BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 3

DCA2103

COMPUTER ORGANIZATION

Unit 2

Basic Arithmetic Operations

Table of Contents

1. INTRODUCTION

In the previous unit, you learned about the fundamentals of computers, its evolution, and other related concepts. In this unit you will learn the basic arithmetic concepts, signed numbers and complements and other related concepts like Booths Algorithm, Floating-point arithmetic, etc. BasicArithmetic or arithmetic's (from the Greek word ἀριθμός, arithmos "number") is the oldest and most elementary branch of mathematics, used by almost everyone, for tasks ranging from simple day-to-day counting to advanced science and business calculations. It involves the study of quantity, especially as the result of operations that combine numbers. In common usage, it refers to the simpler properties when using the traditional operations of addition, subtraction, multiplication, and division with smaller values of numbers. Professional mathematicians sometimes use the term (higher) arithmetic when referring to more advanced results related to number theory, but this should not be confused with elementary arithmetic.

In this unit you will study Integer Addition and Subtraction, Fixed and Floating point numbers, Signed numbers, Binary Arithmetic, and 1's and 2's Complements Arithmetic. You will also be introduced to the concepts like Booths Algorithm, Floating-point representations, IEEE standards, etc.

1.1 Objectives:

After studying this unit, you should be able to:

- *Explain integer addition and subtraction*
- *Explain Fixed and Floating point numbers*
- *Explain Signed numbers*
- *Discuss on 2'complement method*
- *Explain Booths Algorithm*
- *Discuss about Hardware Implementation for design*
- *Explain IEEE standards*
- *Discuss on floating-point arithmetic*
- *Explain the function of the accumulator*
- *Explain Shifts, Carry and Overflow operation*

2. INTEGER ADDITION AND SUBTRACTION

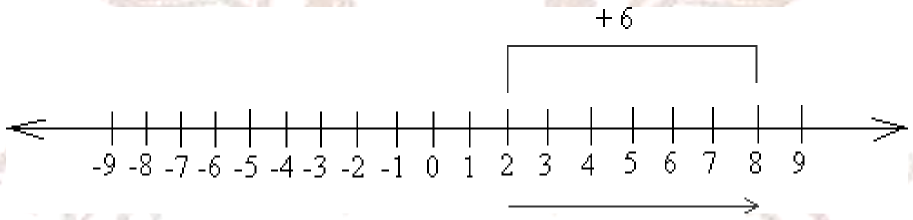
Digital computers use the binary number system, which has two digits 0 and 1. A binary digit is called a ***bit***. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also otherdiscrete symbols, such as decimal digits or letters of the alphabet. Since digital computers process binary numbers, it is important for us to understand the basic arithmetic operations of these binary numbers as well. First, we will introduce the very familiar arithmetic operations of decimal numbers called addition and subtraction.

Addition (+)

Addition is the basic arithmetic operation. In its simplest form, addition combines two numbers (i.e. it adds two numbers) and gives the sum of the numbers. The addition is represented by the symbol Plus (+). When addingintegers, use a number line to help you think through problems. It will be really helpful.

For example 2+6

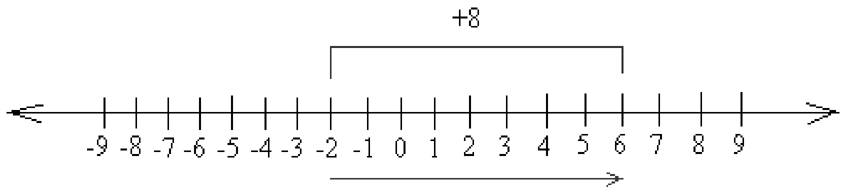
- Start at 2 and move 6 units to the right,
- Since you stopped at 8,
- The answer is 8.



Notice that you would get the same answer if you start at 6 and move 2. to the right.

Perform: -2 +8

- Start at 2 and move 8 units to the right.
- Since you end up at 6.
- The answer is 6



The identity element of addition is 0 that is, adding zero to any number yields that same number. Also, the inverse element of addition (the additive inverse) is the opposite of any number, that is, adding the opposite of any number to the number itself yields the additive identity, 0. For example, the opposite of 7 is −7, so 7 + (−7) = 0.

Examples of addition:

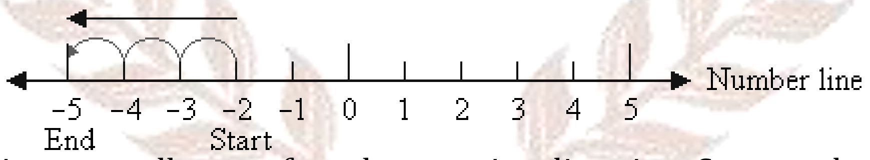
1) 12 + 8 =20
2) 12 + 12=24

Subtraction (−)

Subtraction is the opposite of addition. Subtraction is represented by the symbol minus (–). Subtraction finds the difference between two numbers, the minuend minus the subtrahend. If the minuend is larger than the subtrahend, the difference is positive; if the minuend is smaller than the subtrahend, the difference is negative; if they are equal, the difference is zero.

