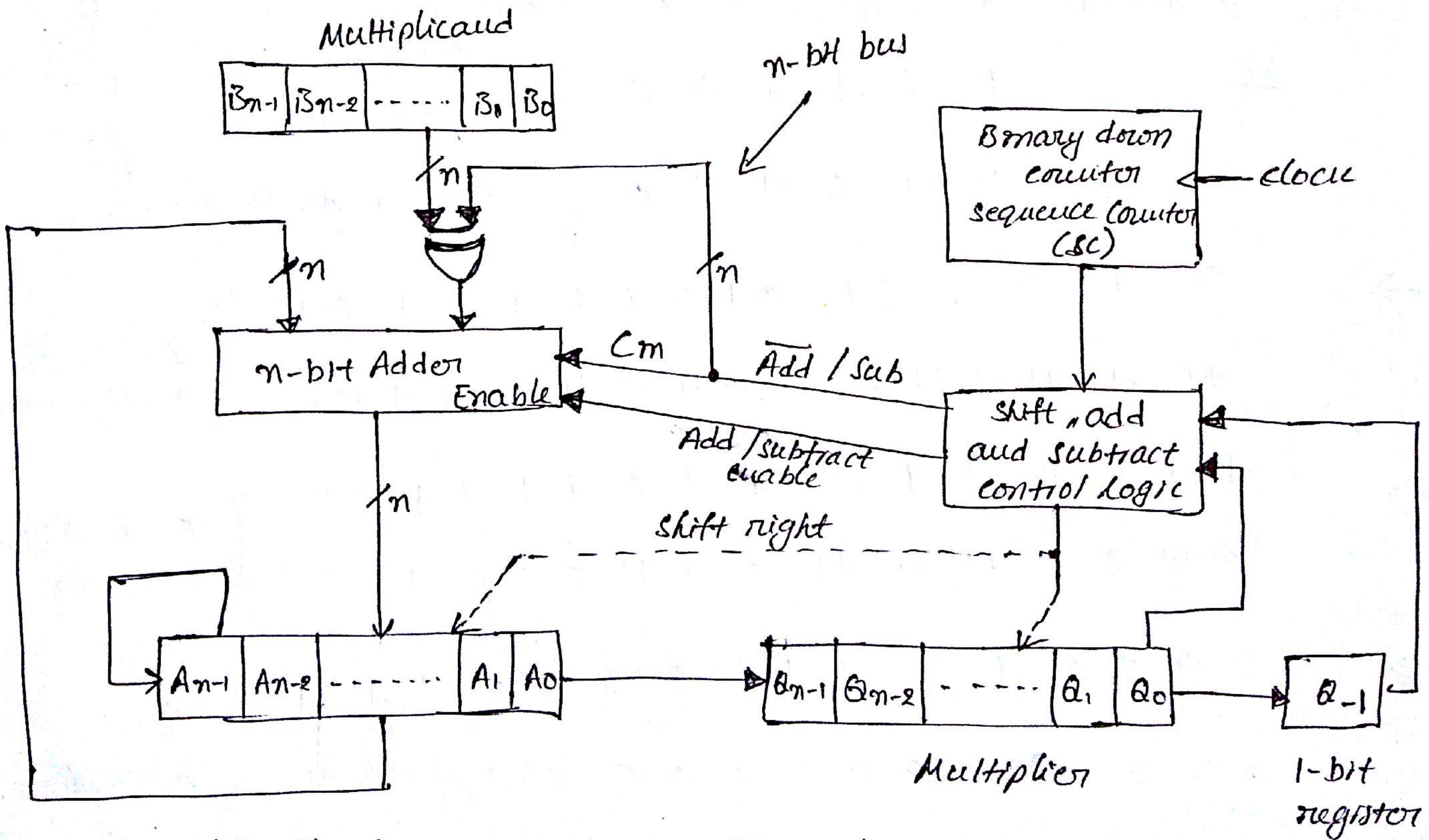


Fig: Hardware implementation of signed binary multiplication

→ initial settings: $A \leftarrow 0$ and $Q_{-1} = 0$

Scanned by CamScanner

Q_0	Q_{-1}	$\overline{Add}$/sub	Add/subtract enable	shift
0	0	x	0	1
0	1	0	1	1
1	0	1	1	1
1	1	X	0	1

fig: truth table for shift, add, and subtract control logic

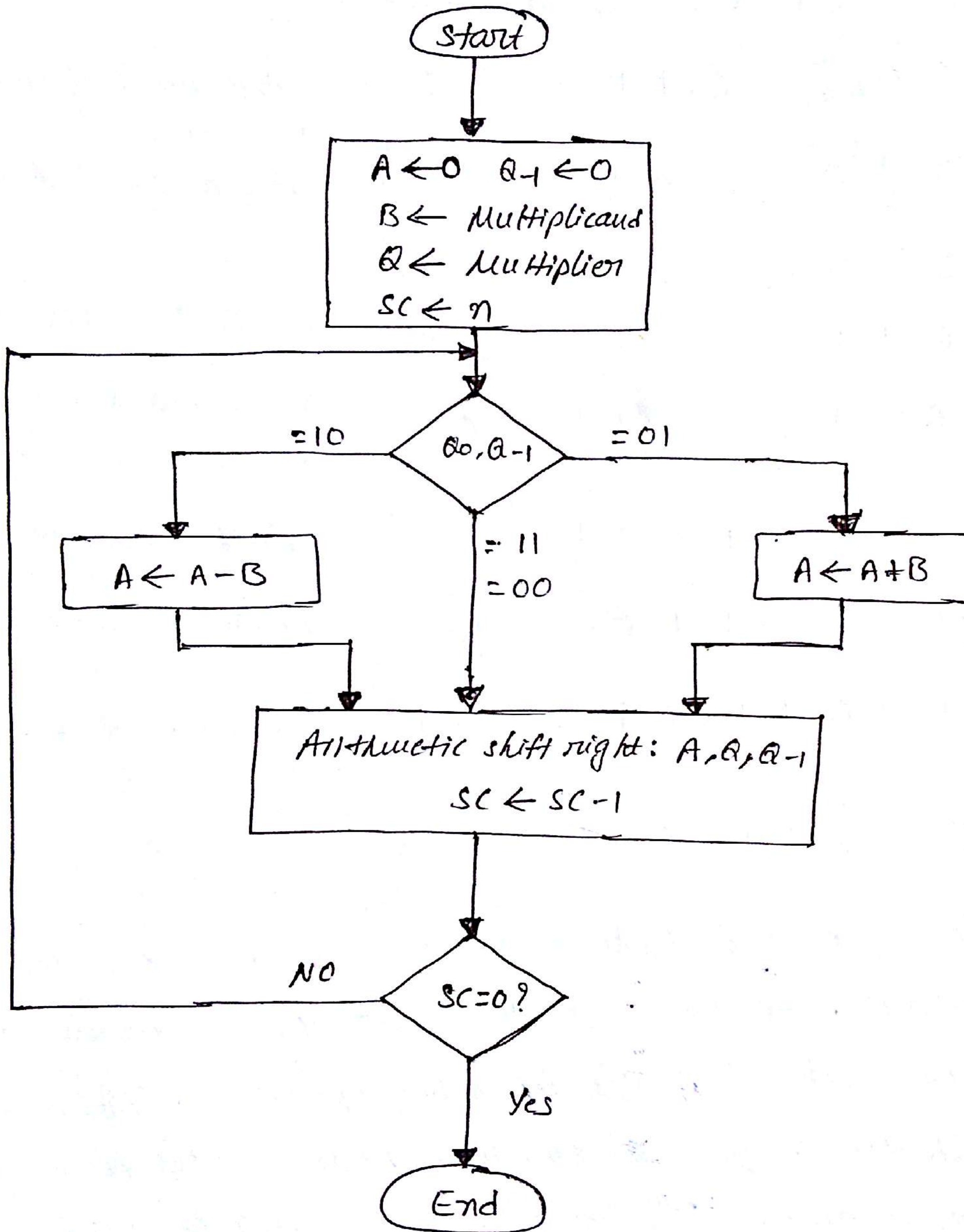


Fig: Booth's algorithm for signed multiplication.

