Chapter 2

Data Types and Computer Arithmetic

Syllabus

- Scalar Data Types
- Fixed and Floating point numbers
- Signed numbers
- Integer Arithmetic
- 2's Complement multiplication
- Booth's Algorithm
- Hardware Implementation
- Division
- Restoring and Non Restoring algorithms
- Floating point representations
- IEEE standards
- Floating point arithmetic

Information is represented in computer in the form of Binary digit, called bit. We have to learn how arithmetic calculations are performed through these bits.

2.1 Number Systems :

A number system of base r is a system with r distinct symbols for r digits.

System	Base	Symbols (digits)
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Binary	2	0, 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A(10), B(11), C(12), D(13), E(14), F(15)

To determine the quantity that the number represents we multiply the digit by an integer power of r (base) depending on the place it is located and then find the sum of weighted digits.

Example 1 :

Let us consider a number '$d_4\ d_3\ d_2\ d_1\ d_0$' with base 'r' then

the value of the number = $d_4 \times r^4 + d_3 r^3 + d_2 r^2 + d_1 r^1 + d_0$.

Solution :

A number $d_4\ d_3\ d_2\ d_1\ d_0$ in base r is written as,

$(d_4\ d_3\ d_2\ d_1\ d_0)_r$

Number base

2.1.1 Decimal Numbers :

Decimal number system has **10 digits** represented by 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Any decimal number can be represented as a string of these digits.

A decimal number 7452.694 should be written as,

$$7 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 + 6 \times 10^{-1} + 9 \times 10^{-2} + 4 \times 10^{-3}$$

2.1.2 Binary Number :

Binary number system has **2 digits** represented by 0, 1. Any binary number can be represented as a string of these two digits called bits. The base of binary number system is 2.

Decimal equivalent :

$$(101101.101)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 2^5 + 2^3 + 2^2 + 1 + \frac{1}{2} + \frac{1}{2^3} = 45.625$$

Thus, $(101101{\cdot}101)_2 = (45.625)_{10}$

2.1.3 Octal Numbers :

As octal system has eight digits represented as 0, 1, 2, 3, 4, 5, 6, 7. Any octal number can be represented as a string of these digits. The base of octal number system is 8.

Decimal equivalent :

$$(45{\cdot}6)_8 = 4 \times 8^1 + 5 \times 8^0 + 6 \times 8^{-1}$$

$$= 37 + \frac{6}{8} = 37.75$$

$$\therefore\ (45{\cdot}6)_8 = (37.75)_{10}$$

2.1.4 Hexadecimal Numbers :

The hexadecimal system has 16 digits, which are represented as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. The base of hexadecimal number system is 16.

Decimal Equivalent :

$$(F3{\cdot}4)_{16} = F \times 16^1 + 3 \times 16^0 + 4 \times 16^{-1}$$

$$= 15 \times 16 + 3 + \frac{4}{16}$$

$$= 243.25$$

$$(F3{\cdot}4)_{16} = (243.25)_{10}$$

2.1.5 BCD (Binary Coded Decimal) Numbers :

In BCD system, each digit of decimal number is represented using a 4 bit binary number.

Decimal digit	Primary representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

BCD equivalent of a decimal number :

$(5943)_{10}$ = 0101 1001 0100 0011 in BCD

5 9 4 3

Decimal	Binary	Octal	hexadecimal		BCD
0	0	0	0		0000
1	1	1	1		0001
2	10	2	2		0010
3	11	3	3		0011
4	100	4	4		0100
5	101	5	5		0101
6	110	6	6		0110
7	111	7	7		0111
8	1000	10	8		1000
9	1001	11	9		1001
10	1010	12	A	0001	0000
11	1011	13	B	0001	0001
12	1100	14	C	0001	0010
13	1101	15	D	0001	0011
14	1110	16	E	0001	0100
15	1111	17	F	0001	0101

Fig. 2.1 : Table of conversion

2.2 Representation of Data in Computer :

The binary number system is most nature for computer. Computer stores information in flip-flops, which are two state devices. Three systems are widely used for representing both positive and negative numbers :

1. Sign magnitude representation
2. 1's – complement representation
3. 2's – complement representation

2.2.1 Fixed Point Representation :

In the fixed point numbers we assume that the position of the binary point is at the end. It implies that all integers are represented using **fixed point representation**. Real numbers are represented using **floating point representation**.

Sign magnitude representation :

In the above representation, both sign and magnitude of a number are represented separately.

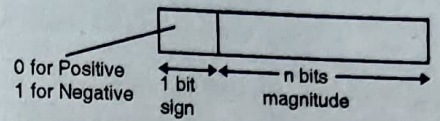


Fig. 2.2 : A (n + 1) bit number

The sign of the number is represented using the left most bit. The left most bit is 0 for positive numbers and 1 for negative numbers. Thus, a number of n bits can be represented as n + 1 bit number, where $(n + 1)^{th}$ bit is the sign bit and rest n bits represent its magnitude.

Example 2 :

Represent 12 and – 12 using sign magnitude representation. Assume size of register = 8 bits.

Solution :

Number	Representation
+ 12	0 0001100 (Sign, Magnitude)
– 12	1 0001100 (Sign, Magnitude)

Disadvantages of sign magnitude representation :

1. There are two representation for 0.
   + 0 is represented as 0 000 0000
   – 0 is represented as 1 000 0000
2. To do the arithmetic addition with one positive number and one negative number, we have to check the magnitude of numbers. The number having smaller magnitude is then subtracted from the bigger number and the sign of bigger number is selected. The implementation of such a scheme in digital hardware is complicated.

1's complement representation :

In 1's complement representation, negative values are obtained by complementing each bit of the corresponding positive number.

Example 3 :

Represent 12 and – 12 using 1's complement representation. Assume size of register = 8 bits.

Solution :

Number	Representation
+ 12	0 0001100 — Sign bit 0 indicates it is a positive number
– 12	1 1110011 — Sign bit 1 indicates it is a negative number

Disadvantages :

There are two representations for 0.

+ 0 is represented as 00000000

– 0 is represented as 11111111

2's complement representation :

In 2's complement representation, negative number is obtained by adding 1 to 1's complement representation.

Example 4 :

Represent 12 and – 12 using 2's complement representation. Assume size of register = 8 bits.

Solution :

Number	Representation
+ 12	0000 1100
– 12	1's complement of (0000 1100) + 1 = 1111 0011 + 1 = 11110100

Advantages :

1. Only one representation for 0.
2. Subtraction is carried out like addition.

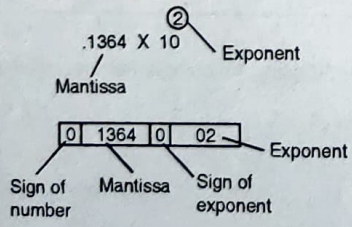
2.2.2 Floating Point Representation :

Floating point number representation consists of two parts :

(1) Mantissa

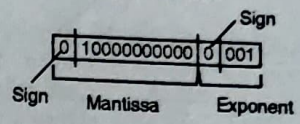
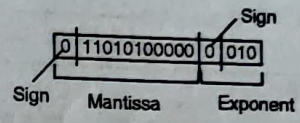
(2) Exponent

A number 13.64 can be written as $\cdot 1364 \times 10^2$.



Exponent indicates the correct decimal location. An exponent of + 2, indicates that actual position of decimal point is two places to the right of the assumed position, while – 2 indicates that the assumed position of the point is two places towards the left of assumed position. Decimal point is assumed and it is never represented physically.

Floating point numbers are often represented in normalized form. A floating point number whose mantissa does not contain zero as the most significant digit of the number is considered to be in normalized form.

Binary number	Normal form	Representation
.0100	$.100 \times 2^{-1}$	
11.0101	$.110101 \times 2^{2}$	

Register size is assumed to be of 16 bits. 12 bits are for mantissa and four bits are for exponent.

2.3 Computer Arithmetic :

2.3.1 Addition of Positive Numbers :

Binary addition is performed in a manner similar to that of decimal addition. Corresponding bits are added, and if a carry 1 is produced, it is added to the binary digit at the left.

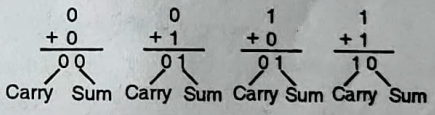


Fig. 2.3(a) : Addition of 1 bit numbers

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
1	1	0	0	1
1	1	1	1	1

Fig. 2.3 (b) : Addition of 1 bit numbers with previous carry

Example 5 :

Find the sum of two binary numbers 1111·01 and 0111·10.

Solution :

$$1111{\cdot}01 = (15.25)_{10}$$

$$+\ 0111{\cdot}10 = (7.50)_{10}$$

$$10110.11 = (22.75)_{10}$$

Initial carry = 0

1 1 1 1 0 0 0 ← Carry in
1 1 1 1 · 0 1 ← First number
0 1 1 1 · 1 0 ← Second number
1 0 1 1 0 · 1 1 ← Sum

Example 6 :

Find the expression for sum s_i and carry-out c_{i+1}, in terms of x_i, y_i and c_i.

Solution :

From the truth table given in Fig. 2.4, we have,

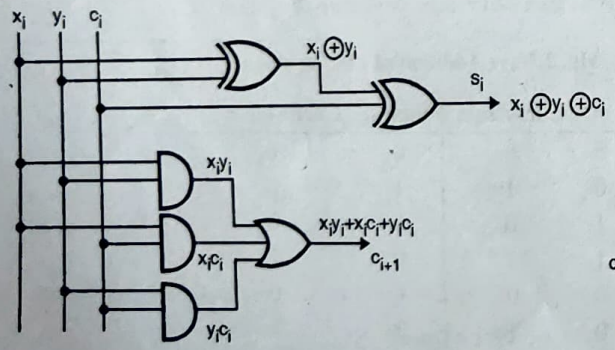
$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i$$

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

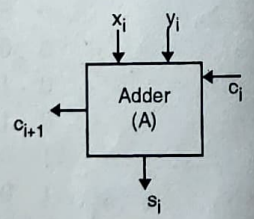
exclusive or

$$c_{i+1} = \bar{x}_i y_i c_i + x_i \bar{y}_i c_i + x_i y_i \bar{c}_i + x_i y_i c_i$$

$$= x_i y_i + x_i c_i + y_i c_i$$



(a) Logic for one bit adder (full adder)



(b) Schematic symbol for 1 bit full adder

Fig. 2.4 : One bit full adder

The implementation of full adder follows directly from the logic expression of truth table in Fig. 2.4.

Example 7 :

Design an adder to add two 4-bit numbers.

Solution :

First number $X = x_3\ x_2\ x_1\ x_0$

Second number $Y = y_3\ y_2\ y_1\ y_0$

Here x_i and y_i (i = 0 to 3) represent a bit.

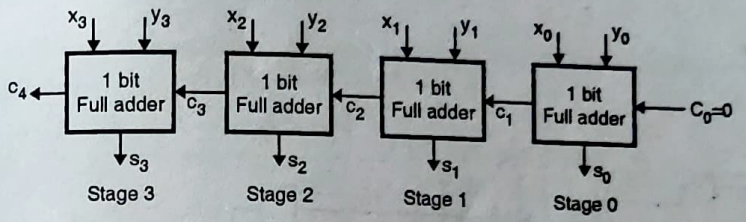


Fig. 2.5 : Logic circuit for addition of two 4-bit binary numbers using carry propagation

s_3, s_2, s_1, s_0 represents the overall sum and overall carry is c_4, from the third stage. The main feature of this adder is that the carry of each lower bit is fed to the next higher bit addition stage, it implies that addition of the next higher bit has to wait for the previous stage addition. Such an adder is called ripple carry adder. The ripple carry addition becomes time consuming when we add two long binary numbers.

- To overcome the problem of excessive delay in ripple carry adder, a fast adder, carry look ahead adder can be designed.
- In carry look-ahead adders, the carry-in for various stages can be generated directly by the logic expressions.

Example 8 :

Design a 4-bit carry look ahead adder.

Solution :

We know that, $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$

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

Let us introduce two more variables g_i and p_i or generate and propagate respectively. g_i and p_i are defined by the following logic equations,

$$g_i = x_i y_i$$

$$p_i = x_i + y_i$$

Now the equation $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$ can be written in terms of g_i and p_i.