Subtracting a Positive Integer from a Negative Integer

Consider the value of (–2) – (3).



The minus sign, –, tells us to face the negative direction. So, to evaluate (–2) – (3), start at –2, face the negative direction and move3 units forwards.

$\therefore (-2) - (3) = -5$

We notice that:

$$(-2) + (-3) = (-2) - (3)$$

That is:

$$-2 + -3 = -2 - 3$$

This suggests that:

Adding a negative integer is the same as subtracting a positive integer.

$$\begin{aligned} \therefore (-2)+(-3) &= -2+-3 \\ &= -2-3 \\ &= -5 \end{aligned}$$

When we subtract a negative number, it is the equivalent of adding apositive number. Just remember, the negative of negative is positive.

Ex: 5– (-1) = 5 + 1 = 6

Self-Assessment Questions – 1

1. A binary digit is called a______________.
2. If the minuend is larger than the subtrahend, the difference is______________.

3. FIXED AND FLOATING-POINT NUMBERS

The term 'fixed point' refers to the corresponding manner in which numbers are represented, with a fixed number of digits after, and sometimes before, the decimal point. With floating-point representation, the placement of the decimal point can 'float' relative to the significant digits of the number. For example, a fixed-point representation with a uniform decimal point placement convention can represent the numbers 123.45, 1234.56, 12345.67, etc., whereas a floating-point representation could represent 1.234567, 123456.7, 0.00001234567, 1234567000000000, etc. As such, the floating- point can support a much wider range of values than a fixed point,with the ability to represent very small numbers and very large numbers.

Floating point

Floating-point describes a method of representing real numbers in a way that can support a wide range of values. Numbers are, in general, represented approximately to a fixed number of significant digits and scaled using an exponent. The base for the scaling is normally 2, 10, or 16. The typical number that can be represented exactly is of the form:

$$\text{Significant digits} \times \text{base}^{\text{exponent}}$$

3.1 Floating-Point Representation

The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the ***mantissa***. The second part designates the position of the decimal (or binary) point and is called the ***exponent***. The fixed-point mantissa may be a fraction or an integer. For example, the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	**Exponent**
+0.6132789	+04

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in thefraction. This representation is equivalent to the scientific notation

$+0.6132789 \times 10^{+4}$.

Floating-point is always interpreted to represent a number in the following form:

$$\mathbf{m \times r^e}$$

Only the mantissa m and the exponent e are physically represented in the register (including their signs). The radix r and the radix-point position of themantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.

A floating-point binary number is represented in a similar manner exceptthat it uses base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	**Exponent**
01001110	000100

The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$m \times 2^e = +(.1001110)_2 \times 2^{+4}$$

A floating-point number is said to be ***normalized*** if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not. Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0s. The number can be normalized by shifting it three positions to the left and discarding the leading 0s to obtain 11010000. The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0s in the mantissa and exponent.

Arithmetic operations with floating-point numbers are more complicated thanarithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware. However, floating-point representation is a must for scientific computations because of the scaling problems involved with fixed-point computations. Many computers and all electronic calculators have the built-in capability of performing floating-point arithmetic operations. Computers that do not have hardware for floating- point computations have a set of subroutines to help the user programscientific problems with floating-point numbers.

4. SIGNED NUMBERS

In computing, **signed number representations** are required to encodenegative numbers in binary number systems. In mathematics, '+' sign is used for a positive number and '−'sign for negative numbers in any base. However, in computer hardware, numbers are represented in bit vectors only without extra symbols. The best-known methods of extending the binary numeral system to represent signed numbers are:

(i) Sign-and-magnitude method

(ii) One's complement method

(iii) Two's complement method

Sign-and-magnitude method

In this approach, one bit is reserved for representing sign and the remaining bits of the given number represent magnitude. We represent the signed numbers by using one **sign bit** to represent the sign: set that bit (often the most significant bit) to 0 for a positive number, and set it to 1 for a negative number. The remaining bits in the number indicate the magnitude. Hence in a byte with only 7 bits (apart from the sign bit), the magnitude can rangefrom 0000000 (0) to 1111111 (127). Thus you can represent numbers from -127_{10} to $+127_{10}$ once you add the sign bit (the eighth bit). A consequenceof this representation is that there are two ways to represent zero, 00000000 (0) and 10000000 (−0). Decimal −43 encoded in an eight-bit byte this way is 10101011.

One's complement method

Alternatively, a system known as one's complement can be used to represent negative numbers. The ones' complement form of a negativebinary number is the bitwise NOT applied to it — the "complement" of its positive counterpart. Like sign-and-magnitude representation, one'scomplement has two representations of 0: 00000000 (+0) and 11111111 (−0).

As an example, the ones' complement form of 00101011 (43) becomes 11010100 (−43). The range of signed numbers using one's complement is represented by $-(2^{N-1}-1)$ to $(2^{N-1}-1)$

and ±0. A conventional eight-bit byte is -127_{10} to $+127_{10}$ with zero being either 00000000 (+0) or 11111111 (−0).