Scanned by CamScanner

Example:- Multiply 5 X -4

Multiplicand (B) = 0101 (5) , Multiplier (Q) = 1100 (-4)

B: 0101

SC	A	Q	Q_{-1}	
100	0000	1100	0	→ initial values
011	0000	0110	0	Arithmetic Shift right
010	0000	0011	0	Arithmetic Shift right
	0000			
	1011			$Q_0 Q_{-1} = 10$
	1011	0011	0	∴ $A \leftarrow A - B$
001	1101	1001	1	Arithmetic shift right
000	1110	1100	1	Arithmetic Shift right.

Result: 11101100 1 = (-20) (2's Complement of 20).

FAST MULTIPLICATION :-

There are two techniques for speeding up the multiplication process. In the first technique the maximum number of summands are reduced to n/2 for n-bit operands. The second technique called the carry save addition reduces the time needed to add the summand. The first approach will be called as Bit-pair recording of multipliers.

Scanned by CamScanner

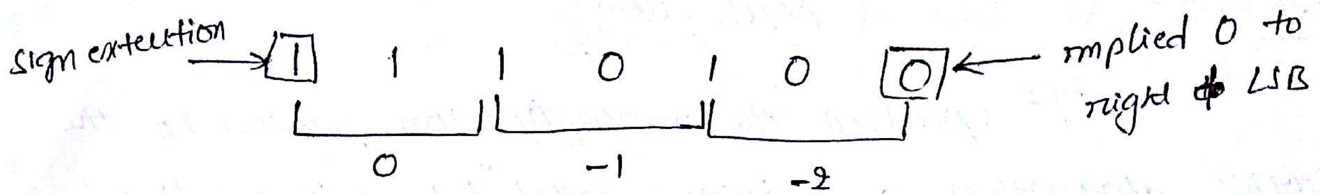
Bit-Pair Recoding of Multipliers:-

To speedup the multiplication process in the Booth's algorithm a technique called "bit-pair recoding" is used. It is also called "modified Booth's algorithm". It decreases the no. of summands. In this technique, the Booth-recoded multiplier bits are grouped in pairs. Then each pair is represented by it's equivalent single bit multiplier reducing total no. of multiplier bits to half. In this technique for a n-bit multiplier we will get n/2 summands. The pair (+1, -1) is equivalent to the pair (0 +1), similarly (+1, 0) is equivalent to (0 +2). The following table indicates the bit-pair code for all possible multiplier bit options.

Multiplier bit-pair		Multiplier bit on the right	Bit pair recoded multiplier bit at position i
i+1	i	i-1	
0	0	0	0
0	0	1	+1
0	1	0	+1
0	1	1	+2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Scanned by CamScanner

Ex: Find the bit-pair code for multiplier 11010



Ex: Multiply given signed 2's complement numbers using bit pair recoding

A = 110 101

B = 011 011

```
      1   1   0   1   0   1
         +2      -1      -1
-----------------------------------
0 0 0 0 0 0 0 0 1 0 1 1
0 0 0 0 0 0 1 0 1 1
1 1 1 0 1 0 1 0
-----------------------------------
1 1 1 0 1 1 0 1 0 1 1 1   (-297)
-----------------------------------
```

Carry - Save Addition Of Summands :-

Multiplication requires the addition of several summands. A technique called carry - save addition (CSA) speeds up the addition process.

Example :- Multiply 01101 (+13) and 11010 (-6) using booth's recoding and bit-pair recoding techniques

Sol :- recoding of multiplier 11010 in booth's technique is

0 -1 +1 -1 0.

Scanned by CamScanner

```
                 0    1    1    0    1
                 0   -1   +1   -1    0
          ---------------------------------
0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  0  0  1  1
0  0  0  0  1  1  0  1
1  1  1  0  0  1  1
0  0  0  0  0  0
---------------------------------------
1  1  1  0  1  1  0  0  1  0      (-78)
---------------------------------------
```

```
                 0    1    1    0    1
                 0         -1        -2
          ---------------------------------
1  1  1  1  1  0  0  1  1  0
1  1  1  1  0  0  1  1
0  0  0  0  0  0
---------------------------------------
1  1  1  0  1  1  0  0  1  0      (-78).
---------------------------------------
```

while multiplying with -2, take the 2's complement for 26 as the multiplicand is 13

$$13 \times -2 = -26$$

2.

The following diagrams are the circuitary diagrams of Ripple-carry and carry save arrays for the multiplication operation $M \times Q = P$ for 4-bit operands