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i \quad \ldots(1)$$
$$= g_i + p_i c_i$$

Similarly, c_i can be expressed in terms of g_{i-1}, p_{i-1} and c_{i-1}.

$$c_i = g_{i-1} + p_{i-1} c_{i-1} \quad \ldots(2)$$

on substituting Equation (2) into Equation (1) we obtain,

$$c_{i+1} = g_i + p_i (g_{i-1} + p_{i-1} c_{i-1})$$

Or,

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1}$$

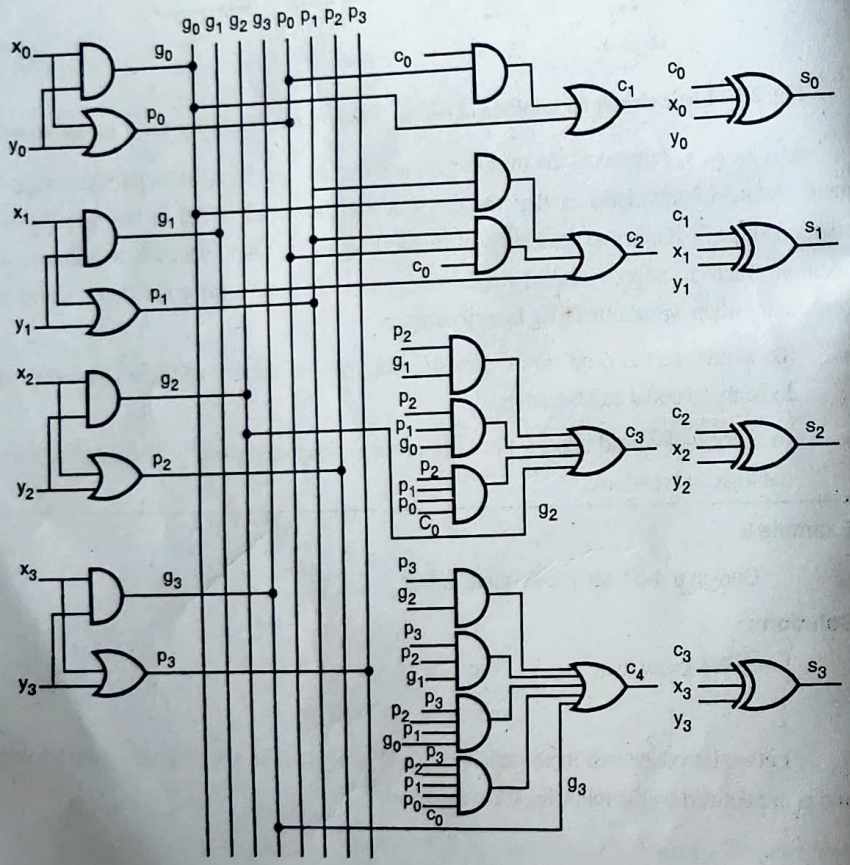


Fig. 2.6 : 4 bit carry look ahead adder

Continuing in this way, c_{i+1} can be expressed as a sum of products functions of the p and g outputs of all the preceding stages. The carries in a four-stage carry-look ahead are defined as follows :

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$
$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$
$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

2.3.2 Signed Addition or Subtraction :

2's complement representation is the best choice for addition of two signed numbers or for finding difference of two numbers.

If x and y are two positive numbers then :

$x - y =$ x + 2's complement of y

$-x - y =$ 2's complement of x + 2's complement of y

$-x + y =$ 2's complement of x + y

Example 9 :

Find the value of the following expressions using 2's complement arithmetic

(1) 55 + 27

(2) – 55 + 27

(3) 55 – 27

(4) – 55 – 27

Assume register size to be of 8 bits.

Solution :

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

(1) 55 + 27 :

	55	00110111
+	27	+ 00011011
	82	01010010

(2) – 55 + 27 :

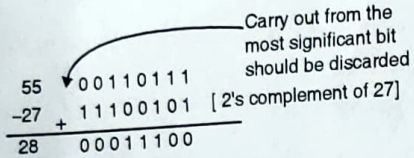
– 55	11001001	[2's complement of 55]
+ 27 +	00011011	
– 28	11100100	

We can verify that – 28 in 2's complement representation is 11100100, by taking 2's complement of 11100100. It should yield 28(00011100).

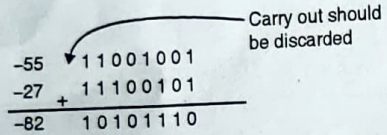
2's complement of 11100100 = 1 + 00011011

= 00011100

(3) 55 – 27 :



(4) – 55 – 27 :



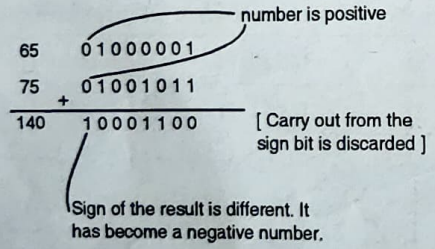
2.3.2.1 Overflow/Underflow in 2's Complement Arithmetic :

Overflow can occur when two positive numbers are added. An overflow is said to have occurred if the sign of the resultant sum of two numbers is different.

Example 10 :

Add two numbers 65 and 75 using 2's complement arithmetic. Assume register size = 8 bits.

Solution :



Since, the sign of the result has changed, an overflow has occurred.

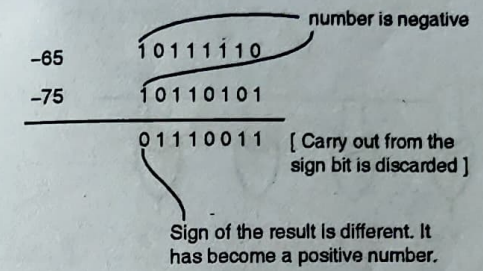
- Underflow can occur when two negative numbers are added. An underflow is said to have occurred if the sign of the resultant sum of two numbers is different.

Example 11 :

Add two numbers – 65 and – 75 using 2's complement arithmetic. Assume register size = 8 bits.

Solution :

Number	Binary equivalent	2's complement
65	01000001	10111110
75	01001011	10110101



Since, the sign of the result has changed, an underflow has occurred.

Example 12 :

Design a 4-bit subtractor using four full adders.

Solution :

$$X = x_3\, x_2\, x_1\, x_0$$

$$Y = y_3\, y_2\, y_1\, y_0$$

$$\therefore X - Y = X + 2\text{'s complement of } Y$$

$$= x_3\, x_2\, x_1\, x_0 + \overline{y}_3\, \overline{y}_2\, \overline{y}_1\, \overline{y}_0 + 1$$

Above expression can be realized by adding X and complement of Y. Initial carry c_0 should be set to 1.

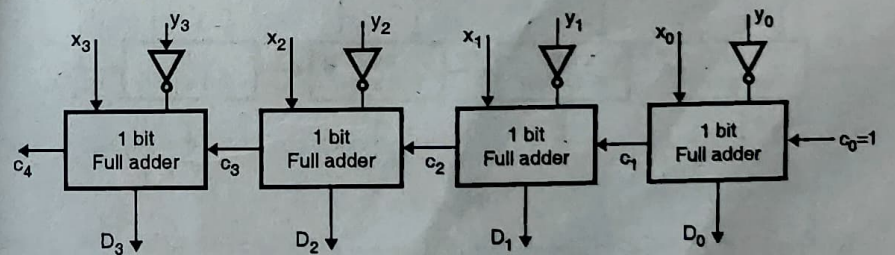


Fig. 2.7 : Logic circuit for subtraction of two 4-bit numbers

Example 13 :

Design a 4-bit addition/subtraction circuit if the external control ctrl = 0 it should perform addition of the two 4-bit numbers and if ctrl =1, it should perform subtraction.

Solution :

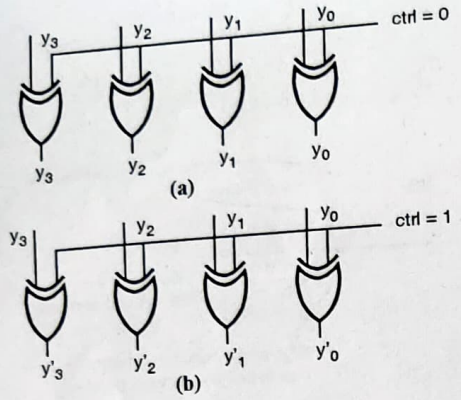


Fig. 2.8

- When the signal ctrl = 0, the binary number passed through the network of exclusive or gates will appear on output without any change. We know,

$$x \oplus 0 = x \text{ [signal passes without any change]}$$

$$\text{and } x \oplus 1 = x' \text{ [signal is inverted]}$$

- When the signal ctrl = 1, the binary number passes through the network of exclusive OR gates will appear inverted.

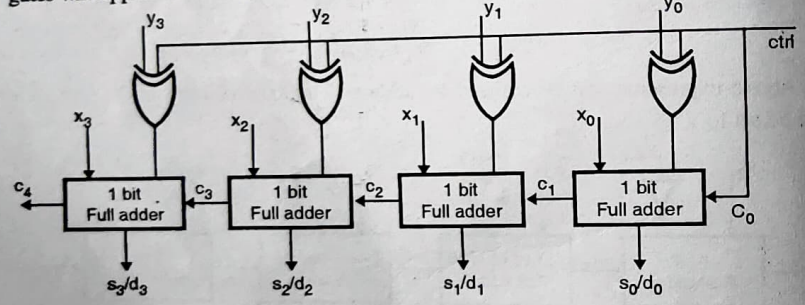


Fig. 2.9

When ctrl = 0, circuit will produce sum of the input numbers

$$X + Y = x_3 x_2 x_1 x_0 + y_3 y_2 y_1 y_0 + 0$$

when ctrl = 1, circuit will produce difference of input numbers

$$X - Y = x_3 x_2 x_1 x_0 + \bar{y}_3 \bar{y}_2 \bar{y}_1 \bar{y}_0 + 1$$

2.3.3 Binary Multiplication (Unsigned Numbers) :

The usual algorithm for multiplying two integers is shown in the Fig. 2.10. This algorithm can be used for multiplication of two unsigned or positive numbers. The product of two n-bit numbers can be accommodated in 2n bits.

Multiplicand M = 11

Multiplier Q = 13

	b_3	b_2	b_1	b_0			bit position
	1	0	1	1		—	(11)
×	1	1	0	1		—	(13)
	1	0	1	1		—	p_0
	0	0	0	0		—	p_1
	1	0	1	1		—	p_2
1	0	1	1			—	p_3
1	0	0	0	1	1	—	(143)

Partial products

p_1 is added to p_0 after right shifting p_0.

The total product is produced by summing the partial products.

Algorithmic approach

Let us take a variable x initialized with 0, i.e.,

$$x = 0000 \text{ [a four bit binary number]}$$

Step 1 : Since the b_0 bit of multiplier is 1

$$x = x + 1011 = 0000 + 1011 = 1011$$

Step 2 : Right shift x

b_3	b_2	b_1	b_0	
0	1	0	1	1

Step 3 : Since, the b_1 bit of multiplier is 0

$$x = x + 0000 = 0101 \mid 1 + 0000 = 0101 \mid 1$$

Step 4 : Right shift the result

	b_3	b_2	b_1	b_0	
x =	0	0	1	0	11

Example 13 :

Design a 4-bit addition/subtraction circuit if the external control ctrl = 0 it should perform addition of the two 4-bit numbers and if ctrl =1, it should perform subtraction.

Solution :

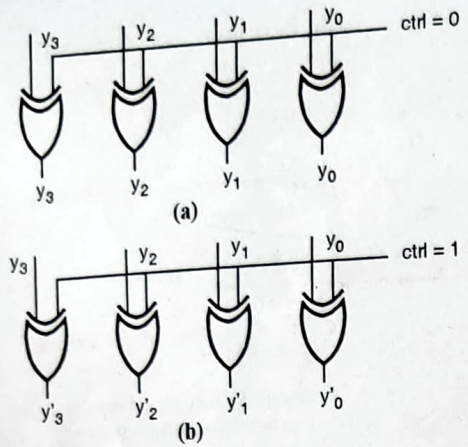


Fig. 2.8

- When the signal ctrl = 0, the binary number passed through the network of exclusive or gates will appear on output without any change. We know,

$$x \oplus 0 = x \text{ [signal passes without any change]}$$

$$\text{and } x \oplus 1 = x' \text{ [signal is inverted]}$$

- When the signal ctrl = 1, the binary number passes through the network of exclusive OR gates will appear inverted.

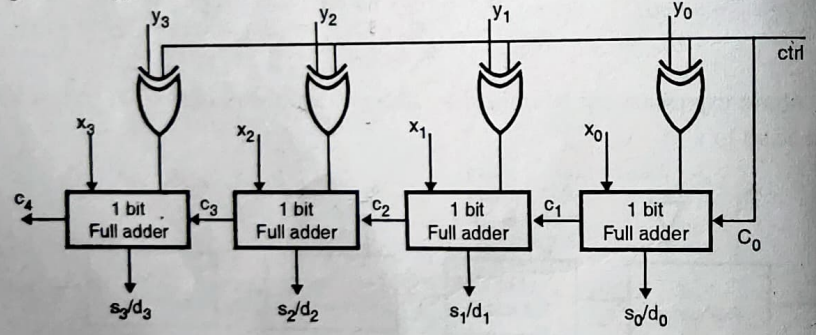


Fig. 2.9

When ctrl = 0, circuit will produce sum of the input numbers

$$X + Y = x_3 x_2 x_1 x_0 + y_3 y_2 y_1 y_0 + 0$$

when ctrl = 1, circuit will produce difference of input numbers

$$X - Y = x_3 x_2 x_1 x_0 + \bar{y}_3 \bar{y}_2 \bar{y}_1 \bar{y}_0 + 1$$

2.3.3 Binary Multiplication (Unsigned Numbers) :

The usual algorithm for multiplying two integers is shown in the Fig. 2.10. This algorithm can be used for multiplication of two unsigned or positive numbers. The product of two n-bit numbers can be accommodated in 2n bits.

Multiplicand M = 11

Multiplier Q = 13

	b_3	b_2	b_1	b_0		bit position
	1	0	1	1	—	(11)
×	1	1	0	1	—	(13)
	1	0	1	1	—	p_0
	0	0	0	0	—	p_1
1	0	1	1		—	p_2
1	0	1	1		—	p_3
1	0	0	0	1	1	1 1 — (143)

Partial products

p_1 is added to p_0 after right shifting p_0.

The total product is produced by summing the partial products.