Two's complement method

The problems of multiple representations of 0 and the need for the end-around carry are circumvented by a system called **two's complement**. In two's complement, negative numbers are represented by the bit pattern which is one greater (in an unsigned sense) than the ones' complement of the positive value. In two's-complement, there is only one zero (00000000). Negating a number (whether negative or positive) is done by inverting all the bits and then adding 1 to that result.

4.1 Binary Arithmetic

Let us have a study on how basic arithmetic can be performed on binary numbers.

Binary Addition

There are four basic rules with Binary Addition

$0_{(2)} + 0_{(2)} = 0_{(2)}$

$0_{(2)} + 1_{(2)} = 0_{(2)}$ Addition of two single bits result into single bit

$1_{(2)} + 0_{(2)} = 1_{(2)}$

$1_{(2)} + 1_{(2)} = 10_{(2)}$ Addition of two 1's resulted in Two bits

Example: perform the binary addition on the following

1 1		1 1 1 1	1	1	1
$011_{(2)}$	$3_{(10)}$	$1101_{(2)}$	$13_{(10)}$	$11100_{(2)}$	$28_{(10)}$
$+\ 011_{(2)}$	$+\ 3_{(10)}$	$+\ 0111_{(2)}$	$+\ 07_{(10)}$	$+\ 10011_{(2)}$	$+\ 19_{(10)}$
$110_{(2)}$	$6_{(10)}$	$10100_{(2)}$	$20_{(10)}$	$101111_{(2)}$	$47_{(10)}$

Binary Subtraction

There are four basic rules associated while carrying Binary subtraction

$0_{(2)} - 0_{(2)} = 0_{(2)}$

$1_{(2)} - 1_{(2)} = 0_{(2)}$

$1_{(2)} - 0_{(2)} = 1_{(2)}$

$0_{(2)} - 1_{(2)}$ = invalid therefore obtain a borrow 1 from MSB andperform binary subtraction

$10_{(2)} - 1_{(2)} = 1_{(2)}$

Note: In the last rule it is not possible to subtract 1 from 0 therefore a 1 is borrowed from the immediate next MSB to have a value of 10 and then the subtraction of 1 from 10 is carried out

Example: perform the binary subtraction on the following

		1 1		1 1	
$011_{(2)}$	$3_{(10)}$	$1101_{(2)}$	$13_{(10)}$	$11100_{(2)}$	$28_{(10)}$
$-11_{(2)}$	$-3_{(10)}$	$-0111_{(2)}$	$-07_{(10)}$	$-10011_{(2)}$	$-19_{(10)}$
$000_{(2)}$	$0_{(10)}$	$0110_{(2)}$	$06_{(10)}$	$01001_{(2)}$	$09_{(10)}$

Binary Multiplication

There are four basic rules associated while carrying Binary multiplication

$0_{(2)} \times 0_{(2)} = 0_{(2)}$

$0_{(2)} \times 1_{(2)} = 0_{(2)}$

$1_{(2)} \times 0_{(2)} = 0_{(2)}$

$1_{(2)} \times 1_{(2)} = 1_{(2)}$

Note: While carrying binary multiplication with binary numbers the rule of shift and add is made used similar to the decimal multiplication.

i.e. multiplication is first carried out with the LSB of the multiplicand on the multiplier bit-by-bit basis. While multiplying with the MSB bits, first the partial sum is obtained. Then the result is shifted to the left by one bit and added to the earlier result obtained.

Example: perform the binary multiplication on the following

$011_{(2)}$	3	$1101_{(2)}$	$13_{(10)}$
X $1_{(2)}$	**X**$1_{(10)}$	**X** $11_{(2)}$	**X** $03_{(10)}$
$011_{(2)}$	$3_{(10)}$	$1101_{(2)}$	
		$1101_{(2)}$	
		$100111_{(2)}$	$39_{(10)}$

Binary Division

The binary division is similar to the decimal procedure

Example: perform the binary division

```
          101(2)                     10.1(2)
  ___________________       ____________________
  11   |111                 110   |1111.0
       | 11                       |110
       |-----                     |-----
       |001                       |0011
       |000                       | 000
       |-----                     |-----
       |   11                     |  110
       |   11                     |  110
       |-----                     |-----
       |   00                     |  000
```

4.2 1’s and 2’s Complements Arithmetic

1’s complement of a given binary number can be obtained by replacing all0s by 1s and 1s by 0s. Let us describe the 1’s complement with the following examples

Examples: 1’s complement of the binary numbers

Binary Number	1’s Complement
1101110	0010001
111010	000101
110	001
11011011	00100100

Binary subtraction using 1’s complementary Method:

Binary number subtraction can be carried out using the method discussed in the binary subtraction method. The complementary method also can be used. While performing the subtraction the 1’s complement of the subtrahend is obtained first and then added to the minuend. Therefore 1’s complement method is useful in the sense subtraction can be carried with adder circuitsof ALU (Arithmetic logic unit) of a processor.

Two different approaches were discussed here depending on, whether the subtrahend is smaller or larger compared with minuend.