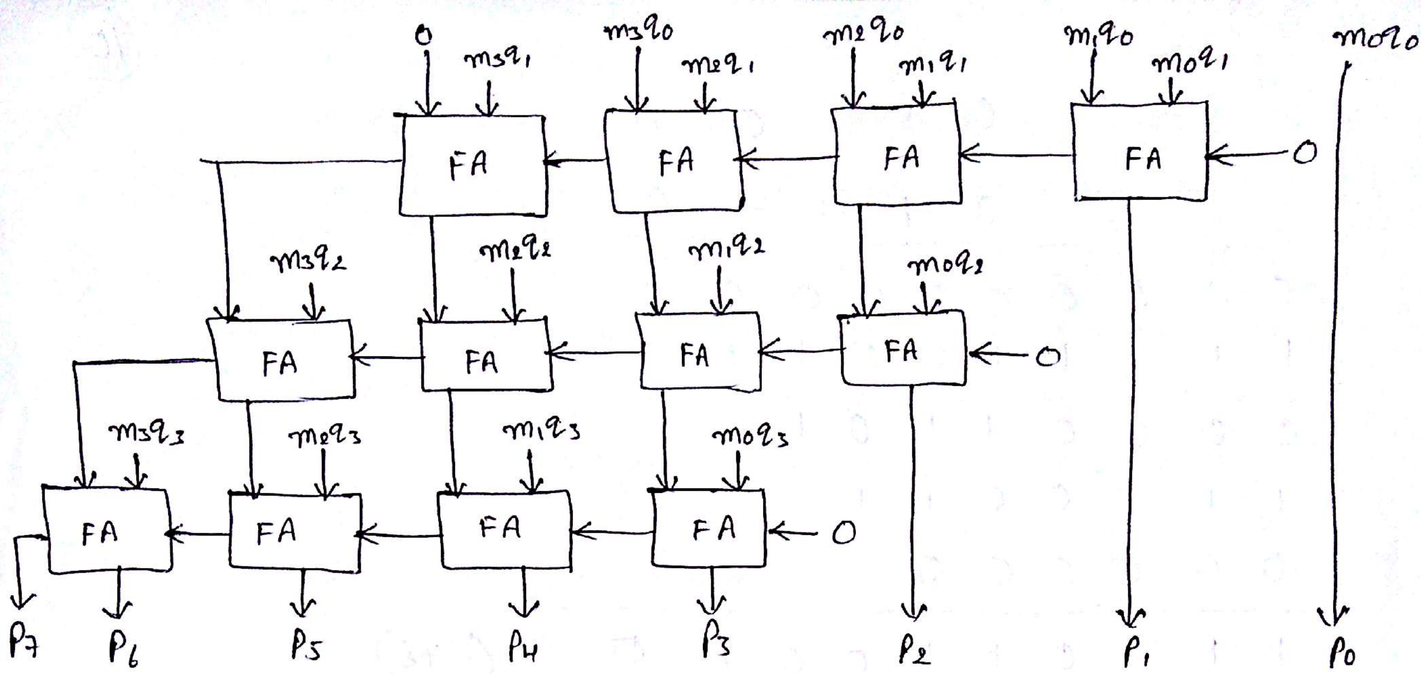


Fig: ripple carry array

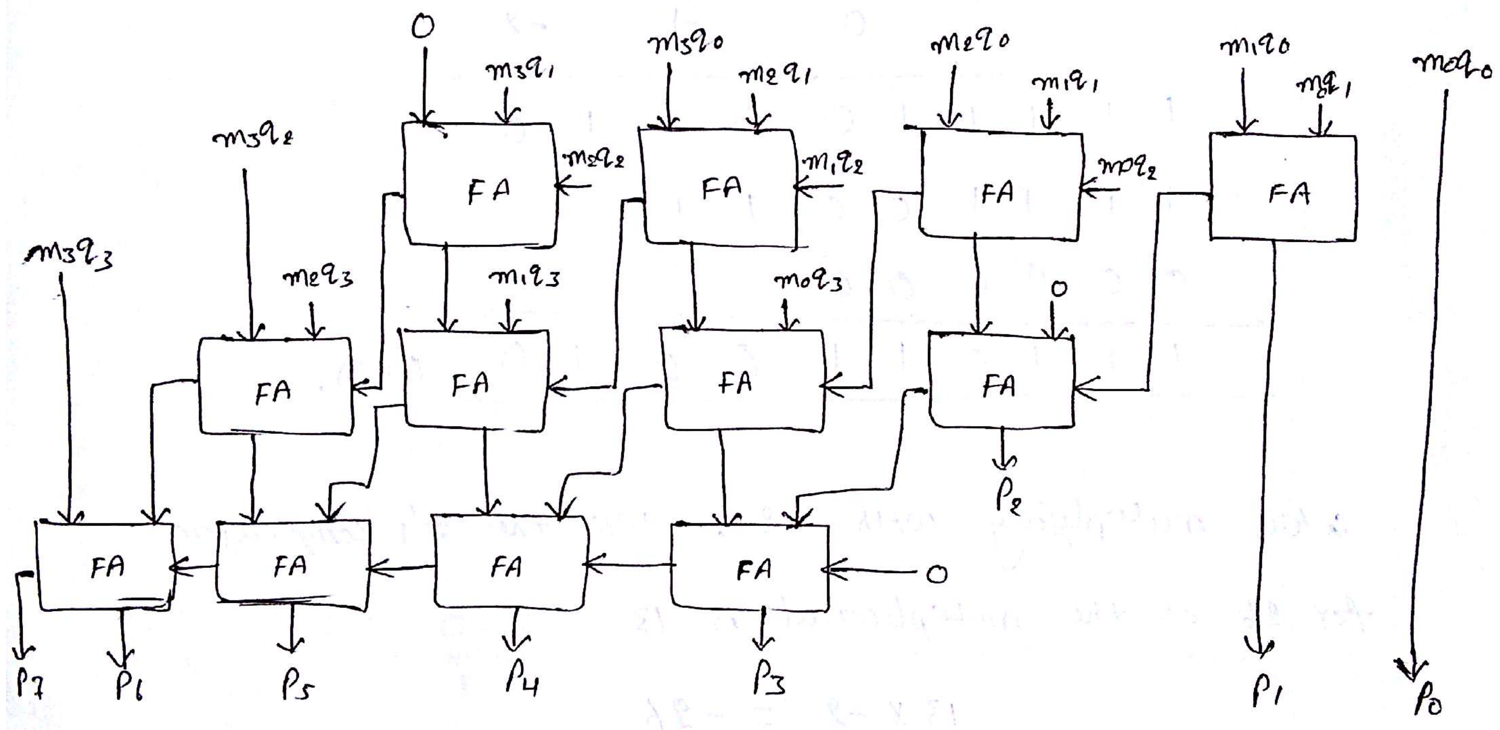


Fig: carry save array

Insted of letting the carries ripple along the rows, they can be saved and introduced into the next row. at the correct weighted positions. This frees up an input to three full adders in the first row. Delay through the carry save array is some what less than delay through the ripple carry array.

Consider a situation where addition of many summands required in the multiplication of longer operands. We can group the summands in threes and performs carry save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay. Next we group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay. We continue with this process until there are only two vectors remaining. Then they can be added in a ripple carry or a carry lookahead adder to produce the desired product.

```
          1 0 1 1 0 1      (45)   M
      ×   1 1 1 1 1 1      (63)   Q
      -----------------
          1 0 1 1 0 1              A
        1 0 1 1 0 1                B
      1 0 1 1 0 1                  C
    1 0 1 1 0 1                    D
  1 0 1 1 0 1                      E
1 0 1 1 0 1                        F
-------------------------
1 0 1 1 0 0 0 1 0 0 1 1   (2835)  product
```

Same multiplication can be done in carry-save addition as follows:

Scanned by CamScanner

```
            101101        M
          × 111111        Q
          ----------
            101101            A
           101101             B
          101101              C
          ----------
          11000011            S1
         00111100             C1

         101101               D
        101101                E
       101101                 F
       ----------
       11000011               S2
      00111100                C2

                              S1 + C1 + S2
        11000011
       00111100
      11100001*
      ------------
      1101010001 1            S3
     000010110000             C3   + C2
     0011111 00
     --------------
     01011101 00 11           S4
     0 1010100000 0           C4
     --------------
     1011000100 11    product
```

INTEGER DIVISION :-

The devision operation is more complex than multiplication. For simplicity we will see division for positive numbers only. The division operation will be done manually like below.

```
          → dividend
   12) 169 (14  ← Quotient
divisor ←  12
          ----
           49   ← partial remainder
           48
          ----
            1   ← remainder value
```

same in binary

```
1100) 10101001 (111
      1100
      ------
       10010
        1100
      -------
        01100
         1100
        ------
        00001   ← remainder
```

The following diagram indicates the binary division.

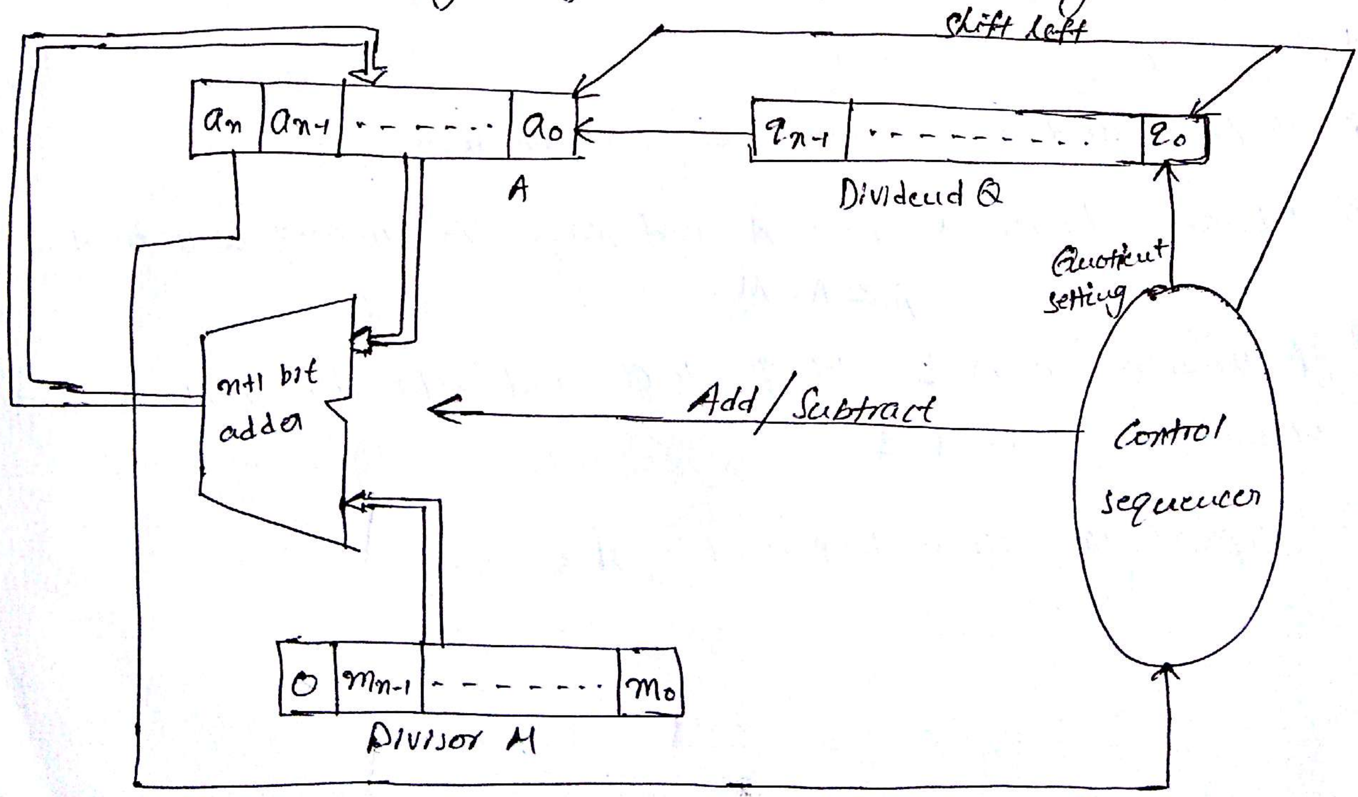


Fig: circuit arrangement for binary division

In both the decisions, division process is same, only in binary division quotient bits are 0 and 1. we will now see binary division in detail. First the bits of the dividend are examined from left to right, weather it is equal or greater than divisor. Until the condition occurs 0's are replaced in the quotient from left to right. when the condition satisfied, a 1 is placed in the quotient and the divisor is subtracted from the partial ~~product~~ dividend. The process continues until all the bits of the dividend are brought down and result is still less than the divisor

Restoring division:

The divisor ~~and~~ which is n-bit positive number is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of division operation. Register A is initially sets to zero. The division operation is then carried out. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A.

division operation steps:

1. shift A and Q left one binary position.
2. Subtract divisor M from A and place the answer back in A.