Algorithmic approach

Let us take a variable x initialized with 0, i.e.,

x = 0000 [a four bit binary number]

Step 1 : Since the b_0 bit of multiplier is 1

$$x = x + 1011 = 0000 + 1011 = 1011$$

Step 2 : Right shift x

b_3	b_2	b_1	b_0	
0	1	0	1	1

Step 3 : Since, the b_1 bit of multiplier is 0

$$x = x + 0000 = 0101 \mid 1 + 0000 = 0101 \mid 1$$

Step 4 : Right shift the result

	b_3	b_2	b_1	b_0	
x =	0	0	1	0	11

Step 5 : Since, the b_2 bit of multiplier is 1

$$x = x + 1011$$

i.e.

	0010	11
+	1011	
	1101	11

Step 6 : Right shift the result

$$x = 0110 \mid 111$$

Step 7 : Since, the b_3 bit of multiplier is 1

$$x = x + 1011$$

	0110	111
+	1011	
	10001	111

Step 8 : Right shift the result

$$x = 1000 \mid 1111$$

Example 14 :

Show multiplication of two numbers 11 and 13 using 4-bit registers.

Solution :

$$(11)_{10} = (1011)_2$$

$$(13)_{10} = (1101)_2$$

Let us assume existence of the following 4-bit registers :

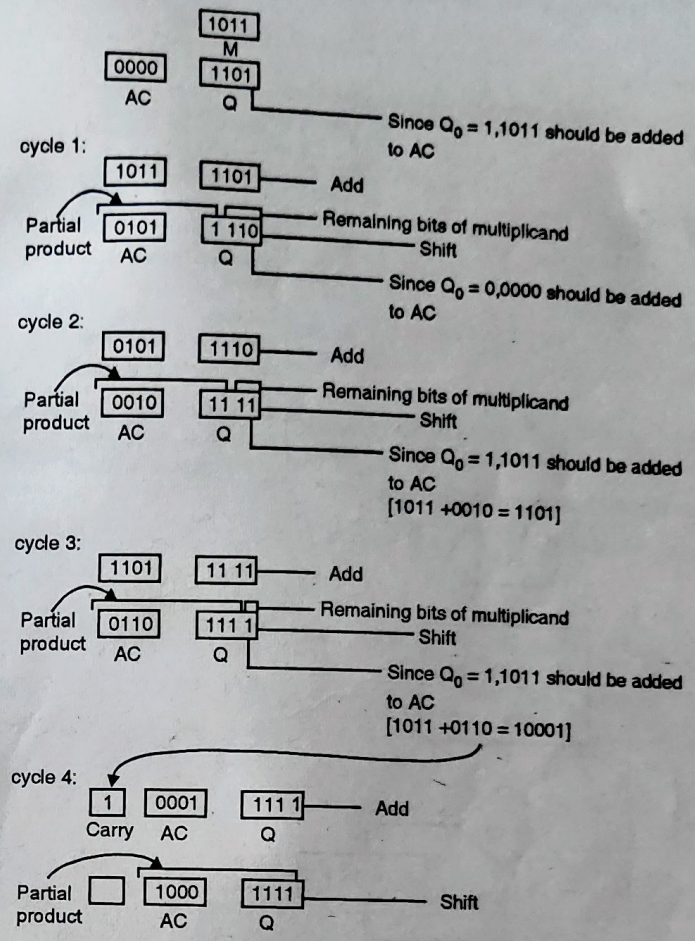
AC → Accumulator

M → Multiplicand

Q → Multiplier

Algorithm discussed above can be implemented with the help of three registers AC, M and Q.

Initial value of registers :



- Two 4-bit numbers are multiplied in four cycles.
- If Q_0 is 1 then M is added to AC and combined AC, Q register is shifted right by 1 bit.
- If Q_0 is 0 then combined AC, Q register is shifted right by 1 bit (Addition not required).

Example 15 :

Show multiplication of two numbers 13 and 14 using 4-bit registers.

Solution :

M = 1101 (13)

Carry	AC	Q		
0	0000	1110		initial values
0	0000	0111		shift] 1st cycle since Q_0=0, no addition is required
0	1101	0111	AC = AC+M	Add(M) shift] 2nd cycle since Q_0=1, at the end of first cycle, M is added
0	0110	1011		
1	0011	1011	AC = AC+M	Add(M) shift] 3rd cycle since Q_0=1, at the end of 2nd cycle, M is added
0	0110	1101		
1	0110	1101	AC = AC+M	Add(M) shift] 4th cycle since Q_0=1, at the end of 3rd cycle, M is added
0	1011	0110		

Flowchart for unsigned binary multiplication :

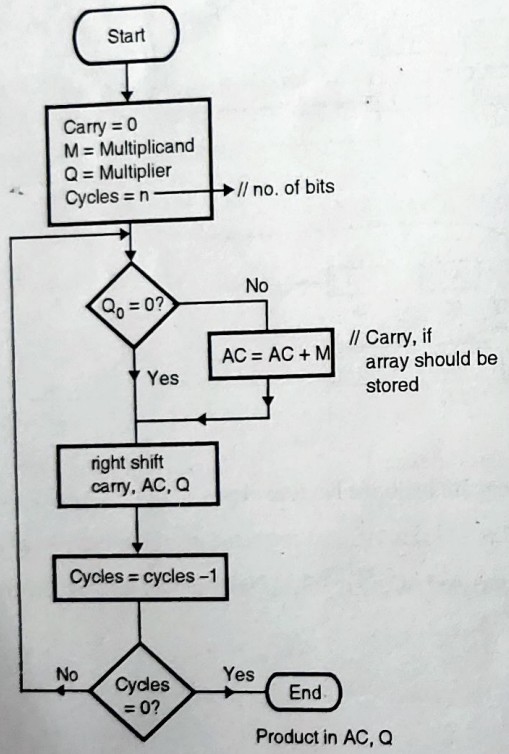


Fig. 2.10 (a)

Hardware implementation of unsigned binary multiplication :

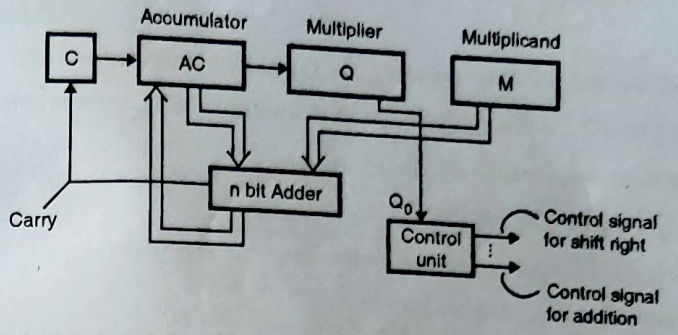
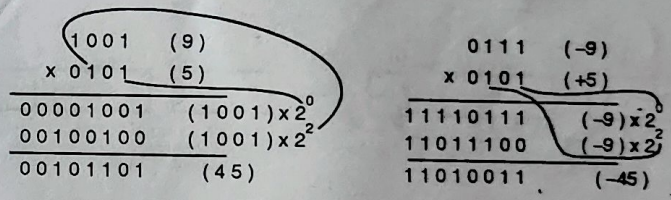


Fig. 2.10 (b)

- Control unit checks the bits Q_0. If the Q_0 bit is found to be 1 then : control unit generates the required control signals to perform AC = AC + M and subsequently, C, AC, Q registers are right shifted by 1 bit.
- If the Q_0 bit is found to be 0 then : control unit generates the required control signals to right shift C, AC, Q registers.

2.3.4 Binary Multiplication (Signed Number) using Booth's Algorithm :

Straight forward multiplication does not work with negative numbers. The same can be understood through an example :



(a) Unsigned integers **(b) 2's complement integers**

Fig. 2.11 : Comparison of multiplication of unsigned and 2's complement integers

Multiplier can not be used directly in the manner it is used for unsigned numbers. One of the most common algorithms for multiplication 2's complement numbers is Booth's algorithm.

2.3.4.1 Booth's Algorithm :

May 2005, May 2007

- Multiplier and multiplicand are placed in the Q and M registers respectively.
- A 1-bit register Q_{-1} is placed right of the least significant bit Q_0 of the register Q.
- The final result will appear in AC and Q registers.
- AC and Q_{-1} registers are initialized to 0.
- Multiplication of number is done in n cycles.
- In each cycle Q_0 and Q_{-1} bits are examined :
  - (i) If Q_0 and Q_{-1} are (1 – 1 or 0 – 0) then all the bits of AC, Q and Q_{-1} registers are shifted to the right by 1-bit.
  - (ii) If Q_0 and Q_{-1} are 01 then multiplicand is added to AC. After addition AC, Q and Q_{-1} registers are shifted to the right by 1 bit.
  - (iii) If Q_0 and Q_{-1} are 10 then multiplicand is subtracted from AC. After subtraction AC, Q and Q_{-1} registers are shifted to the right by 1-bit.

Booth's algorithm uses the concept of arithmetic right shift. In arithmetic right shift, the left most bit of AC is not only shifted right by 1 bit but it also remains in the original position.

Example 16 :

Right shift contents of AC, Q, Q_{-1}.

Solution :

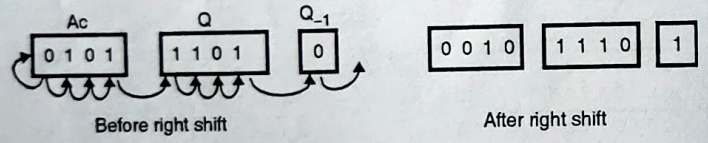


Fig. (a) Positive numbr

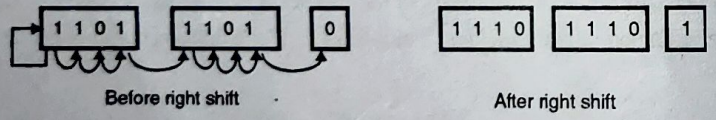


Fig. (b) Negative number

Fig. 2.12 : Arithmetic right shift operation on signed numbers

2.3.4.2 Flowchart of Booth's Algorithm :

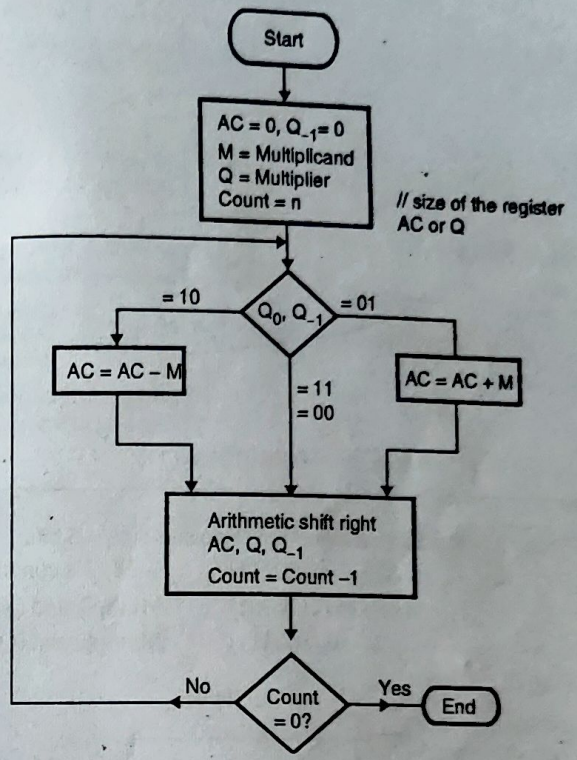


Fig. 2.13 : Flowchart of Booth's algorithm for 2's complement multiplication

Example 17 :

Multiply 7 and 3 using Booth's algorithm. Register size = 4-bits.

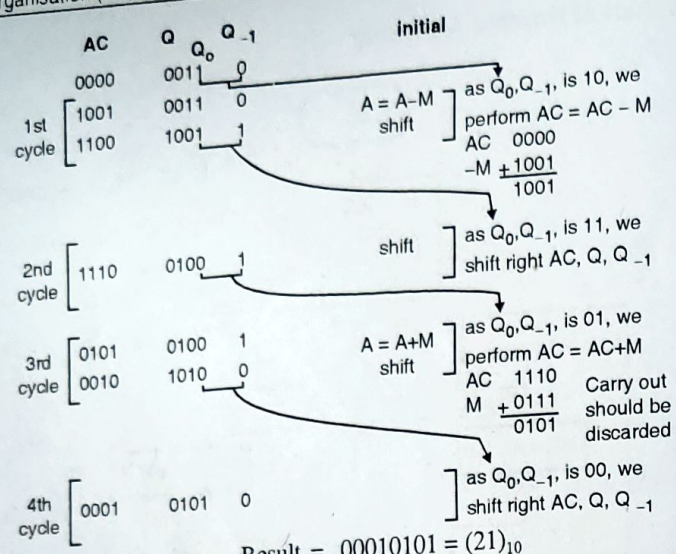
Solution :

$(7)_{10} = (0111)_2$ Multiplicand (M)

$(-7)_{10} = (1001)_2$ 2's complement of $(7)_{10}$ (– M)

$(3)_{10} = (0011)_2$ Multiplier (Q)

- Multiplication will require 4 cycles as the register size n = 4-bit.
- AC-M is same as AC + 2's complement of M.