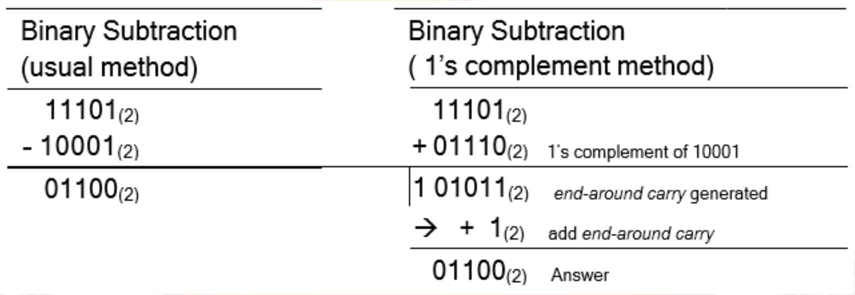
Case i) Subtrahend is smaller compared to minuend

Step 1: Determine the 1’s complement of the subtrahend

Step 2: 1’s complement is added to the minuend, which results in a carrygeneration known as *end-around carry*.

Step 3: From the answer remove the *end-around carry* thus generated andadd to the answer.

Example: Perform the subtraction using 1’s complement method



Binary Subtraction (usual method)	Binary Subtraction (1’s complement method)
$11101_{(2)}$	$11101_{(2)}$
$- 10001_{(2)}$	$+ 01110_{(2)}$ 1’s complement of 10001
$01100_{(2)}$	$1\ 01011_{(2)}$ *end-around carry* generated
	→ $+ 1_{(2)}$ add *end-around carry*
	$01100_{(2)}$ Answer

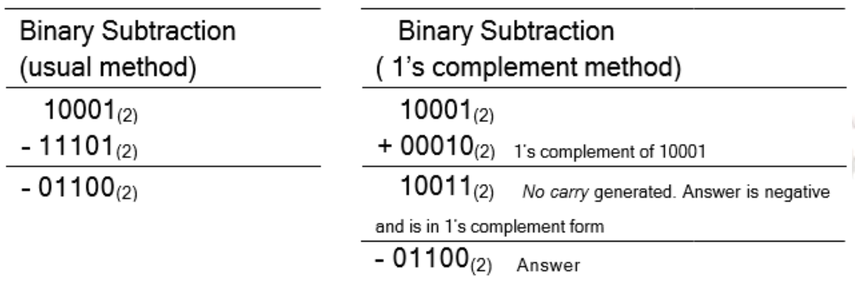
Case ii) Subtrahend is larger compared to the minuend

Step 1: Determine the 1’s complement of the subtrahend

Step 2: Add the 1’s complement to the minuend and no carry is generated.Step 3: Answer is negative signed and is in 1’s complement form.

Therefore obtain the 1’s complement of the answer and indicate with a negative sign.

Example: Perform the subtraction using 1’s complement method

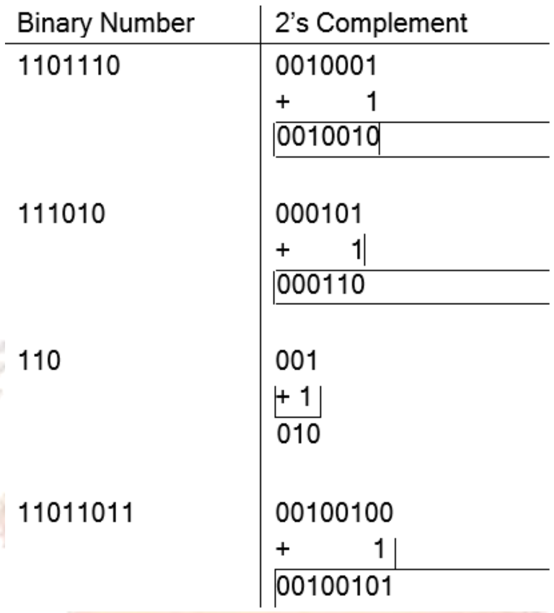


Binary Subtraction (usual method)	Binary Subtraction (1’s complement method)
$10001_{(2)}$	$10001_{(2)}$
$- 11101_{(2)}$	$+ 00010_{(2)}$ 1’s complement of 10001
$- 01100_{(2)}$	$10011_{(2)}$ *No carry* generated. Answer is negative and is in 1’s complement form
	$- 01100_{(2)}$ Answer

Binary subtraction using 2’s complementary Method:

2’s complement of a given binary number can be obtained by first obtaining 1’s complement and then add 1 to it. Let us obtain the 2’s complement of the following.

Examples: 2’s complement of the binary numbers



Binary Number	2’s Complement
1101110	0010001
	+ 1
	0010010
111010	000101
	+ 1
	000110
110	001
	+ 1
	010
11011011	00100100
	+ 1
	00100101

Binary number subtraction can be carried out using 2’s complement method also. While performing the subtraction the 2’s complement of the subtrahendis obtained first and then added to the minuend.

Two different approaches were discussed here depending on, whether the subtrahend is smaller or larger compared with minuend.