   $A \leftarrow A - M$.
3. If the sign of A is 1, set q_0 to 0 and add divisor M back to A. otherwise set Q_0 to 1
4. Repeat the steps 1, 2 and 3 n times.

Scanned by CamScanner

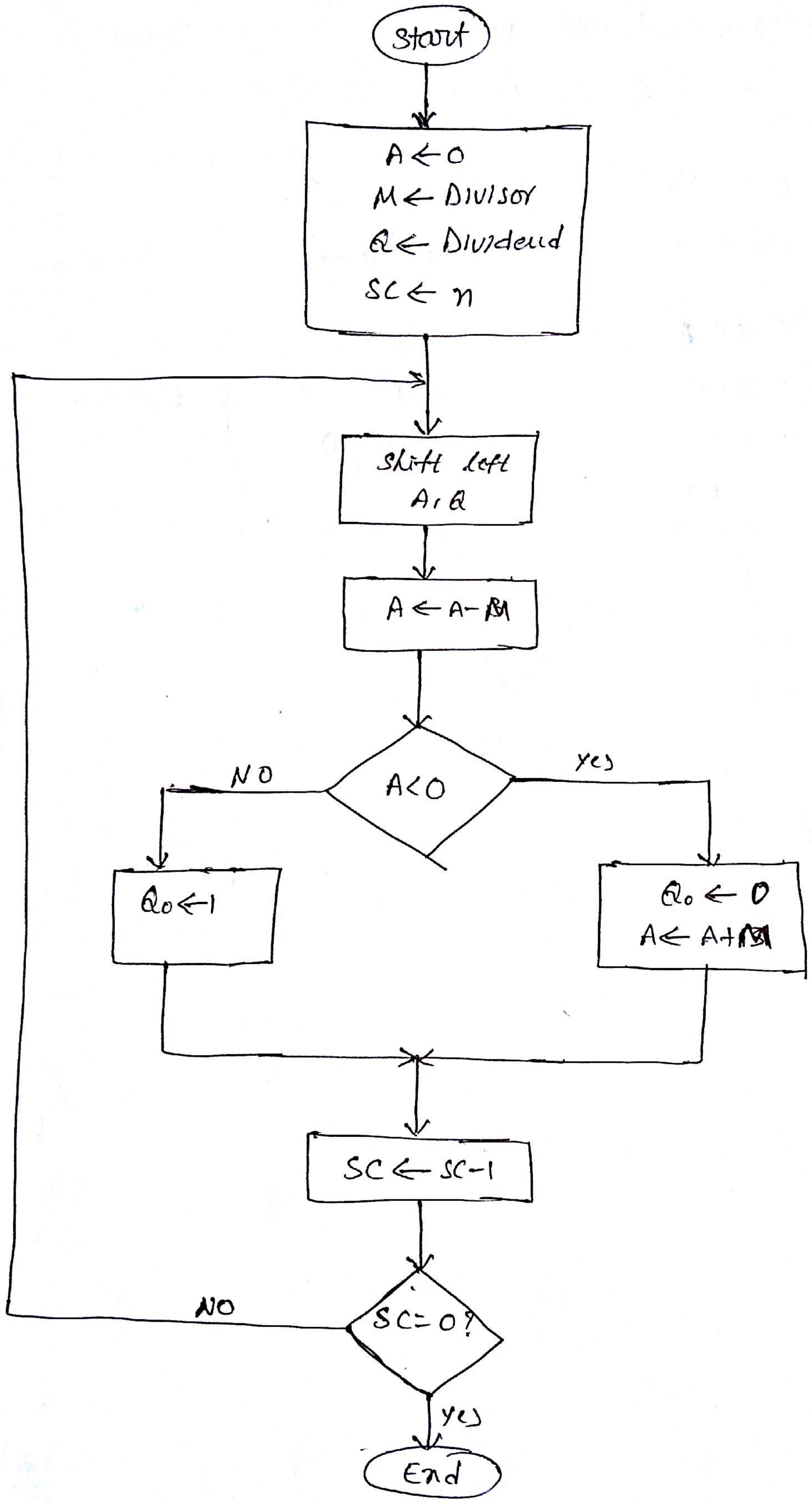


Fig:- flow chart for restoring division operation

Scanned by CamScanner

Example:- Consider 4-bit dividend and 2-bit divisor

Dividend = 1 0 0 0

Divisor = 0 0 1 1

	E	A register	Q register
initially	0	0 0 0 0	1 0 0 0
shift	0	0 0 0 1	0 0 0
subtract M	1	1 1 1 0	0 0 0
Set Q_0 to 0	1	1 1 1 0	0 0 0 0
Restore	1	1 1 1 1	
	0	0 0 0 1	0 0 0 0
shift	0	0 0 1 0	0 0 0
subtract M	1	1 1 1 0	

(1'st cycle)

Examples: consider 4-bit dividend and 2-bit divisor.

Dividend = 1 0 0 0 Q

Divisor = 0 0 1 1 M

	A		Q	
Initially	0	0000	1000	
Shift	0	0001	000	1'st cycle
Subtract M	1	1101		
Set Q_0	1	1110		
	1	1110	0000	
Restore		11		
	0	0001	0000	
Shift	0	0010	000	2nd cycle
Subtract M	1	1101		
Set Q_0	1	1111		
	1	1111	0000	
Restore		11		
	0	0010	0000	
Shift	0	0100	000	3rd cycle
Subtract M	1	1101		
Set Q_0	0	0001		
	0	0001	0001	
Shift	0	0010	001	
Subtract M	1	1101		
Set Q_0	1	1111		
	1	1111	0010	
Restore		11		
Remainder ←	0	0010	0010 → Quotient	

The division algorithm just discussed needs restoring register A after each unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative. Therefore it is referred to as restoring division algorithm. This algorithm is improved, giving non-restoring division algorithm.

Non-restoring algorithm :-

The steps for non-restoring algorithm are

Step 1 : If the sign of A is 0, shift A and Q left one bit position and subtract divisor from A. Otherwise shift A and 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.

Step 2 : Repeat step 1 for n times.

Step 3 : If the sign of A is 1, add divisor to A.

Ex: perform division operation for the following using non-restoring algorithm.