Result = $00010101 = (21)_{10}$

Example 18 :

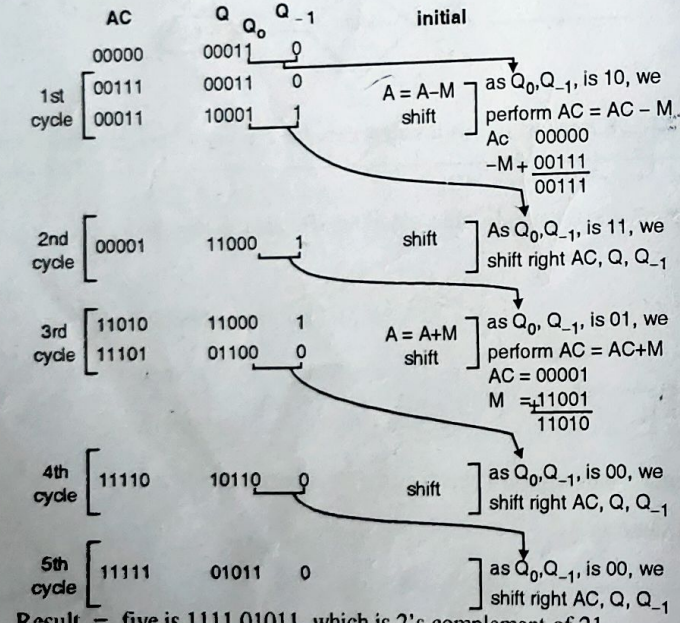
Multiply – 7 and 3 using Booth's algorithm. Register size = 5 bits.

Solution :

$(7)_{10} = (00111)_2$ (– M) 2's complement of M

$\therefore \quad (-7)_{10} = (11001)_2$ Multiplicand (M)

$(3)_{10} = (00011)_2$ Multiplicand (Q)



Result = five is 1111 01011, which is 2's complement of 21

Example 19 :

Multiply – 7 and – 3 using Booth's algorithm. Register size = 4 bits.

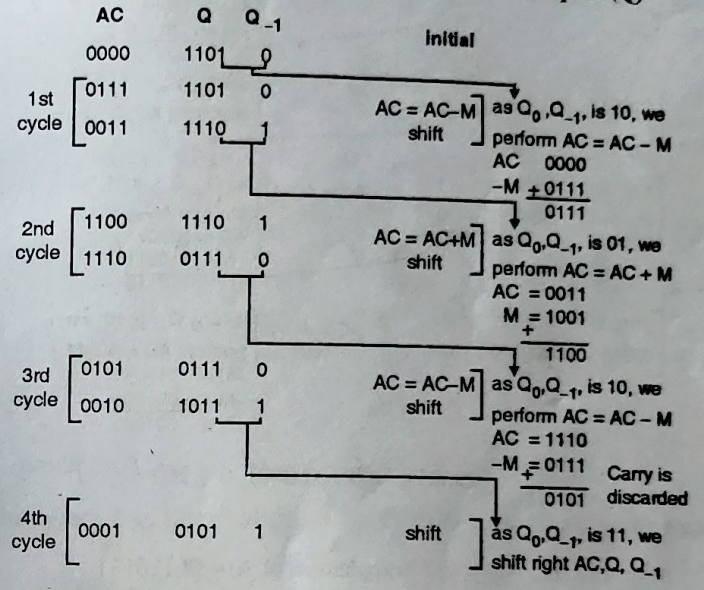
Solution :

$(7)_{10} = (0111)_2$ (– M) 2's complement of M

$(-7)_{10} = (1001)_2$ Multiplicand (M)

$(3) = (0011)_2$

$(-3) = (1101)_2$ Multiplier (Q)



Result = 0001 0101 = $(21)_{10}$

Example 20 :

Explain Booth's algorithm to multiply the following pair of signed 2's complement numbers :

A = 110011 multiplicand

B = 101100 multiplier **(May 2005, Dec. 2006)**

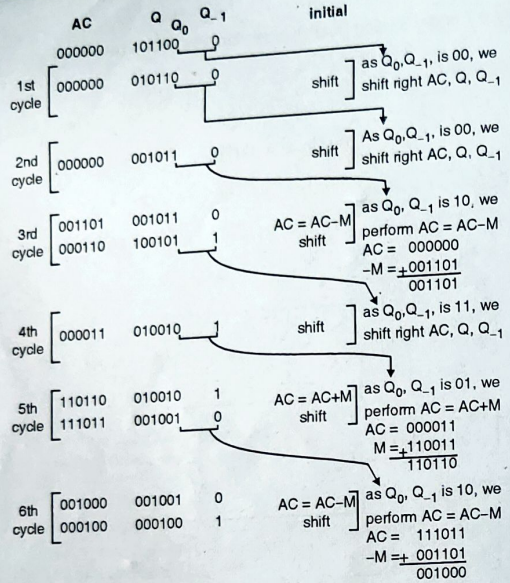
Solution :

Multiplicand (M) = 110011

2's complement of multiplicand = 001101 = (– M)

Multiplier Q = 101100

Multiplication will require 6 cycles as the register size n = 6-bits.



Result = 000 100 000100 = 260

Verification of result :

$A = 110011, -A = (2\text{'s complement of } A) = 001101$

$= (13)_{10}$

$\therefore A = 110011 = (-13)_{10}$

$B = 101100, -13 = (2\text{'s complement of } 13) = 010100$

$= (20)_{10}$

$\therefore B = 101100 = (-20)$

$\therefore A \times B = (-13) \times (-20) = 260$

Example 21 :

Using Booth's algorithm multiply the followings :

Multiplicand = + 15

Multiplier = – 6 **(Dec. 2005)**

Solution :

Since, the representation of signed 15, requires 5 bits to represent (4 bits for magnitude, 1 bit for sign) :

Register size, n = 5

$(15)_{10} = (01111)_2$ Multiplicand (M)

$(-15)_{10} = (10001)_2$ 2's complement of M = (– M)

$(6)_{10} = (00110)_2$

$\therefore (-6)_{10} = (11010)_2$ Multiplier (Q)

AC	Q	Q_{-1}		
00000	11010	0	Initial	
00000	01101	0	Shift	1st cycle
10001	01101	0	AC = AC – M	
11000	10110	1	Shift	2nd cycle
00111	10110	1	AC = AC + M	
00011	11011	0	Shift	3rd cycle
10100	11011	0	AC = AC – M	
11010	01101	1	Shift	4th cycle
11101	00110	1	Shift	5th cycle

Result = 11101 00110 [1110100110 = – 0001011010 = – 90]

2.3.4.3 Hardware Implementation of Booth's Algorithm :

May 2006

- The Booth's algorithm can be implemented as shown in Fig. 2.14. The circuit is similar to circuit for multiplication of positive numbers.
- It consists of n-bit adder, shift, add subtract control logic and registers AC, M, Q and Q_{-1}.
- Multiplier and multiplicand are loaded into registers Q and M respectively. Registers AC and Q_{-1} are initially set to 0.
- The n-bit adder performs addition of two inputs. One input is the AC register and the other input is either M or $\overline{M}$.
- In case of addition, $\overline{\text{Add}}$ /sub line is 0, therefore $C_0 = 0$ and multiplicand is directly applied as a second input to the n-bit adder.

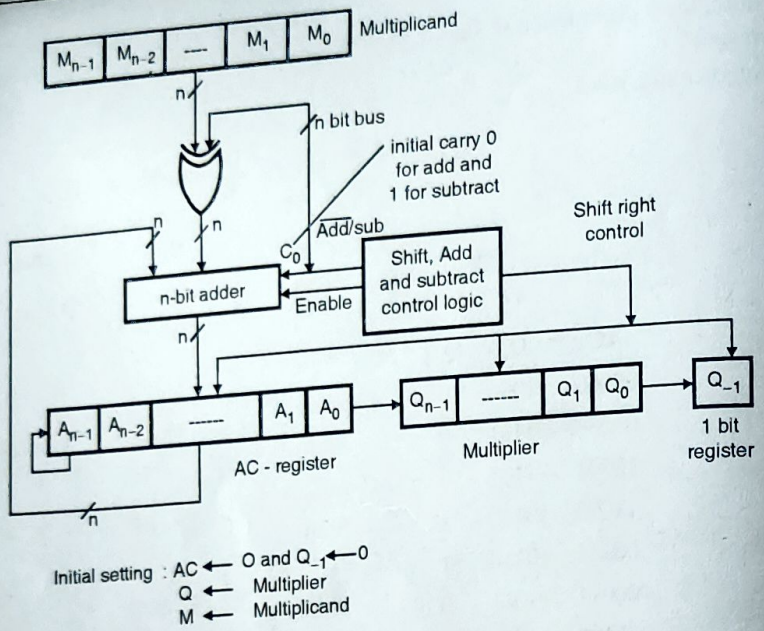


Fig. 2.14 : Hardware implementation of Booth's algorithm

- In case of subtraction, $\overline{\text{Add}}$/sub line is 1, therefore $C_0 = 1$ and multiplicand is complemented and then applied to the n-bit adder. As a result, the 2's complement of multiplicand (M) is added to the register AC.

2.3.5 Bit-pair Recording of Multipliers (A Fast Multiplication Method) :

Above technique halves the maximum number of summands. Bit-pair recording technique is derived directly from the Booth's algorithm. In Booth's algorithm, multiplier is examined two bits at a time, starting from the right. Table given below gives operation based on pair of bits.

Q_0	Q_{-1}	Operation	Notation
0	0	No add/subtract	0
1	1	No add/subtract	0
0	1	Add	+1
1	0	Subtract	−1

'0' stands for no operation. + 1 stands for addition and − 1 stands for subtraction.

Fig. 2.15 : Operations as per Booth's algorithm

Multiplier (Q) = $27 = (11011)_2$

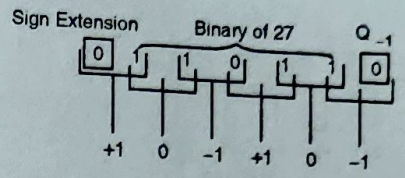
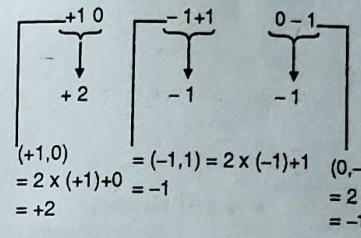


Fig. 2.16 : Operations for the multiplier = 27, using 0, + 1 and − 1 notation

In bit-pair recording, two adjacent operations of Booth's algorithm are combined together.



The weight of the i^{th} bit of a binary number is 2 times the weight of the $(i-1)^{th}$ bit.

Let us try to understand operations + 2, − 2, + 1, − 1, 0 on the data M = 9 and register size n = 5 bits.

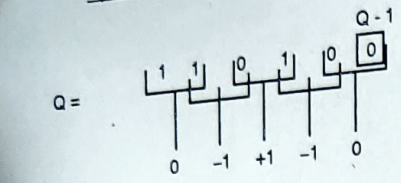
M = 9 = $(01001)_2$ = 00000 01001
(10 bit representation of '9' as the register size = 5)

−9 = 2's complement of 9 = 11111 10111

Operation	Value (9)	Multiplicand
0	00000 00000	
+1	00000 01001	
−1	11111 10111	
+2	00000 10010	← data is left shifted
−2	11111 01110	← −9 is left shifted

01001 (+9) $(6)_{10} = (00110)_2$
× 11010 (−6) $\therefore (-6)_{10} = (11010)_2$

Operations as per booth's algorithm



$\Downarrow$

0 1 0 0 1
0 -1 +1 -1 0

$\Downarrow$

0 1 0 0 1
0 -1 +1 -1 0

Pairing of operations

$\Downarrow$

0 1 0 0 1
0 -1 -2
0 1 0 0 1
0 -1 -2

Using multiplier bit-pair recording multiplication requires only half the number of Summands

0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	0	1	1	1		
0	0	0	0	1	0	0	1			
1	1	1	0	1	1	1				
0	0	0	0	0	0					
1	1	1	1	0	0	1	0	1	0	= (-54)

Results is 2's complement of 54

1	1	1	1	1	0	1	1	1	0	
1	1	1	1	0	1	1	1			
0	0	0	0	0	0					
1	1	1	1	0	0	1	0	1	0	= (-54)

Example 22 :

Implement multiplication of the following pair of signed 2's complement numbers using bit-pair recording.

A = 110011 multiplicand

B = 101100 multiplier

(May 2005)

Solution :

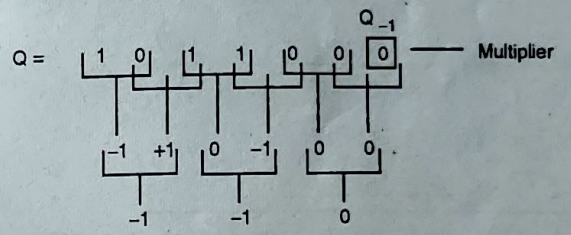
Multiplicand M = 111111 110011

As n = 6, number is represented in 12 bits

As the number given is a negative number, it is extended by 1's.

– M = 2's complement of M = 000000 001101

Operation	Value (M)	
0	000000 000000	
+1	111111 110011	
–1	000000 001101	
+2	111111 100110	← M is left shifted
–2	000000 011010	← – M is left shifted



						1	1	0	0	1	1	– Multiplicand	
							–1		–1		0		
0	0	0	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	1	1	0	1				
+ 0	0	0	0	1	1	0	1						
0	0	0	1	0	0	0	0	0	1	0	0	= (260)	

Example 23 :

Multiply the following numbers using bit-pair recording method.