Case i) Subtrahend is smaller compared to the minuend

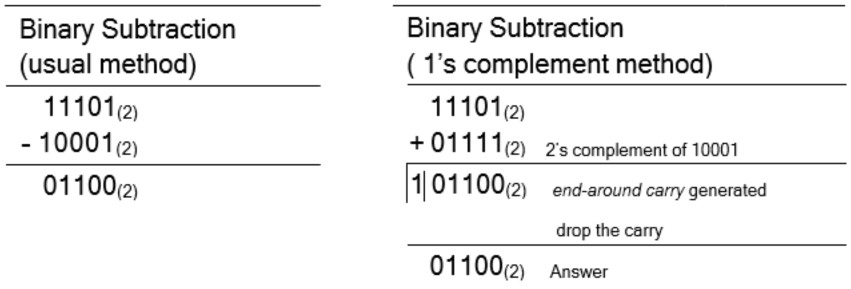
Step 1: Determine the 2’s complement of the subtrahend

Step 2: 2’s complement is added to the minuend generating an *end-around carry*.

Step 3: From the answer remove the *end-around carry* and drop it.

Example: Perform the subtraction using 2’s complement method

Binary Subtraction(usual method) Binary Subtraction (1’s complement method)



Binary Subtraction (usual method)
$11101_{(2)}$
$-\ 10001_{(2)}$
$01100_{(2)}$

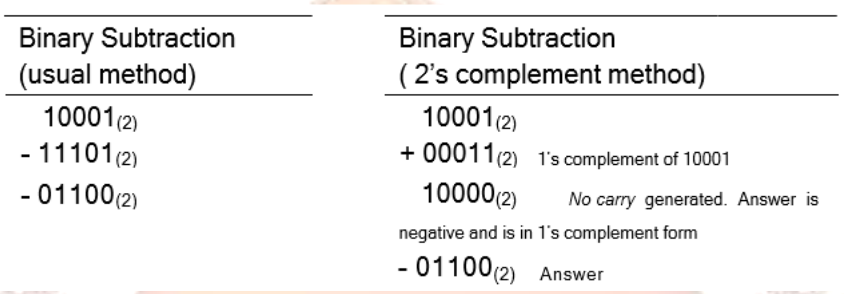
Binary Subtraction (1’s complement method)	
$11101_{(2)}$	
$+\ 01111_{(2)}$	2’s complement of 10001
$1\ 01100_{(2)}$	*end-around carry* generated
	drop the carry
$01100_{(2)}$	Answer

Case ii) Subtrahend is larger compared to minuend

Step 1: Determine the 2’s complement of the subtrahend

Step 2: Add the 2’s complement to the minuend and no carry is generated.

Step 3: Answer is negative signed and is in 2’s complement form. Thereforeobtain the 2’s complement of the answer and indicate a negative sign.



Binary Subtraction (usual method)	Binary Subtraction (2’s complement method)
$10001_{(2)}$	$10001_{(2)}$
$- 11101_{(2)}$	$+ 00011_{(2)}$ 1’s complement of 10001
$- 01100_{(2)}$	$10000_{(2)}$ *No carry* generated. Answer is negative and is in 1’s complement form
	$- 01100_{(2)}$ Answer

Self-Assessment Questions – 2

3. Floating point describes a method of representing _______________numbers in a way that can support a wide range of values.
4. The advantage of________________support a much wider range of values.
5. Representation is that it canIn computing, signed number representations are required to encode _______numbers in binary number systems.
6. Ones' complement can be used to represent negativenumbers.(True/False)

5. BOOTH’S ALGORITHM

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithmwas invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm is of interest in the study of computer architecture. Booth's algorithm examines adjacent pairsof bits of the N-bit multiplier Y in signed two's complement representation, including an implicit bit below the least significant bit, $y_{-1} = 0$. For each bit y_i, for i running from 0 to N-1, the bits y_i and y_{i-1} are considered. Where these two bits are equal, the product accumulator P remains unchanged. Where$y_i = 0$ and $y_{i-1} = 1$, the multiplicand times 2^i is added to P; and where $y_i =$

1 and $y_{i-1} = 0$, the multiplicand times 2^i is subtracted from P. The final value of P is the signed product.

Booth's Algorithm - a Flowchart

Figure 2.1 shows the Booth's Algorithm - Flowchart.