(i) Dividend = 1010

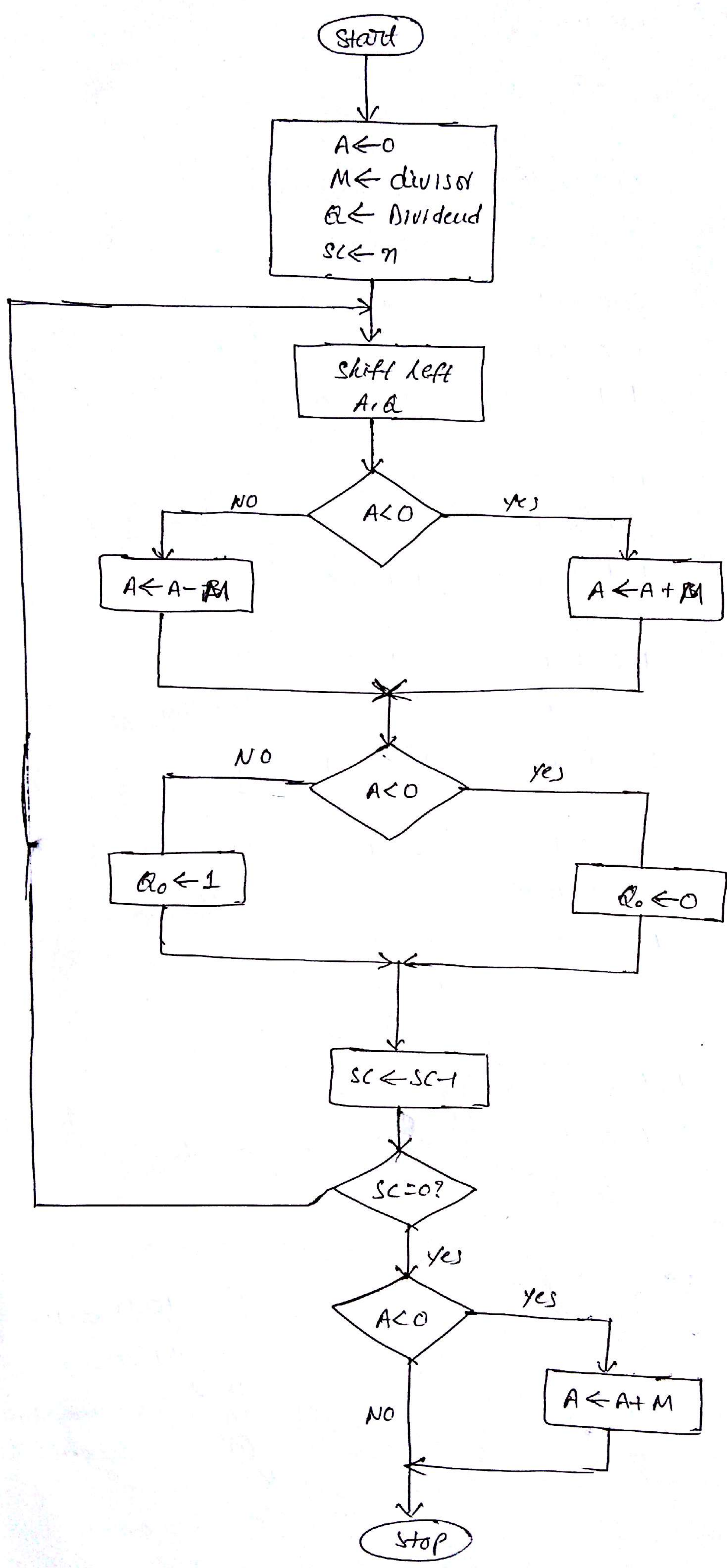
Divisor = 0011

(ii) Dividend = 1011

Divisor = 0101

The following diagram gives the flow of non-restoring algorithm.

Scanned by CamScanner



Scanned by CamScanner

Ex: Dividend = 1011 Q

Divisor = 0101 M

	E	A register	Q Register	
Initially	0	0000	1011	
Shift	0	0001	011	first cycle
subtract	1	1011		
Set Q_0	1	1100	0110	
Shift	1	1000	110	Second cycle
Add	0	0101		
Set Q_0	1	1101	1100	
Shift	1	1011	100	third cycle
Add	0	0101		
Set Q_0	0	0000	1001	
Shift	0	0001	001	fourth cycle
subtract	1	1011		
Set Q_0	1	1100	0010 (Quotient)	

Add

1	1100
0	0101
0	0001

} Restore remainder

00001 → Remainder

1011 = 11

0101 = 5

5) 11 (2 → ~~Remainder~~ = 0010 Quotient

10

(1) → Remainder = 0001

Floating-point numbers and operations :-

Till now we have seen fixed-point numbers and have considered them as integers. If B is a binary vector, then we have seen that it can be interpreted as an unsigned integer by :

$$V(B) = b_{n-1} \cdot 2^{n-1} + b_{n-2} 2^{n-2} + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0 .$$

This vector has an implicit binary point to it's immediate right, then it can represented as

$b_{n-1}\, b_{n-2}\, b_{n-3} \cdots b_1\, b_0$ • implicit binary point

Suppose if the binary vector is interpreted with the implicit binary point is just left of the sign bit :

implicit binary point • $b_{n-1}\, b_{n-2}\, b_{n-3} \cdots b_1\, b_0$

Then the value of b is given by

$$V(B) = b_{n-1} 2^{-1} + b_{n-2} 2^{-2} + b_{n-3} 2^{-3} + \cdots + b_1 2^{-n-1} + b_0 2^{-n}$$

While using floating point numbers we are going to have fraction numbers. Those fractions can be represented by using above notations.

Scanned by CamScanner

The value of the unsigned binary fraction is given by

$$V(B) = b_{31} \cdot 2^{-1} + b_{30} \cdot 2^{-2} + \cdots\cdots + b_1 2^{-31} + b_0 2^{-32}$$

the range can be given by

$$0 \leq V(B) \leq 1 - 2^{-32} \approx 0.9999999998$$

In general for a n-bit binary fraction, the range value is given by

$$\boxed{0 \leq V(b) \leq 1 - 2^{-n}}$$

IEEE Standard Floating-Point Numbers :-

Previous representations have a fixed point. Either the point is to the immediate right or it is to the immediate left. This is called fixed point representation. Here there is a drawback that the representation can only represent a finite range of numbers.

A more convenient representation is the scientific representation, where the numbers are represented in the form :

$$x = m_1 \cdot m_1 m_2 m_3 \times b^{\pm e}$$

Components of these numbers are :

$m \rightarrow$ Mantissa

$b \rightarrow$ implied base

$e \rightarrow$ exponent.

A number such as the following is said to have 7 significant digits.

$$x = \pm 0 . m_1 m_2 m_3 m_4 m_5 m_6 m_7 \times b^{\pm e}$$

Scanned by CamScanner

Fractions in the range 0.0 to 0.9999999 need about 24 bits of precision. For example the binary fraction with 24 1's:

111111111111111111111111 = 0.999999940

Every real number between 0 and 0.999999940 can be represented by a 24-bit fractional number.

The smallest non-zero number that can be represented is:

000000000000000000000001 = 5.96046×10^{-8}

Every other non-zero number is constructed in increments of this value.

A 24-bit mantissa can approximately represent a 7-digit decimal number, and an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. One bit is needed for the sign of the number. Since the leading nonzero bit of a normalized binary mantissa must be a 1, it doesnot have to be included explicitly in the representation. Therefore a total of 32 bits is needed.

The standard representation of floating-point number in 32-bits has been developed and specified in detail by the Institute of Electrical and Electronics Engineers (IEEE).

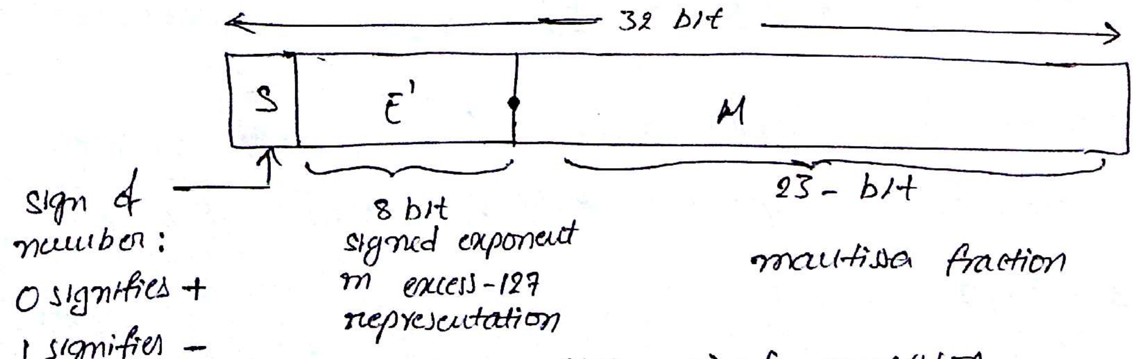


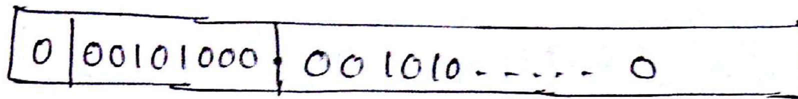
Fig:- single precision

Scanned by CamScanner

value represented in single precision is

$$\pm 1.M \times 2^{E'-127}$$

Ex:



value represented $= 1.001010 \ldots 0 \times 2^{-87}$

Note:- There are two representation for IEEE Standard floating-point representation. They are

(i) single precision
(ii) double precision

→ Single precision floating point represented in 32-bit with the exponent in Excess of 127.