Multiplicand 01111 (15)

Multiplier 10110 (– 10)

(May 2007)

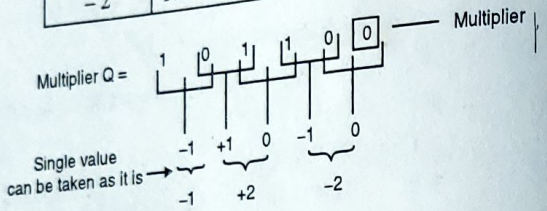
Solution :

Multiplicand (M) = 00000 01111

– M = 2's complement of M = 11111 10001

As n = 5, number is represented in 10 bits

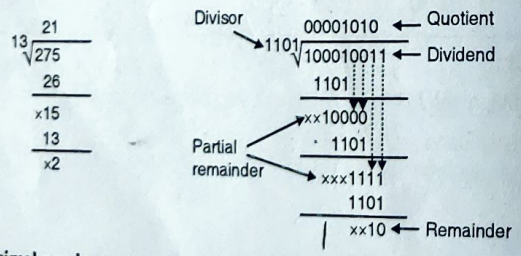
Operation	Value (M)	
0	00000 00000	
+1	00000 01111	
−1	11111 10001	
+2	00000 11110	← M is left shifted
−2	11111 00010	← − M is left shifted




```
                          0   1   1   1   – Multiplicand
                        –1  +2      –2
×
  1   1   1   1   1   0   0   0   1   0
      0   0   0   1   1   1   1   0
+     1   1   1   0   0   0   1
  ---------------------------------------
  1   1   0   1   1   0   1   0   1   0   = (– 150)
```

2.3.6 Integer Division :

Division is carried out on the same principle as integer. But it is some what more complex. The basis for the algorithm is the paper and pencil approach, and the operation involves repetitive shifting and subtraction. Fig. 2.17 shows an example of division of unsigned binary integers.



(a) Decimal numbers (b) Binary numbers

Fig. 2.17 : Paper and pencil approach for division

- Bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to divisor. Until this event occurs, 0's are placed in the quotient from left to right.
- When the divisor becomes greater than or equal to the group of bits of dividend under examination, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend.
- The result so produced is referred to as a partial remainder.
- Additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor.
- Now, the divisor is subtracted from this number to produce a new partial remainder.
- The process continues until all the bits of the dividend are exhausted.

2.3.6.1 Restoring Division :

Dec. 2005, Dec. 2006, May 2007

Division algorithm can be implemented using a machine.

Algorithm :

Register, M ← Divisor
Register, Q ← Dividend
Register, AC ← 0
Register, C ← 0 [1-bit carry register]
Count ← n [size of the register]

} Initial values

1. Shift C, AC, Q left by 1 bit.
2. AC ← AC – M [subtraction and checking the sign can be used for comparison]
3. If the sign of AC is negative (i.e.C = 1), set Q_0 to 0 and add M back to AC; otherwise, set Q_0 to 1.

Fig. 2.18 gives the flowchart for the above algorithm.

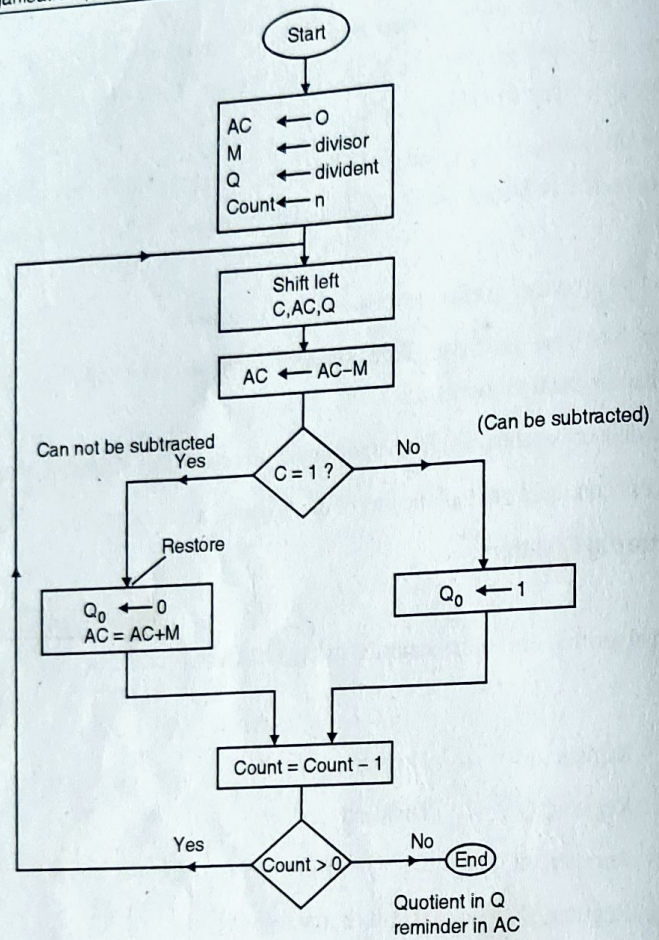


Fig. 2.18 : Flowchart for unsigned binary division (Restoring division)

Example 24 :

Perform division of the following positive numbers using restoring division.

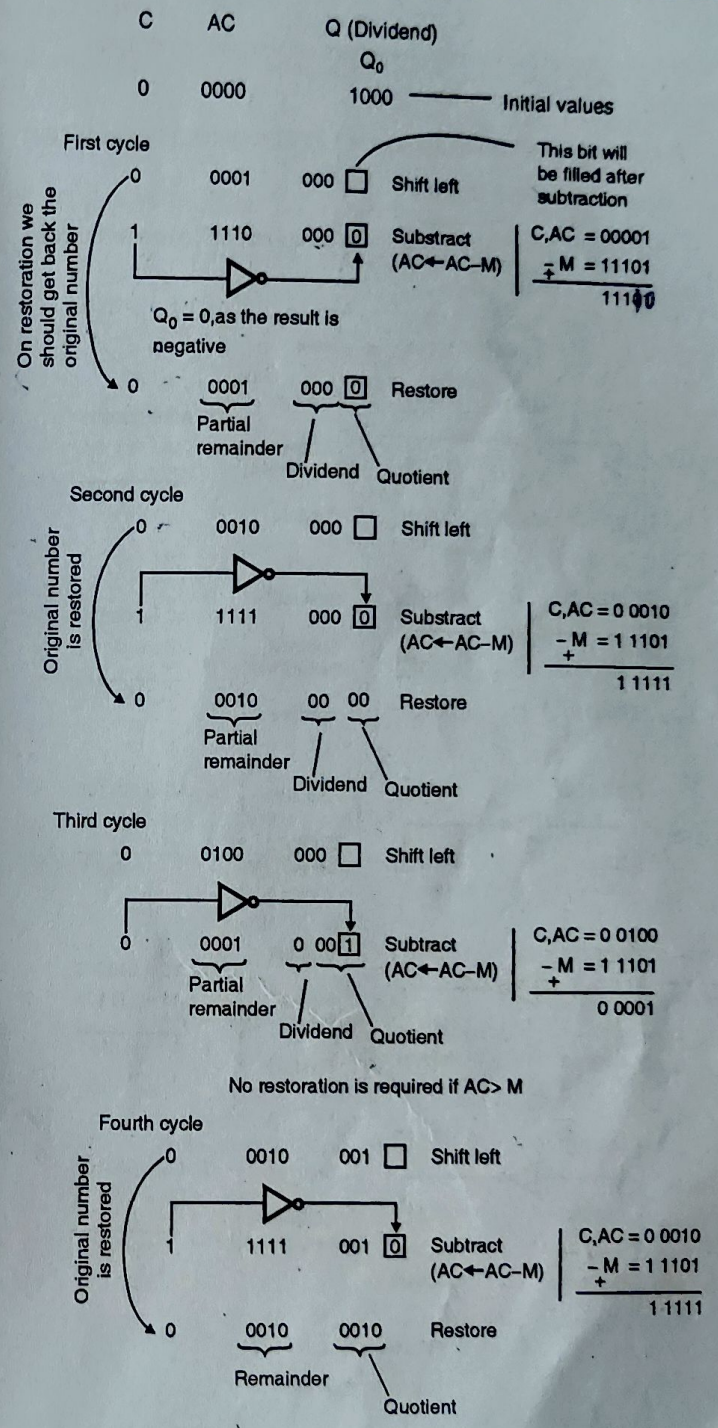
1000 ÷ 11, register size, n = 4 bits.

Solution :

Divisor M = 0 0 0 1 1 and – M = 1 1 1 0 1

(n + 1 bits) (n + 1 bits)

One additional bit has been taken to handle borrow. Borrow (C) will be set whenever M > AC and we perform AC = AC – M.



Example 25 :

Perform division of the following numbers using restoring division, **(Dec. 2005)**

Dividend = 17; Divisor = 03

Solution : Dividend Q = 17 = $(10001)_2$, n = 5 as 17 requires 5 bits of storage

Divisor M = 03 = 000011

n + 1 bits – one additional bit to handle borrow

– M = 111101

	C	AC	Q			
	0	00000	10001	← Intial		
1st cycle	0	00001	0001□	Shift left	C , AC = 000001	
	1	11110	0001 0	Subtract AC←AC –M	–M = 111101	+
	0	00001	00010	Restore		111110
2nd cycle	000010		0010□	Shift left	C , AC = 000010	
	111111		0010 0	Subtract AC←AC –M	–M = 111101	+
	000010		00100	Restore		111111
3rd cycle	000100		0100□	Shift left	C , AC = 000100	
	000001		0100 1	Subtract AC←AC –M	–M = 111101	+
						000001
4th cycle	000010		1001□	Shift left	C , AC = 000010	
	111111		1001 0	Subtract AC←AC –M	–M = 111101	+
	000010		1001 0	Restore		111111
5th cycle	000101		0010□	Shift left	C , AC = 000101	
	000010		0010 1	Subtract AC←AC –M	–M = 111101	+
	Reminder		Quotient			000010

Reminder = 00010 = 2

Quotient = 00101 = 5

Example 26 :

Perform division of the following numbers using restoring division.

1100 ÷ 11 **(May 2005, Dec. 2006)**

Solution : Dividend Q = 1100

Divisor M = 00011

n + 1 bits ← one additional bit to handle borrow

– M = 11101

	C	AC	Q			
	0	0000	1100	Intial		
1st cycle	0	0001	100□	Shift left subtract AC←AC –M	C , AC = 00001	
	1	1110	100 0		–M = 11101	+
	0	0001	100 0	Restore		11110
2nd cycle	0	0011	000□	Shift left subtract AC←AC –M	C , AC = 00011	
	0	0000	000 1		–M = 11101	+
						00000
3rd cycle	0	0000	001□	Shift left Subtract AC←AC –M	C , AC = 00000	
	1	1101	001 0		–M = 11101	+
	0	0000	001 0	Restore		11101
4th cycle	0	0000	010□	Shift left	C , AC = 00000	
	1	1101	010 0	Subtract AC←AC –M	–M = 11101	+
	0	0000	010 0	Restore		11101
	Remainder		Quotient			

∴ Remainder = 0000 = 0

Quotient = 0100 = 4

Example 27 :

Perform division of the following numbers using restoring division algorithm.

A : 1100 B : 0100

Solution : Dividend Q = 1100

Divisor M = $\underbrace{0\,0100}$

n + 1 bits – one additional bit to handle borrow

– M = 11100

	C	AC	Q		
	0	0000	1100	Intial	
1st cycle	0	0001	100□	Shift left	C , AC = 00001
	1	1101	100[0]	Subtract AC←AC –M	–M = 11100
	0	0001	100[0]	Restore AC←AC +M	+ 11101
2nd cycle	0	0011	000□	Shift left	C , AC = 00011
	1	1111	000[0]	Subtract AC←AC –M	–M = 11100
	0	0011	000[0]	Restore	+ 11111
3rd cycle	0	0110	000□	Shift left	C , AC = 00110
	0	0010	000[1]	Subtract AC←AC –M	–M = 11100
					+ 00010
4th cycle	0	0100	001□	Shift left	C , AC = 00100
	1	0000	001[1]	Subtract AC←AC –M	–M = 11100
		Remainder	Quotient		+ 00000

Remainder = 0000

Quotient = 0011

Example 28 :

Perform division of the following numbers using restoring division.