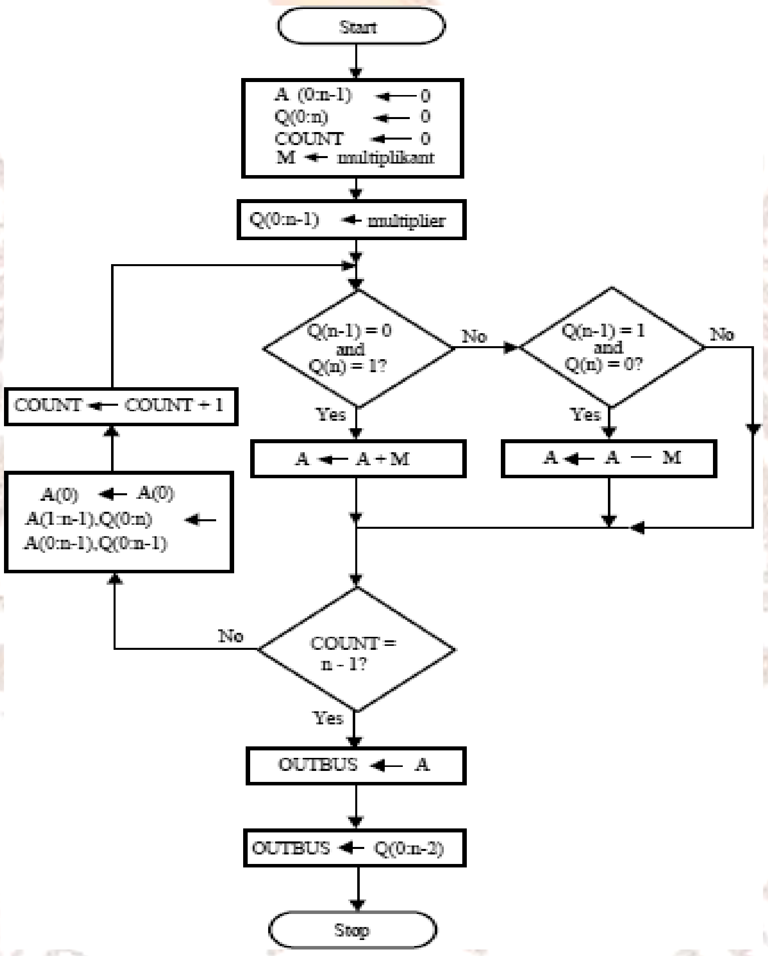


Fig 2.1: Booth's Algorithm - a Flowchart

The algorithm is often described as converting strings of 1's in the multiplier to a high-order +1 and a low-order −1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpreted as a negative of the appropriate value.

A typical implementation

Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values A and S to a product P, then performing a rightward arithmetic shift on P. Let m and r be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in m and r.

1. Determine the values of A and S, and the initial value of P. All of these numbers should have a length equal to (x + y + 1).
    1. A: Fill the most significant (leftmost) bits with the value of m. Fill the remaining (y + 1) bits with zeros.
    2. S: Fill the most significant bits with the value of (−m) in two's complement notation. Fill the remaining (y + 1) bits with zeros.
    3. P: Fill the most significant x bits with zeros. To the right of this, append the value of r. Fill the least significant (rightmost) bit with a zero.
2. Determine the two least significant (rightmost) bits of P.
    1. If they are 01, find the value of P + A. Ignore any overflow.
    2. If they are 10, find the value of P + S. Ignore any overflow.
    3. If they are 00, do nothing. Use P directly in the next step.
    4. If they are 11, do nothing. Use P directly in the next step.
3. Arithmetically shift the value obtained in the 2nd step by a single placeto the right. Let P now equal this new value.
4. Repeat steps 2 and 3 until they have been done y times.
5. Drop the least significant (rightmost) bit from P. This is the product of mand r.

Example

Find 3 × (−4), with m = 3 and r = −4, and x = 4 and y = 4:

- m = 0011, -m = 1101, r = 1100
- A = 0011 0000 0
- S = 1101 0000 0
- P = 0000 1100 0
- Perform the loop four times :

1. P = 0000 1100 0. The last two bits are 00.
    - P = 0000 0110 0. Arithmetic right shift.
2. P = 0000 0110 0. The last two bits are 00.
    - P = 0000 0011 0. Arithmetic right shift.
3. P = 0000 0011 0. The last two bits are 10.

- P = 1101 0011 0. P = P + S.
- P = 1110 1001 1. Arithmetic right shift.

4. P = 1110 1001 1. The last two bits are 11.
   - P = 1111 0100 1. Arithmetic right shift.
   - The product is 1111 0100, which is −12.

The above-mentioned technique is inadequate when the multiplicand is the largest negative number that can be represented (e.g. if the multiplicand has 4 bits then this value is −8). One possible correction to this problem is to add one more bit to the left of A, S, and P.

Self-Assessment Questions – 3

7. Booths Algorithm invented by___________.
8. Booth used_____________that were faster at shifting than addingand created the algorithm to increase their speed.

6. HARDWARE IMPLEMENTATION

The performance of software systems is dramatically affected by how well software designers understand the basic hardware technologies at work in a system. Similarly, hardware designers must understand the far-reaching effects their design decisions have on software applications. For readers in either category, this classic introduction to the field provides a deep look into the computer. It demonstrates the relationship between the software and hardware and focuses on the foundational concepts that are the basis for current computer design.

Division

The binary division is again similar to its decimal counterpart:

For example, the divisor is 101_2, or 5 decimal, while the dividend is 11011_2, or 27 decimal. The procedure is the same as that of decimal long division; here, the divisor 101_2 goes into the first three digits 110_2 of the dividend one time, so a "1" is written on the top line. This result is multiplied by the divisor,and subtracted from the first three digits of the dividend; the next digit (a "1") is included to obtain a new three-digit sequence:

For Example

The procedure is then repeated with the new sequence, continuing until the digits in the dividend have been exhausted:

```
1 0 1
     ___________
101 ) 1 1 0 1 1
    – 1 0 1
     --------
       0 1 1
     – 0 0 0
       --------
        1 1 1
      – 1 0 1
          1 0
```

Thus, the quotient of 11011_2 divided by 101_2 is 101_2, as shown on the top line, while the remainder, shown on the bottom line, is 10_2. In decimal, 27 divided by 5 is 5, with a remainder of 2.