→ double precision floating point represented in 64-bit with the exponent in Excess of 1023.

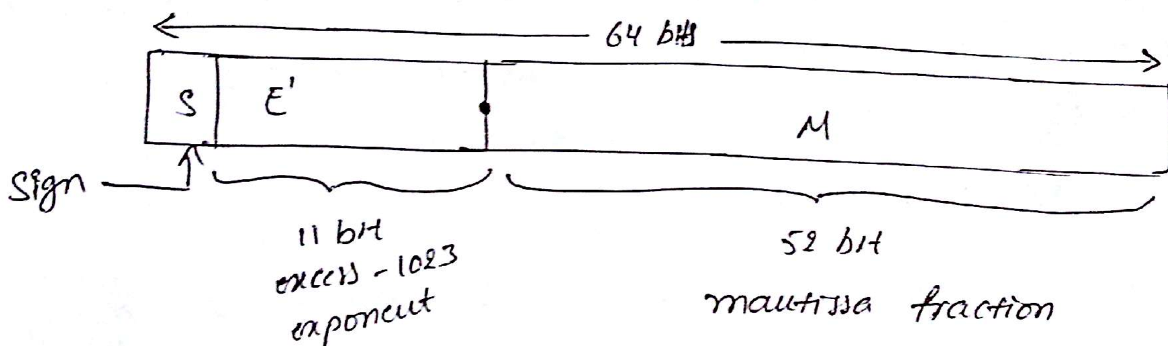


fig: Double precision.

value represented as $= \pm 1.M \times 2^{E'-1023}$

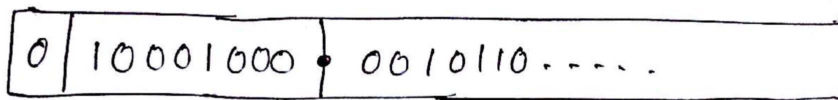
Insted of the signed exponent E, the value actually stored in the exponent field is an unsigned integer $E' = E + 127$.

Scanned by CamScanner

This is called the excess-127 format. Thus E' is in the range $0 \leq E' \leq 225$. The range for normal values is $1 \leq E' \leq 224$. This means that actual exponent E, is in the range $-126 \leq E \leq 127$.

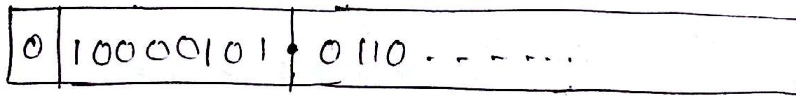
We should consider two aspects of operating with floating-point numbers.

First if a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent.



value represented as = $+0.0010110.... \times 2^9$

Fig:- unnormalized value



value represented as = $+1.0110.... \times 2^6$.

Fig:- Normalized value

Above example is for underflow.

→ underflow occurs when the result of a floating point is larger than the larger positive value or smaller than the smallest negative value

→ overflow occurs when the result of a floating point operation is smaller than the smallest positive number or larger than the largest negative value

Scanned by CamScanner

Arithmetic Operations on Floating point Numbers :-

If the exponents are different, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider the following decimal example in which we wish to ~~add~~ perform addition.

$$2.9400 \times 10^2 \text{ with } 4.3100 \times 10^4$$

Here the exponents are different so we need to shift the floating point number. So we rewrite 2.9400×10^2 to 0.0294×10^4.

Addition and Subtraction :-

Consider two floating point numbers:

$$X = m_1 \cdot \gamma^{e_1} \text{ and}$$

$$Y = m_2 \cdot \gamma^{e_2} \quad \text{Assume } e_1 \geq e_2.$$

Let us see the rules for addition and subtraction.

Step1: Select the number with a smaller exponent and shift it's mantissa right, a number of steps equal to the difference in exponents i.e $|e_2 - e_1|$.

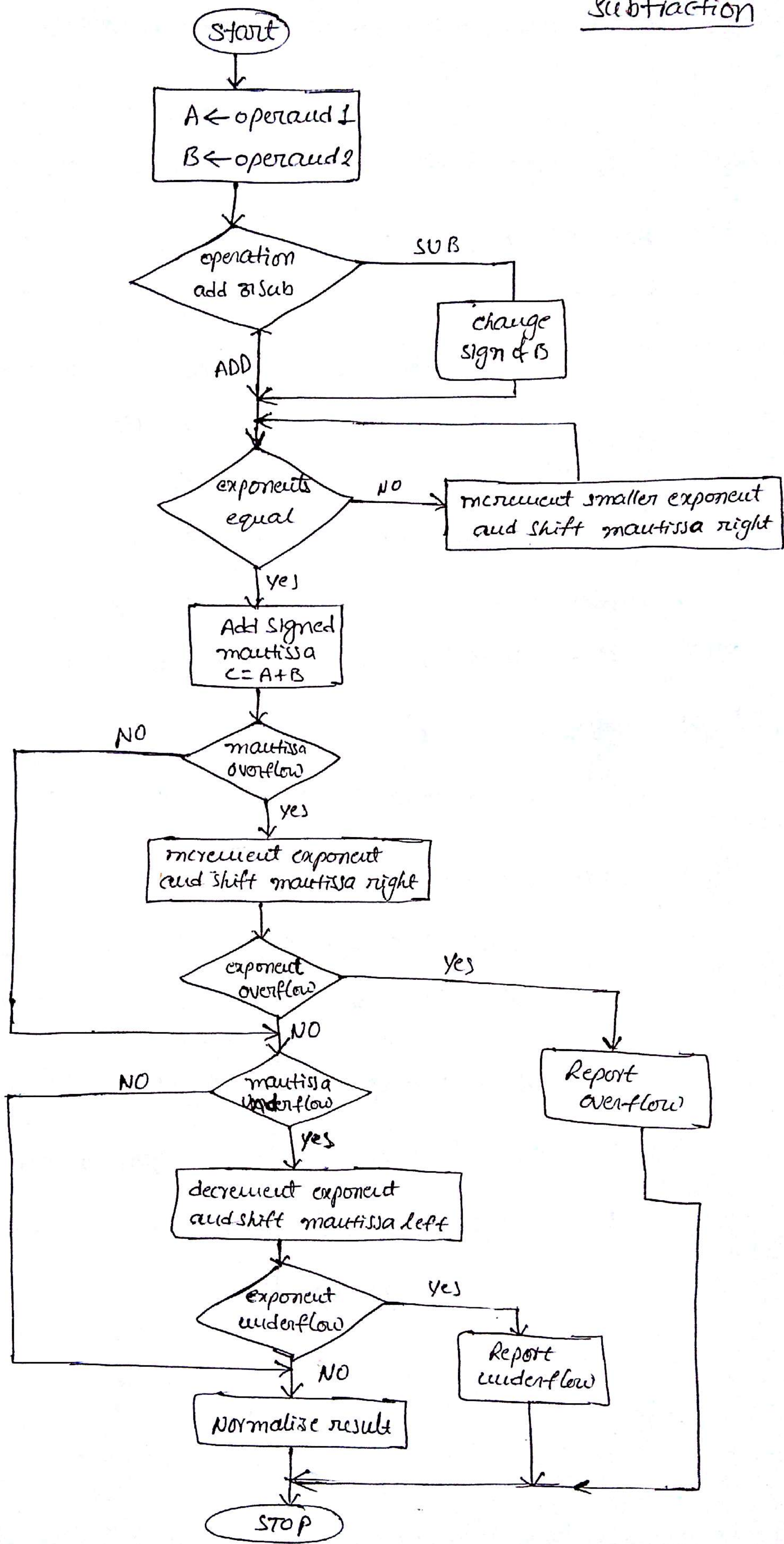
Step2: Set the exponent of the result equal to the larger exponent.

Step3: Perform addition/ subtraction on the mantissa and determine the sign of the result.

Step4: Normalize the result, if necessary

Scanned by CamScanner

flow chart for floating point addition and subtraction



Scanned by CamScanner

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

Sol:-

Step1: Represent numbers in single precision format.