Dividend 1011

Divisor 0011

Solution :

Dividend Q = 1011

Divisor M = $\underbrace{0\,0011}$.

n + 1 bits – one additional bit to handle borrow

– M = 11100

	C	AC	Q		
	0	0000	1011	Initial	
1st cycle	0	0001	011□	Shift left	C , AC = 00001
	1	1110	011[0]	Subtract AC←AC –M	–M = 11101
	0	0001	011[0]	Restore	+ 11110
2nd cycle	0	0010	110□	Shift left	C , AC = 00010
	1	1111	110[0]	Subtract AC←AC –M	–M = 11101
	0	0010	110[0]	Restore	+ 11111
3rd cycle	0	0101	100□	Shift left	C , AC = 00101
	0	0010	100[1]	Subtract AC←AC –M	–M = 11101
					+ 00010
4th cycle	0	0101	001□	Shift left	C , AC = 00101
	0	0010	001[1]	Subtract AC←AC –M	–M = 11101
		Remainder	Quotient		+ 00010

Remainder = 0010 = 2

Quotient = 0011 = 3

Hardware implementation of binary division :

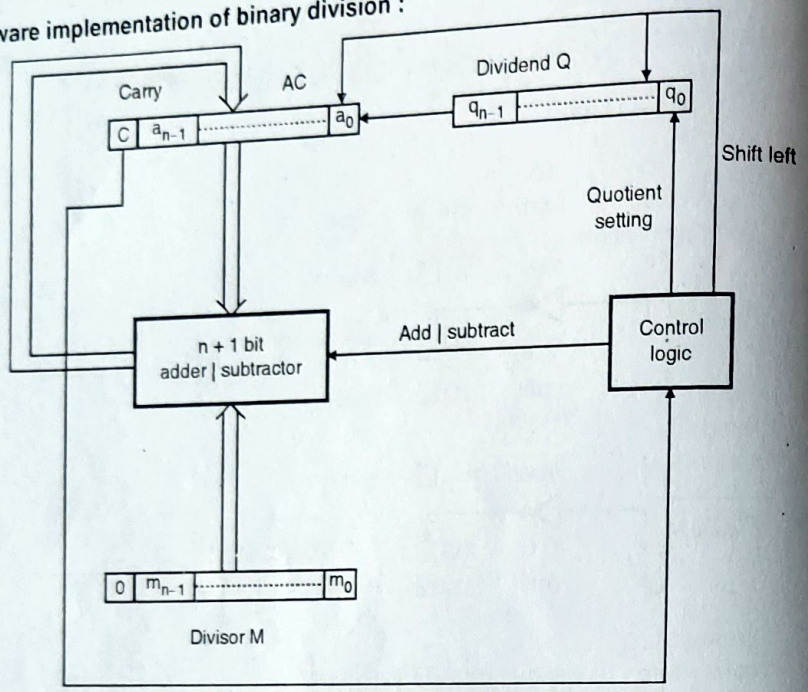


Fig. 2.19 : Circuit for binary division [Both restoring and non-restoring]

- An n-bit positive divisor is loaded into register M.
- An n-bit positive dividend is loaded into register Q.
- Register AC(A) is set to 0.
- Initial carry C is set to 0.
- After the division is complete, the n-bit quotient is in register Q and the remainder is in register AC.

2.3.6.2 Non-restoring Division Technique :

May 2006, Dec. 2006

The algorithm of restoring division can be improved by avoiding **restoring** after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative.

Algorithm for non-restoring division :

Step 1 : Do the following n times :

If the sign of AC is positive (C = 0), then shift C, AC and Q left one bit position and subtract M from AC.

else

Shift C, AC and Q left one bit position and add M to AC.

If the sign of AC is positive then

Set q_0 to 1

else

Set q_0 to 0

Step 2 : If the sign of AC is negative then add M to AC.

Step 2 is needed to leave the proper positive remainder in AC at the end of n cycles.

Flowchart for non-restoring division operation :

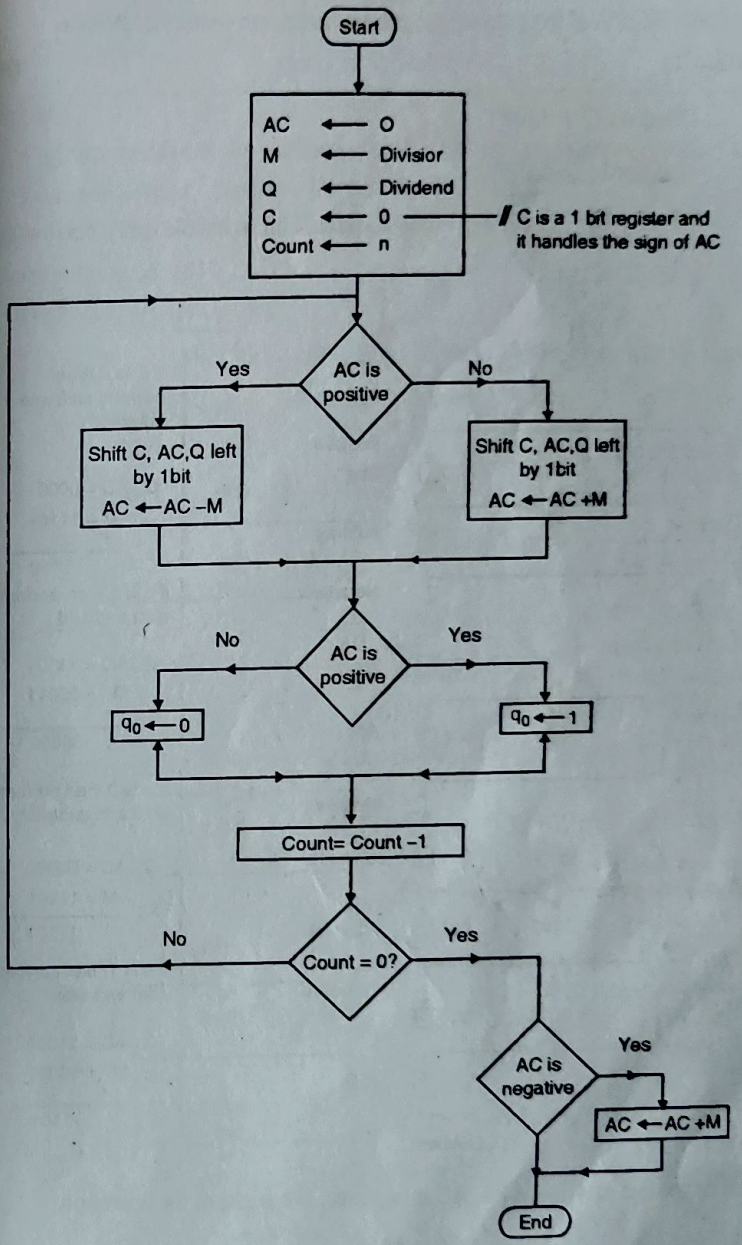


Fig. 2.20 : Flowchart for non-restoring division

Flowchart for non-restoring division is given in the Fig. 2.20. Logic circuit for both restoring and non-restoring division can be handled on the circuit given in Fig. 2.19.

Example 29 :

Perform division of the following numbers using non-restoring division. **(May 2005)**

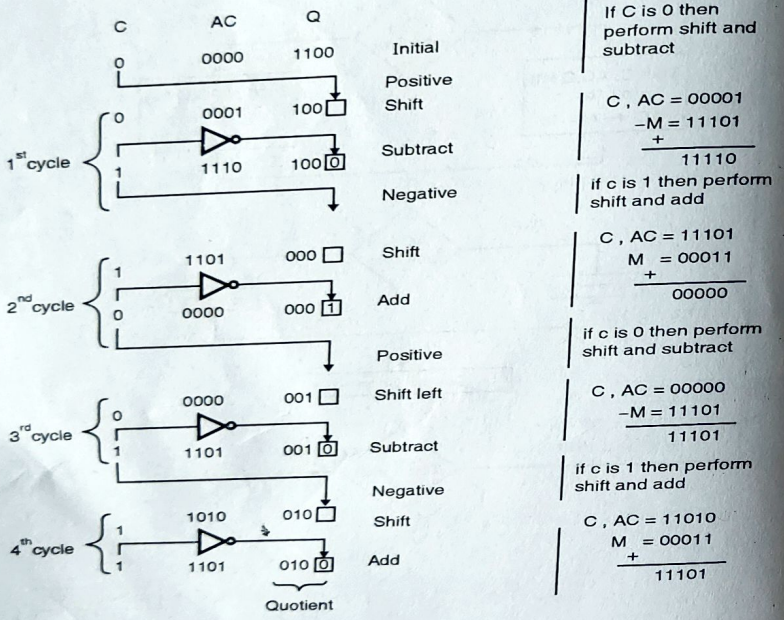
1100 ÷ 11

Solution : Dividend Q = 1100

Divisor M = 0 0011,

n + 1 bits – one additional bit to handle carry/borrow/sign

– M = 11101



Since, at the end of 4(n) cycles, AC is negative. We perform the operation :

AC ← AC + M

C, AC = 11101

M = 00011

00000 ← Remainder

∴ Remainder = 0000

Quotient = 0100 = 4

Example 30 :

Perform division of the following numbers using non-restoring division algorithm.

Dividend = 1011; Divisor = 0011 **(May 2006)**

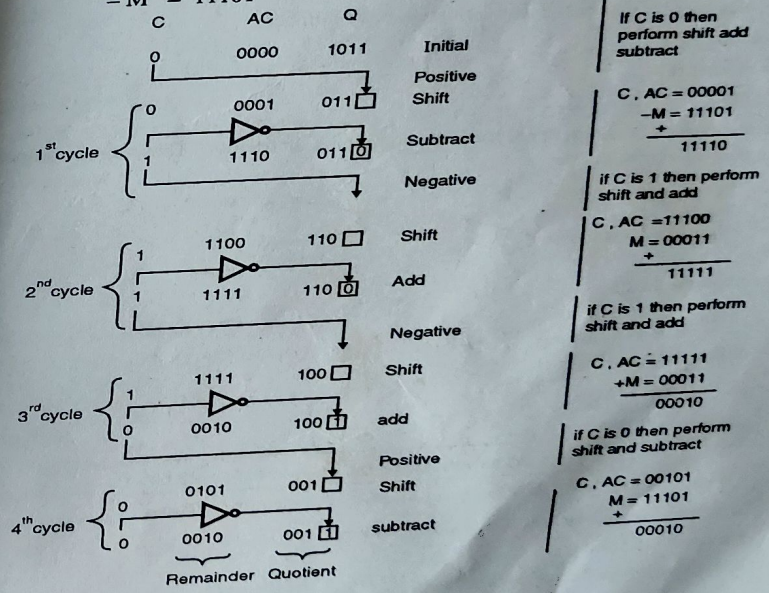
Solution :

Dividend (Q) = 1011

Divisor (M) = 00011

n + 1 bits – one additional bit to handle carry/borrow/sign

– M = 11101



Quotient = 0011 = 3

Remainder = 0010 = 2

2.4 Floating Point Numbers :

The range of numbers that can be represented by a fixed-point number is insufficient for many applications. In scientific applications, very large and very small numbers are encountered. Scientific notation permits us to represent such numbers using relatively few digits.

For example,

2.5×10^{10}

represents a fixed point integer 25000000000.

The floating-point codes used in computers are represented in binary.

Format of representation :

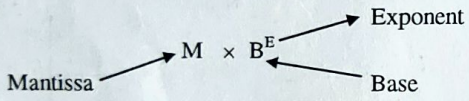
Three numbers are associated with a floating point number :

(1) A mantissa (M)

(2) An exponent (E)

(3) Base (B)

The mantissa M is also referred to as the significant or fraction. These three components together represent the real number.



For example, in 2.5×10^{10}

Mantissa = 2.5

Exponent = 10

Base = 10

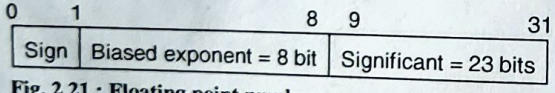


Fig. 2.21 : Floating point number representation (Binary)

In a typical representation of a floating point number :

(1) Exponent is biased.

(2) Mantissa or significand is normalized.

Biased exponent :

Exponent is represented using an excess-128 code. An 8 bit binary number represents values from 0 to 255. However, as we are adding 128 in the biased exponent, the actual exponent values represented will be – 128 to 127.

Exponent value	Exponent value after biasing (+ 128)
– 128	$0\ (000\ 00000)_2$
– 127	$1\ (0000\ 0001)_2$
0	$128\ (1000\ 0000)_2$
127	$255\ (1111\ 1111)_2$

Advantages of biasing :

Very large or small exponents can easily be represented.

For example,

If the exponents are in the – 1050 to – 900, we can select the biasing factor as 1050. With 1050 as biasing,

– 1050 will be represented as – 1050 + 1050 = 0

– 900 will be represented as – 900 + 1050 = 150

Normalized mantissa :

A binary floating point number is represented in a normalized form, that is, the number is of the form ± 0. (Significand starting with non-zero bit) X 2 ± (exponent value).

Example,

Binary number	Its normal form
11.101	$.11101 \times 2^2$
.00101	$.101 \times 2^{-2}$
1.01×2^5	$.101 \times 2^6$

Advantages of normalization :

As for a normalized mantissa, the left most bit can not be zero, therefore, it has to be 1. Thus, it is not necessary to store this first bit and it is assumed implicitly for the number. Therefore, a 23 bit mantissa can represent 23 + 1 = 24 bit significand.

Example 31 :

Represent $(13.54)_{10}$ in 32 bit register with

Exponent = 8 bits

Mantissa = 23 bits

Exponent is biased with biasing of 128 and Mantissa is normalised.

Solution :

Step 1 : Converting 13.54 to its equivalent binary form.