Restoring and Non Restoring algorithms

Restoring Division Algorithm

Put *x* in register A, *d* in register B, *0* in register P, and Perform *n* divide steps (*n* is the quotient word length)

- Each step consists of
  - (i) Shift the register pair (P, A) one bit left
  - (ii) Subtract the contents of B from P; put the result back in P
  - (iii) If the result is –ve, set the low-order bit of A to 0 otherwise to 0
  - (iv) If the result is -ve, restore the old value of P by adding theContents of B back in P.

Figure 2.2 shows the concept of Restoring the Division

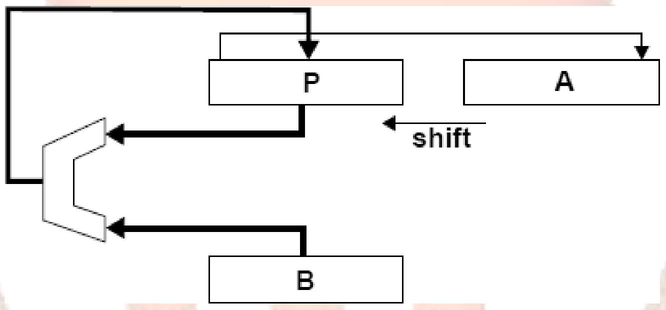
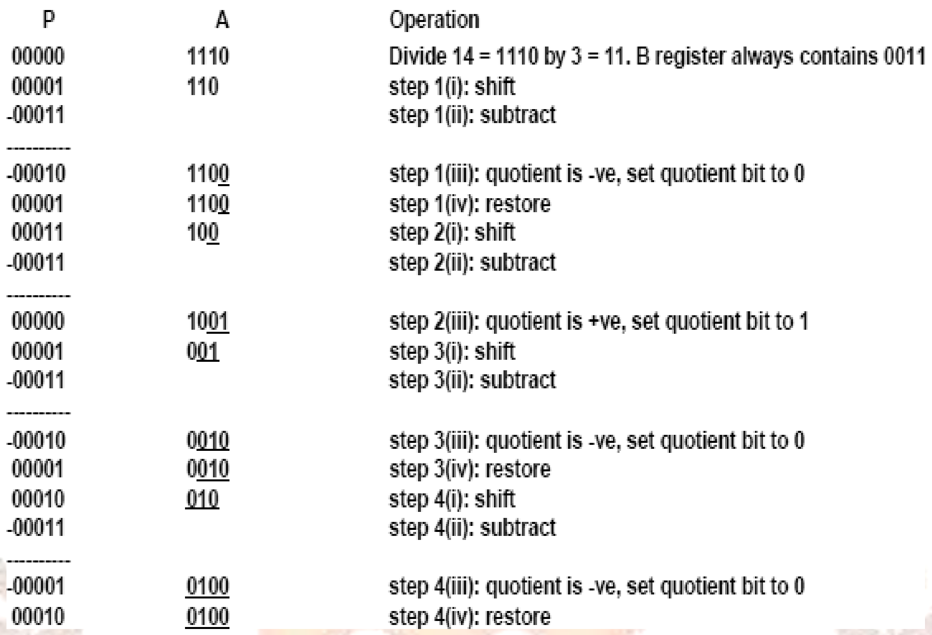


Fig 2.2: Restoring Division

After n cycles, A will contain the quotient, P will contain the remainderThe table 2.1 shows the Restoring Division Example.

Table 2.1 Restoring Division Example



P	A	Operation
00000	1110	Divide 14 = 1110 by 3 = 11. B register always contains 0011
00001	110	step 1(i): shift
-00011		step 1(ii): subtract

-00010	1100	step 1(iii): quotient is -ve, set quotient bit to 0
00001	1100	step 1(iv): restore
00011	100	step 2(i): shift
-00011		step 2(ii): subtract

00000	1001	step 2(iii): quotient is +ve, set quotient bit to 1
00001	001	step 3(i): shift
-00011		step 3(ii): subtract

-00010	0010	step 3(iii): quotient is -ve, set quotient bit to 0
00001	0010	step 3(iv): restore
00010	010	step 4(i): shift
-00011		step 4(ii): subtract

-00001	0100	step 4(iii): quotient is -ve, set quotient bit to 0
00010	0100	step 4(iv): restore

- The quotient is 0100 and the remainder is 00010
- The name restoring because if subtraction by b yields a negative result,the P register is restored by adding b back