A = 0 10001001 0010000 0

B = 0 10000101 0100000 0

Exponent for A = 1000 1001 = 137

Actual exponent = 137 − 127 = 10

Exponent for B = 10000101 = 133

Actual exponent = 133 − 127 = 6

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

Step2: Shift mantissa

Shifted mantissa of B = 00000100 0

Step3: Add mantissa

Mantissa of A = 00100000 0

Mantissa of B = 00000100 0

Mantissa of result = 00100100 0

As both numbers are positive, sign of the result is positive.

∴ Result = 0 10001001 00100100 0

= 44920000H.

Scanned by CamScanner

Problems in Floating Point Arithmetic :-

① **Mantissa overflow :-** The addition of two mantissas of the same sign may result in a carryout of the most significant bit. If so, the mantissa is shifted right and the exponent is incremented.

② **Mantissa underflow :-** In 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.

③ **Exponent overflow :-** Exponent overflow occurs when a positive exponent exceeds the maximum possible exponent value. In some systems this may be designed as $+\infty$ or $-\infty$.

④ **Exponent underflow :-** Exponent underflow occurs when a negative exponent exceeds the maximum possible exponent value. In such cases, the number is designated as zero.

Multiplication rules :-

Floating point multiplication is somewhat easier than addition and subtraction because an alignment of mantissa is not required.

① Add the exponents and subtract bias. i.e 127 in case of single precision number and 1023 in case of double precision numbers.

② Multiply the mantissas and determine the sign of the result.

③ Normalize the resulting value, if necessary

Scanned by CamScanner

Division Rules:-

① Subtract the exponents and add bias (127 in case of single precision numbers and 1023 in case of double precision number).
② Devide the mantissas and determine the sign of the result.
③ Normalize the resulting value, if necessary

Guard Bits and Truncation:-

While adding two floating point numbers with 24-bit mantissas, we shift the mantissa of the number with the smaller exponent to the right until the two exponents are equalized. This implies that mantissa bits may be lost during the right shift. To prevent this floating point operations are implemented by keeping guard bits.

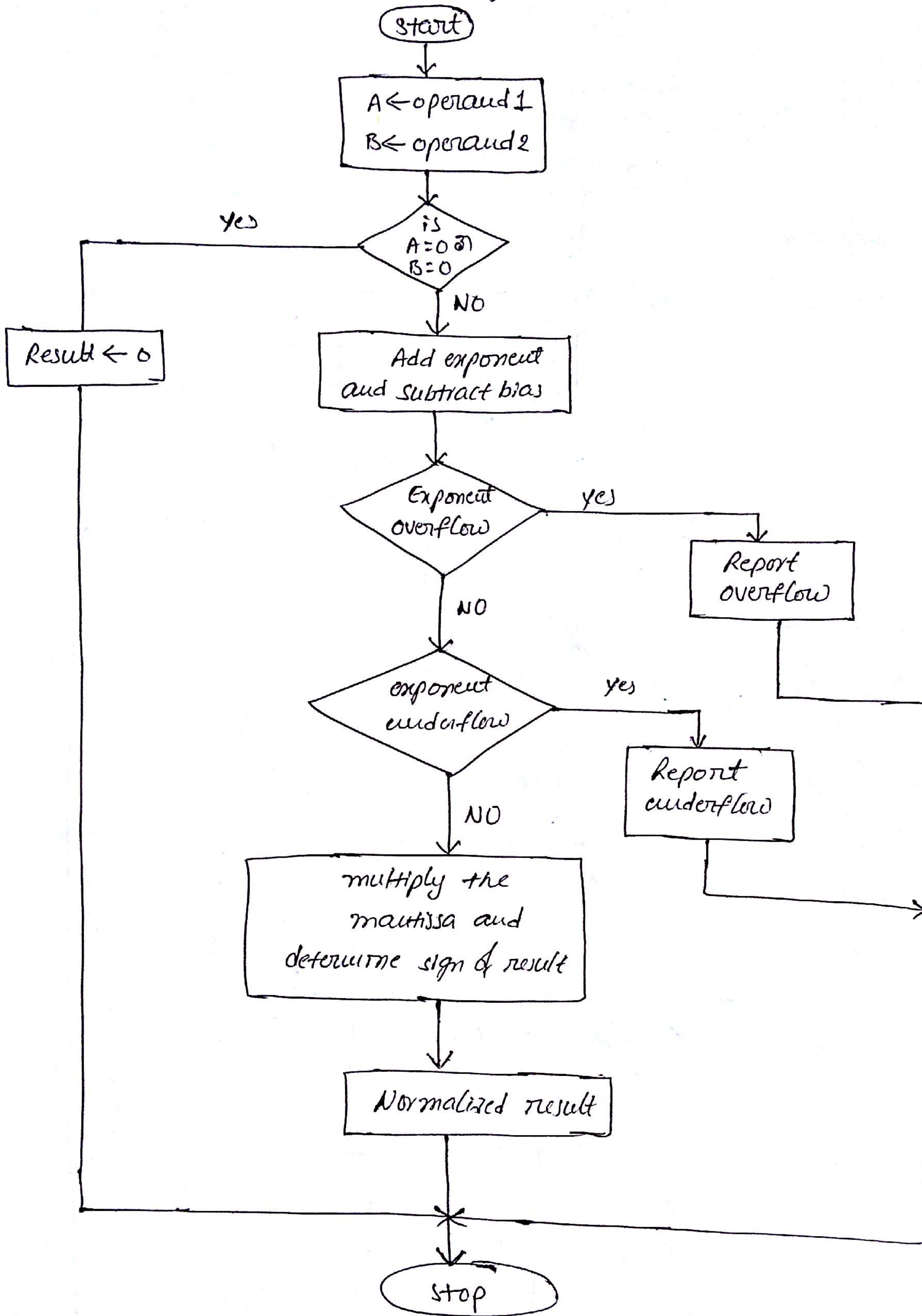
Removing guard bits in generating a final result requires that the extended mantissa be truncated to create 24-bit number that approximates the longer version.

There are several ways to truncate. The simplest way is to remove the guard bits and make no changes in the retained bits. The next simplest method of truncation is Von Neuman rounding. If the guard bits are all 0, they are dropped. However if any bit of the guard bit is a 1, then the LSB of the retained bit is set to 1.

The third truncation method is a rounding procedure. If there is a 1 in the MSB of the guard bit then a 1 is added to the LSB of the retained bits.