$13 = 1101$

$.54 \times 2 = 1.08$

$.08 \times 2 = 0.16$

$.16 \times 2 = 0.32$

$.32 \times 2 = 0.64$

$.64 \times 2 = 1.28$

$.28 \times 2 = 0.56$

$.56 \times 2 = 1.12$

$.12 \times 2 = 0.24$

$.24 \times 2 = 0.48$

$.48 \times 2 = 0.96$

$.96 \times 2 = 1.92$

$.92 \times 2 = 1.84$

$.84 \times 2 = 1.68$

$.68 \times 2 = 1.36$

$.36 \times 2 = 0.72$

$.72 \times 2 = 1.44$

$.44 \times 2 = 0.88$

$.88 \times 2 = 1.76$

$.76 \times 2 = 1.52$

$.52 \times 2 = 1.04$

$.04 \times 2 = 0.08$

$.08 \times 2 = 0.16$

$.16 \times 2 = 0.32$

$.32 \times 2 = 0.64$

Step 2 : Biasing

$\therefore 13.54 = 1101.10001010001111010111000 0$

$= .11011000101000111101011100000 \times 2^4$

Exponent $= 4$

Exponent after biasing $= 128 + 4$

$= 132$

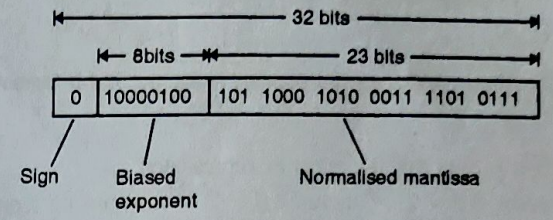
$(132)_{10} = (10000100)_2$

Step 3 : Normalisation :

Mantissa = .1101 1000 1010 0011 1101 0111

First bit is not stored as it is assumed to be implicit (hidden).

Mantissa after normalization = .101 1000 1010 0011 1101 0111



Representation of $(13.54)_{10}$ in floating point format.

Disadvantages of normalisation :

Minimum value of significand :

The implicit first bit as 1 followed by 23 0's.

.1000 0000 0000 0000 0000 0000 (1 is hidden)

Decimal equivalent = $1 \times 2^{-1} = 0.5$

Maximum value of significand :

The implicit first bit 1 followed by 23 1's.

.1111 1111 1111 1111 1111 1111

Decimal equivalent :

Binary : 0.1111 1111 1111 1111 1111 1111

$+$ 0.0000 0000 0000 0000 0000 0001 $= 2^{-24}$

1.0000 0000 0000 0000 0000 0000

$\therefore$ Maximum value of significand $= 1 - 2^{-24}$

Therefore, in normalized mantissa and biased exponent form, the format of Fig. 2.21 can represent binary floating point number in the range.

Lowest negative number : Maximum significand and maximum exponent

$= -(1 - 2^{-24}) \times 2^{127}$

Highest negative numbers = Minimum significand and minimum exponent

$= -0.5 \times 2^{-128}$

Lowest positive number : 0.5×10^{-128}

Highest positive number : $(1 - 2^{-24}) \times 10^{127}$

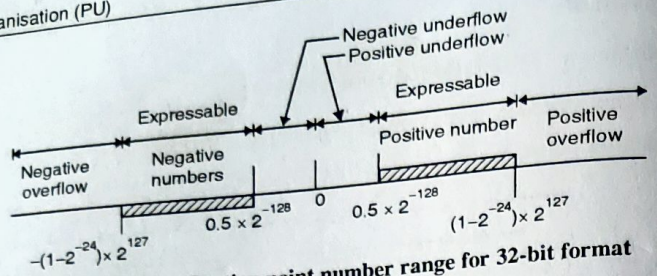


Fig. 2.22 : Binary floating point number range for 32-bit format

Disadvantages :

(1) From the Fig. 2.22, it is clear that 0 can not be represented.

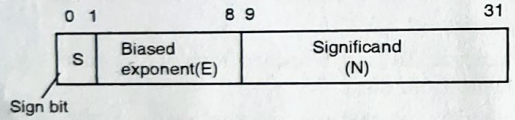
(2) A number in the range $0.5 \times 2^{-128} < x > - 0.5 \times 2^{-128}$ can not be represented as it will cause an underflow.

(3) ∞ can not be represented.

2.4.1 IEEE Standard for Floating Point Numbers :

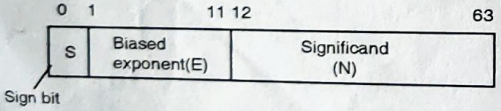
May 2005, Dec. 2006, May 2007

- Representation of floating point number discussed in section 2.4 has many subtle problems.
- IEEE floating point standards addresses a number of such problems.
- Zero has definite representation in IEEE format.
- $\pm \infty$ has been represented in IEEE format. A + ∞ indicated that the result of an arithmetic expression is too large to be stored.
- If an underflow occurs, implying that a result is too small to be represented as a normalized number, it is encoded in a denormalized scale.
- Fig. 2.23 gives the representation of floating point numbers.



(a)Single precision = 32 bits

- base = 2
- Significand is in normalized form i.e the first bit is 1 and it is hidden.
- S is sign bit.



(b)Double precision = 64 bits

Fig. 2.23 : IEEE standard format

Single precision (32 bits) :

	Exponent (E)	Significand (N)	Value/comments
	255	Not equal to 0	Does not represent a number
	255	0	$-\infty$ or $+\infty$ depending on sign bit
Normalized scale	$0 < E < 255$	Any	$\pm (1.N)\ 2^{E-127}$
Denormalized scale	0	Not equal to 0	$\pm (0.N)\ 2^{-128}$
	0	0	± 0 depending on sign bit

Double precision (64 bits) :

	Exponent (E)	Significand (N)	Value/comments
	2047	Not equal to 0	Does not represent a number
	2047	0	$-\infty$ or $+\infty$ depending on sign bit
Normalized scale	$0 < E < 2047$	Any	$\pm (1.N)\ 2^{E-1023}$
Denormalized scale	0	Not equal to 0	$\pm (0.N)\ 2^{-1022}$
	0	0	± 0 depending on sign bit

Fig. 2.24 : Values of floating point numbers as per IEEE format

Example 32 :

Represent $(178.1875)_{10}$ in single and double precision floating point format. **(Dec. 2005)**

Solution :

Convert given decimal number into its equivalent binary

$$178 = 10111100$$

$$.1875 \times 2 = 0.3750$$

$$.3750 \times 2 = 0.7500$$

$$.7500 \times 2 = 1.500$$

$$.500 \times 2 = 1.000$$

$$\therefore (178.1875)_{10} = (10111100.0011)_2$$

(a) Single precision format :

In IEEE format, the value of a number for given exponent (E) and significand (N) is given by $(1.N)\ 2^{E-127}$ in order to represent $(10111100.0011)_2$, we must convert it into the form $(1.N)\ 2^{E-127}$.

$101111.011 = 1.01111011 \times 2^5$

(Decimal is shifted left by five places)

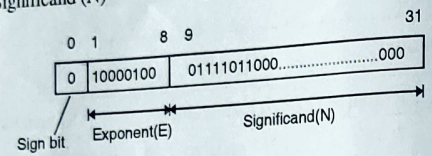
$= 1.01111011 \times 2^{132-127}$

Significand (N) exponent (E)

$5 = E - 127$

$\therefore E = 127 + 5 = 132$

$(132)_{10} = (10000100)_2$



(b) Double precision format :

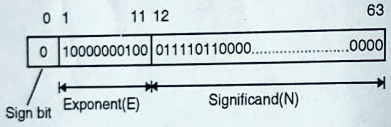
In IEEE double precision format, the value of a number for given exponent (E) and significand (N) is given by $(1.N)\ 2^{E-1023}$.

In order to represent $(10111100.0011)_2$, we must convert it into the form $(1.N)\ 2^{E-1023}$.

$101111.011 = 1.01111011 \times 2^5 = 1.01111011 \times 2^{1028-1023}$

Significand (N) Exponent (E)

$(1028)_{10} = (10000000100)_2$



Example 33 :

Represent $(309.1875)_{10}$ in single precision and double precision format. **(May 2006)**

Solution : Convert given decimal number into its equivalent binary.

$(309)_{10} = (100110101)_2$

$.1875 \times 2 = 0.3750$

$.3750 \times 2 = 0.7500$

$.7500 \times 2 = 1.5000$

$.5000 \times 2 = 1.0000$

$\therefore 309.1875 = 100110101.0011$

(a) Single precision format :

In IEEE single precision format, the value of a number for given exponent (E) and significand (N) is given by $(1.N) \times 2^{E-127}$.

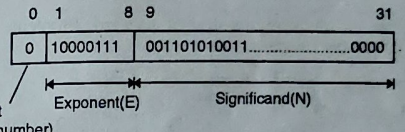
In order to represent 100110101.0011, we must convert it into the form $(1.N) \times 2^{E-127}$.

$100110101.0011 = 1.001101010011 \times 2^8$

$= 1.001101010011 \times 2^{135-127}$

Significand (N) Exponent (E)

$(135)_{10} = (10000111)_2$



(b) Double precision format :

In IEEE double precision format, the value of a number for given exponent (E) and significand (N) is given by,

$(1.N) \times 2^{E-1023}$

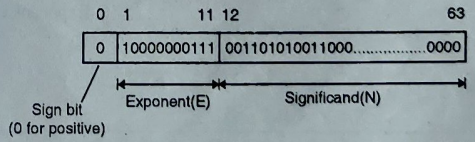
In order to represent 100110101.0011, we must convert it into the form $(1.N) \times 2^{E-1023}$.

$100110101.0011 = 1.001101010011 \times 2^8$

$= 1.001101010011 \times 2^{1031-1023}$

Significand (N) Exponent (E)

$(1031)_{10} = (10000000111)_2$



2.4.2 Floating Point Arithmetic :

A number of operations are performed on floating point numbers. Some of them are :

(1) Addition (2) Subtraction
(3) Multiplication (4) Division

For addition and subtraction, it is necessary that both the numbers must have the same exponent. This may require shifting the decimal point of one of the number to achieve alignment. Consider a decimal example in which we wish to add 2.4500×10^2 to 4.5100×10^5.

We rewrite 2.4500×10^2 as

$.0024500 \times 10^5$ and then perform the addition.

$$\begin{array}{r} 4.5100 \times 10^5 \\ +\ 0.0245 \times 10^5 \\ \hline 4.6145 \times 10^5 \end{array}$$

Multiplication and division are straight forward. Problems may arise as the result of these operations. These are :

- Exponent overflow
- Exponent underflow
- Significand underflow
- Significand overflow

2.4.2.1 Addition and Subtraction :

May 2005, May 2006, Dec. 2006

In floating point arithmetic, addition and subtraction are more complex than multiplication and division. Addition and subtraction operations are carried out in four basic phases :

(1) Check for zeros. (2) Align the significands
(3) Add or subtract the significands (4) Normalize the result

First number $X = M \times B^x$

Second number $Y = N \times B^y$

Arithmetic operations :

$$\left.\begin{array}{l} X + Y = (M \times B^{x-y} + N)\, B^y \\ X - Y = (M \times B^{x-y} - N)\, B^y \end{array}\right\} \text{Exponent } x \le \text{exponent } y$$

$$X * Y = (M * N)\, B^{x+y}$$

$$X \div Y = (M \div N) \times B^{x-y}$$

Fig. 2.25 : Floating point numbers and arithmetic operations

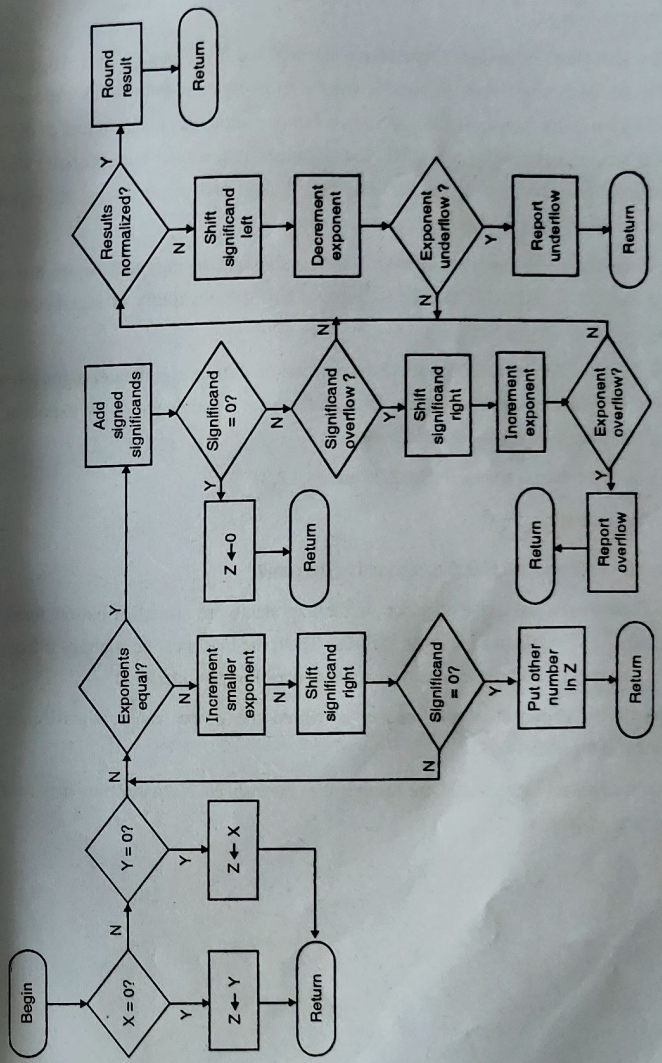


Fig. 2.26 : Floating point addition (Z = X + Y)

- Addition and subtraction are almost similar operations. In subtraction, sign of the second number is changed.
- In the next phase, exponents of the two numbers X and Y are made equal. Alignment is achieved by shifting either the smaller number to its right (increasing its exponent) or shifting the larger number to the left. Since either operation may result in loss of digits, it is the smaller number that is shifted. The alignment is achieved by repeatedly shifting the magnitude portion of the significand right 1 digit and incrementing the exponent until the two digits exponents are equal.
- Next, the two significands are added together, taking into account their signs. Since the sign may differ, the result may be 0. There is also the possibility of significand over flow.
- Next, result is normalized. Normalization consists of shifting significand digits left unitl the most significant digit is nonzero. Each shift causes a decrement of the exponent and thus could cause an exponent underflow.
- Detailed flowchart is shown in Fig. 2.26 and Fig. 2.27.