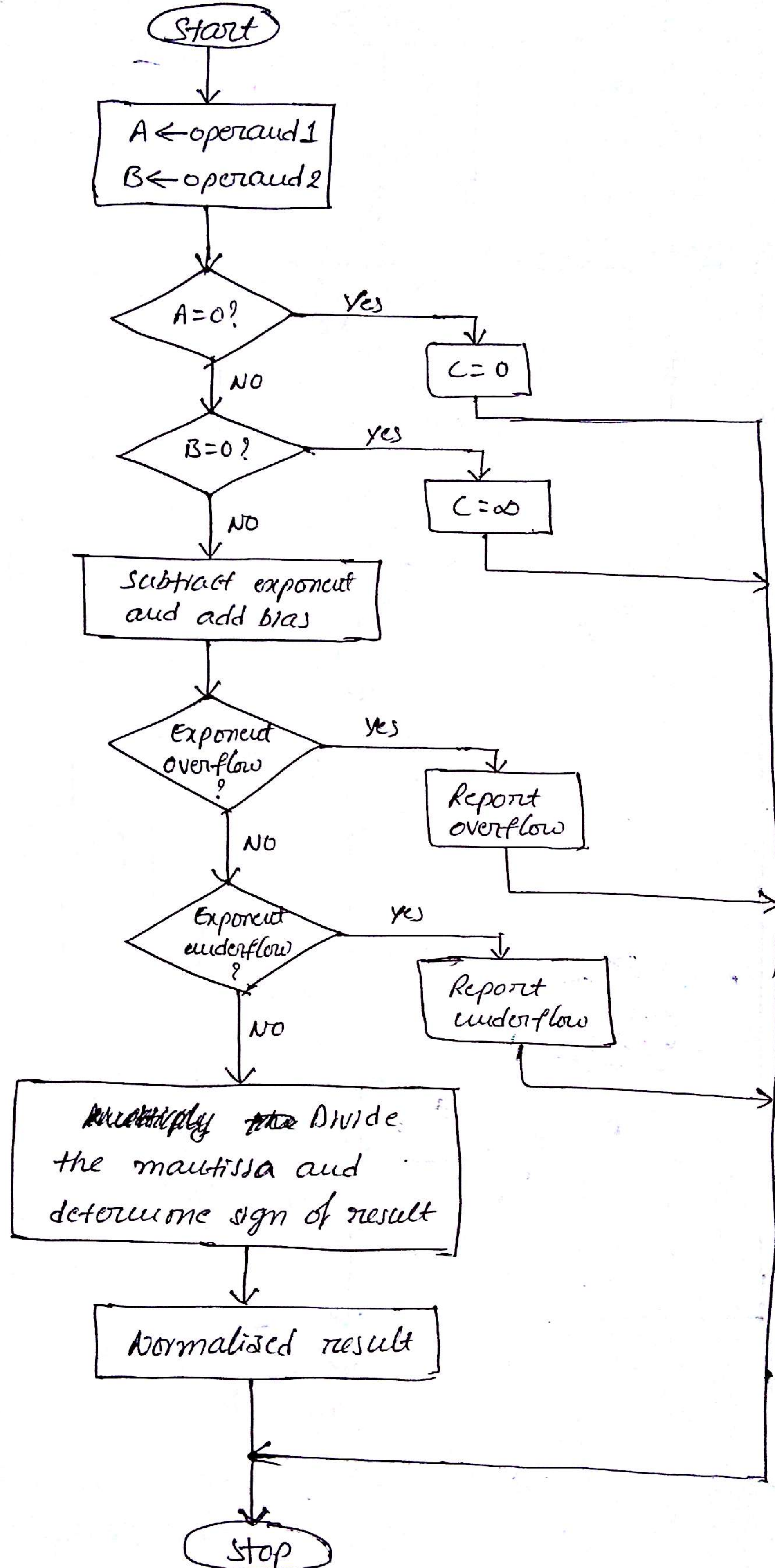
Scanned by CamScanner

flow chart for floating point multiplication.



Scanned by CamScanner

flow chart for floating point Division



Scanned by CamScanner

Implementing Floating - Point Operations :-

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by s/w routines. In most general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware. The following diagram gives the implementation of floating-point operations.

In addition/subtraction operation the first rule is to compare exponents to determine how far to shift the mantissa of the number with the smaller exponent. The shift-count value n, is determined by the 8-bit subtractor circuit in the upper left corner of the figure. The magnitude of the difference $E'_A - E'_B$, or n is sent to the SHIFTER unit.

The sign difference from comparing exponents determines which mantissa is to be shifted. The sign is sent to the SWAP network. If the sign is 0 then $E'_A \geq E'_B$ and the mantissas M_A and M_B are sent straight through the SWAP network. This results in M_B being sent to the SHIFTER, to be shifted n positions to the right. The other mantissa, M_A is sent directly to the mantissa adder/subtractor. If the sign is 1, then $E'_A < E'_B$ and the mantissas are swapped before they are sent to the SHIFTER.

Step 2 :- The exponent of the result E' is determined as E'_A if $E'_A \geq E'_B$, or E'_B if $E'_A < E'_B$ based on the sign of the difference resulting from comparing exponents in step1. This is done by two-way multiplexer.

Scanned by CamScanner

32-bit operands { A : S_A, E_A', M_A ; B : S_B, E_B', M_B }

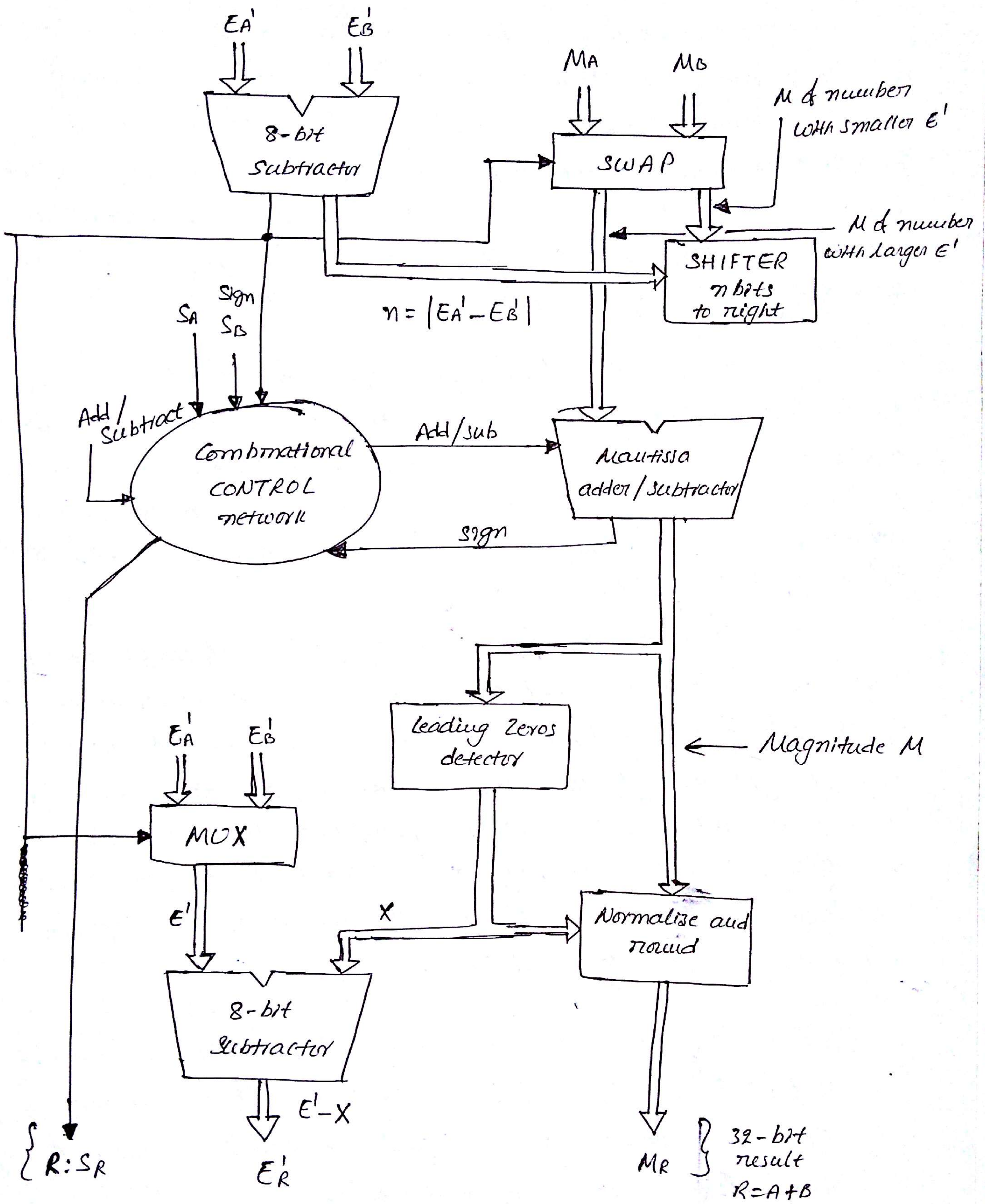


Fig: Floating-point addition-subtraction unit.

Scanned by CamScanner

Step3: involves the major component i.e. the mantissa adder / subtractor. The control logic determines whether the mantissa are to be added or subtracted. It is decided by the signs of operands i.e S_A and S_B. The control logic also determines the sign of result S_R.

For example ~~the sign~~ A is negative ($S_A = 1$) and B is positive ($S_B = 0$) and the operation is A−B, then the mantissas are added and the sign of result is negative ($S_R = 1$).

On the other hand if A and B are positive and the operation is ~~positive~~ A−B, then the mantissas are subtracted. The sign of the result S_R now depends up on the mantissas subtraction.

(i) if $E_A' > E_B'$ then $M_A - (\text{Shifted } M_B)$ is positive and the result is positive.

(ii) if $E_B' > E_A'$ then $M_B - (\text{Shifted } M_A)$ is positive and the result is negative.

Step4: Add / Subtraction operation result will be normalized if necessary

Scanned by CamScanner