2.4.2.2 Multiplication :

- If either of the operand is 0, 0 is reported as the result.
- Next, exponents are added together. If the exponents are stored in biased form, the exponent sum would have doubled the bias. Thus, the bias must be subtracted from the sum. The result could cause either an exponent overflow or underflow.
- Next, if the exponent of the product is within the proper range, significands are multiplied together.
- After the product is calculated, the result is then normalized. Normalization may result in exponent underflow

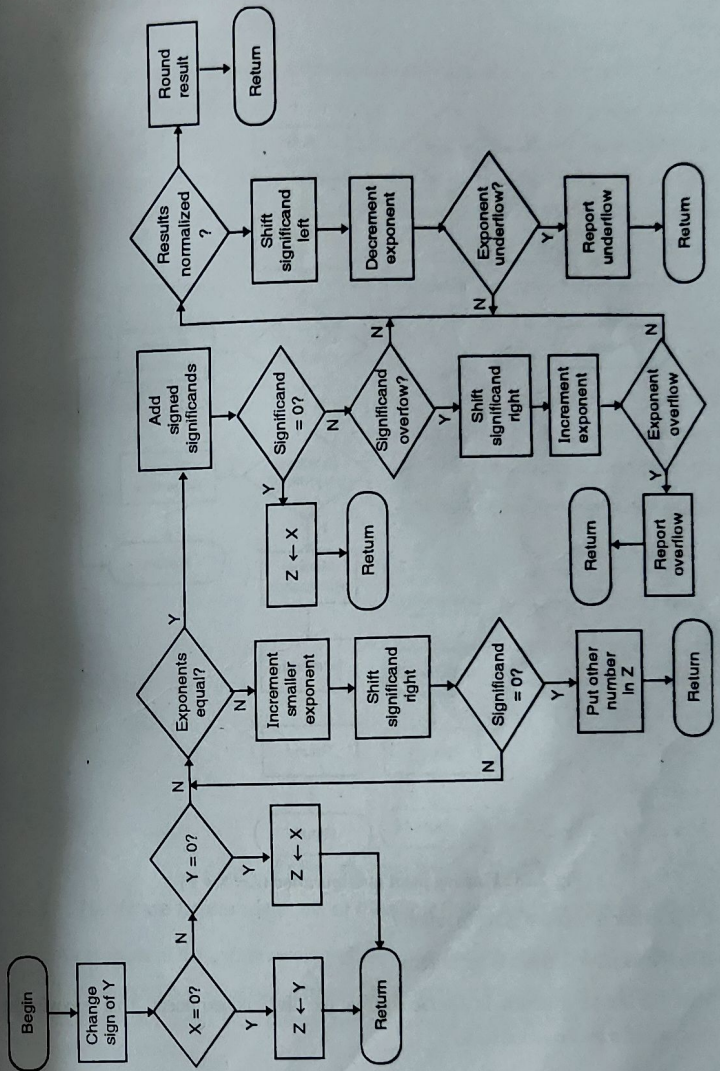


Fig. 2.27 : Floating point subtraction (Z = X – Y)

2.4.2.3 Division :

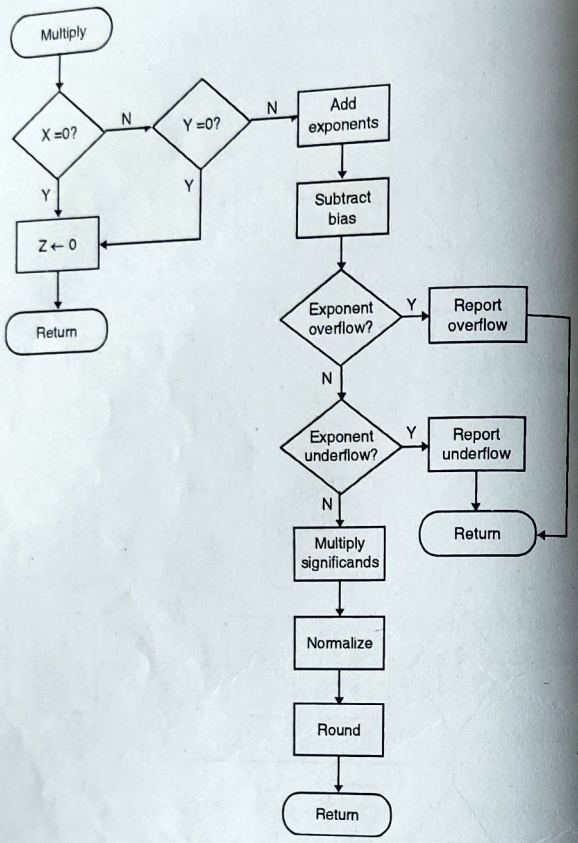


Fig. 2.28 : Floating point multiplication (Z = X * Y)

- If the divisor is 0, result is set to infinity.
- If the dividend is 0, result is set to zero.
- Next, the divisor exponent is subtracted from the dividend exponent. This removes the bias, which must be added back.

- Exponent is checked for underflow or overflow.
- Next, significands are divided.
- Normalization and rounding are carried out subsequently

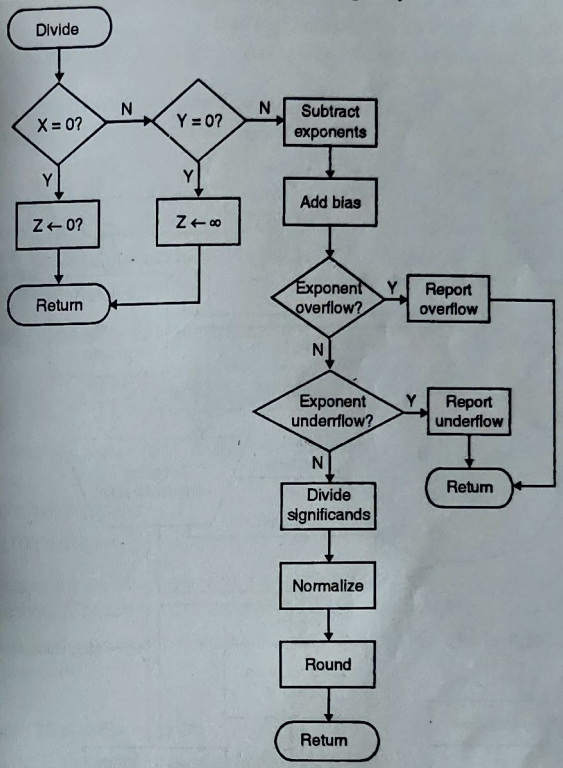


Fig. 2.29 : Floating point division (Z = X/Y)

2.4.3 Hardware Implementation of Floating Point Addition/Subtraction :

An example of the implementation of floating point operation is shown in the Fig. 2.30.

- The first step is to compare exponents to determine how far to shift the mantissa of the number with smaller exponent. The shift count value n is determined by the 8-bit subtractor circuit.

- The magnitude of the difference $E'_A - E'_B$ sent to the SHIFTER unit.
- The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. Therefore, the sign is sent to the SWAP unit.
- One of the mantissas is sent directly to the adder/subtractor whereas other mantissa is sent to the adder/subtractor after shifting by n bits.
- Multiplexer unit is used to select the exponent of the result. Exponent of the result E' is E'_A if $E'_A \geq E'_B$ or E'_B if $E'_A < E'_B$.

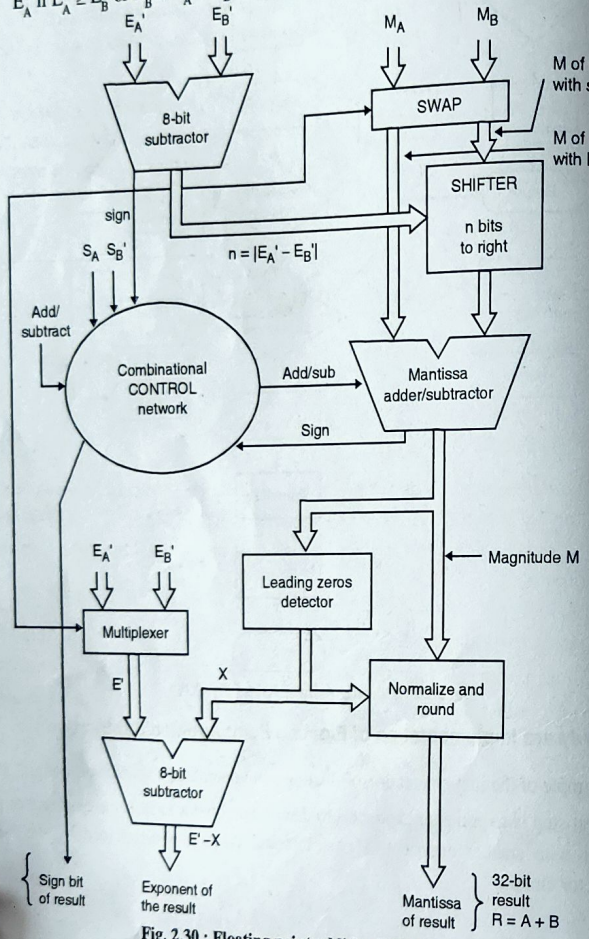


Fig. 2.30 : Floating point addition/subtraction unit

A : S_A = Sign bit of the number A
E'_A = Exponent of number A
M_A = Mantissa of number A

B : S_B = Sign bit of the number B
E'_B = Exponent of number B
M_B = Mantissa of number B

- The control logic determines whether the mantissas are to be added or subtracted. This is decided by the signs of the operands (S_A and S_B) and the operation (Add or subtract) that is to be performed on the operands.
- After Add/subtract, normalization is carried out on the mantissa M. The number of leading zeroes in M determines the number leading zeroes in M determine, the number of bit shifts, X, to be applied to M. The value of X is also subtracted from the tentative result exponent E' to generate the true result exponent, E'_R.

2.5 University Questions and Answers :

May 2005 : Total Marks 32

Q. 1 Explain Booth's algorithm to multiply the following pair of signed two's complement numbers :

A = 110011 multiplicand

B = 101100 multiplier

Also, implement the above using bit-pair recording and explain how it achieves faster multiplication. **(Section 2.3.4.1, Examples 20 and Example 22)** **(12 Marks)**

Q. 2 Compare restoring and non-restoring division algorithm with the help of the following binary numbers

1100 ÷ 11

(Example 26 and Example 29) **(12 Marks)**

Q. 3 Explain IEEE floating point number formats. **(Section 2.4.1)** **(4 Marks)**

Q. 4 Draw the flowchart for floating point addition. **(Section 2.4.2.1)** **(4 Marks)**

Dec. 2005 : Total Marks 26

Q. 5 Draw the flowchart for restoring division algorithm and solve the following using above algorithm.

Dividend = 17

Divisor = 3

(Section 2.3.6.1 and Example 25) **(10 Marks)**

Q. 6 Represent $(178.1875)_{10}$ in single and double precision floating point format. **(8 Marks)**
(Example 32)

Q. 7 Using Booth's algorithm multiply the following :
Multiplicand = + 15
Multiplier = – 6 **(8 Marks)**
(Example 21)

May 2006 : Total Marks 32

Q. 8 Draw the hardware implementation of Booth's algorithm and explain the same. **(8 Marks)**
(Section 2.3.4.2)

Q. 9 Represent $(309.1875)_{10}$ in single precision and double precision format. **(8 Marks)**
(Example 33)

Q. 10 Draw the flowchart for non-restoring division algorithm and perform division of the following numbers using non-restoring division algorithm :
Dividend = 1011
Divisor = 0011
(Section 2.3.6.2 and Example 30) **(10 Marks)**

Q. 11 Draw the flowchart for floating point addition and explain. **(Section 2.4.2.1)** **(6 Marks)**

Dec. 2006 : Total Marks 36

Q. 12 Compare restoring and non-restoring division algorithm. Perform division of the following numbers using restoring division algorithm :
Dividend = 1101
Divisor = 11
(Sections 2.3.6.1, 2.3.6.2 and Example 26) **(12 Marks)**

Q. 13 Describe the IEEE standards for single and double precision floating point numbers. **(Section 2.4.1)** **(6 Marks)**

Q. 14 Multiply the following pair of signed two's complement numbers using Booth's algorithm :
A : 110011
B : 101100
(Example 20) **(12 Marks)**

Q. 15 Explain and draw the flowchart for floating point subtraction. **(Section 2.4.2.1)** **(6 Marks)**

May 2007 : Total Marks 24

Q. 16 Draw the flowchart for Booth's algorithm and solve the following using bit-pair recording method :
Multiplicand = 01111
Multiplier = 10110
(Section 2.3.4.1 and Example 23) **(8 Marks)**

Q. 17 Draw flowchart for restoring division algorithm and perform division of the following numbers using restoring division algorithm.
A : 1100 B : 0100
(Section 2.3.6.1 and Example 27) **(10 Marks)**

Q. 18 Write short note on IEEE standards. **(Section 2.4.1)** **(6 Marks)**

□□□