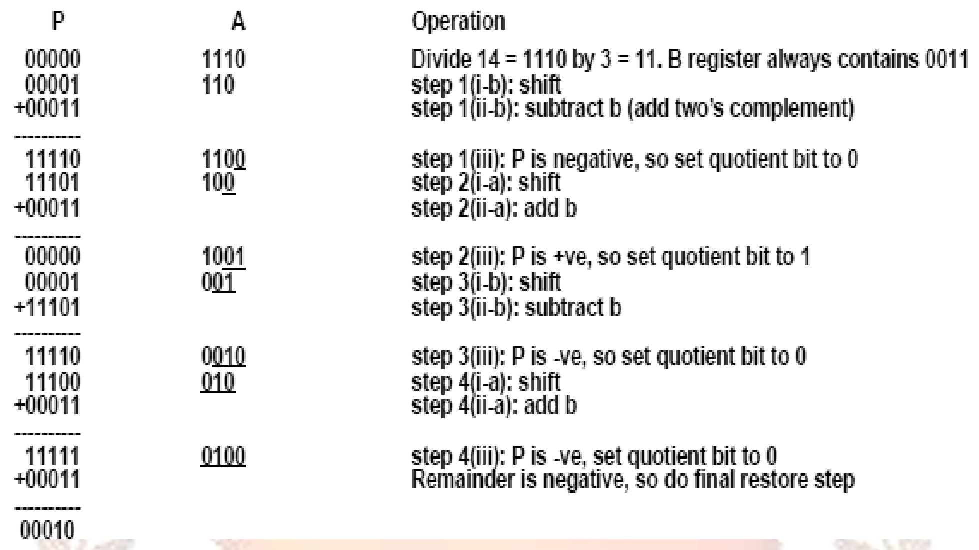
Non-Restoring Division Algorithm

A variant that skips the restoring step and instead works with negativeresiduals

- If P is negative
- (i-a) Shift the register pair (P,A) one bit left
- (ii-a) Add the contents of register B to P
- If P is positive
- (i-b) Shift the register pair (P,A) one bit left
- (ii-b) Subtract the contents of register B from P
- If P is negative, set the low-order bit of A to 0, otherwise set it to 1After n cycles
- The quotient is in A
- If P is positive, it is the remainder, otherwise, it has to be restored (add Bto it) to get the remainder

The table 2.2 shows the Non-Restoring Division Example

Table 2.2: Non-Restoring Division Example



P	A	Operation
00000	1110	Divide 14 = 1110 by 3 = 11. B register always contains 0011
00001	110	step 1(i-b): shift
+00011		step 1(ii-b): subtract b (add two's complement)

11110	1100	step 1(iii): P is negative, so set quotient bit to 0
11101	100	step 2(i-a): shift
+00011		step 2(ii-a): add b

00000	1001	step 2(iii): P is +ve, so set quotient bit to 1
00001	001	step 3(i-b): shift
+11101		step 3(ii-b): subtract b

11110	0010	step 3(iii): P is -ve, so set quotient bit to 0
11100	010	step 4(i-a): shift
+00011		step 4(ii-a): add b

11111	0100	step 4(iii): P is -ve, set quotient bit to 0
+00011		Remainder is negative, so do final restore step

00010		

- The quotient is 0100 and the remainder is 00010
- restoring division seems to be more complicated since it involves extraaddition in step (iv)
- This is not true since the sign resulting from the subtraction is Tested atadder o/p and only if the sum is +ve, it is loaded back to the p register.

Self-Assessment Questions – 4

9. _____________ algorithms produce one digit of the final quotient periteration.

10. Slow division algorithms produce_____________of the final quotient periteration

7. IEEE STANDARDS

IEEE 754-1985 was an industry standard for representing floating-point numbers in computers, officially adopted in 1985 and superseded in 2008 byIEEE 754-2008. During its 23 years, it was the most widely used format for floating-point computation. It was implemented in software, in the form of floating-point libraries, and in hardware, in the instructions of many CPUs and FPUs. The first integrated circuit to implement the draft of what was to become IEEE 754-1985 was the Intel 8087. IEEE 754-1985 represents numbers in binary, providing definitions for four levels of precision, of which the two most commonly used and are shown in table 2.3.

Table 2.3: IEEE standards levels of precision

Level	Width	range	precision*
Single precision	32 bits	$\pm 1.18\times 10^{-38}$to $\pm 3.4\times 10^{38}$	approx.7decimal digits
double precision	64 bits	$\pm 2.23\times 10^{-308}$ $\pm 1.80\times 10^{308}$	approx.15decimaldigits

Precision: The number of decimal digits precision is calculated via number_of_mantissa_bits * Log_{10} (2). The standard also defines representations for positive and negative infinity, a "negative zero", fiveexceptions to handle invalid results like division by zero, special values called NaNs for representing those exceptions, dermal numbers to represent numbers smaller than shown above, and four rounding modes.

Self-Assessment Questions – 5

11. In IEEE standards Single precision Width is___________.
12. The first integrated circuit to implement the draft of what was to become IEEE 754-1985 was_______________.

8. FLOATING-POINT ARITHMETIC

IEEE Standard 754

- Established in 1985 as a uniform standard for floating-point arithmetic before that, many idiosyncratic formats supported by all major CPUs driven by numerical concerns
- Nice standards for rounding, overflow, underflow
- Hard to make go fast

Numerical analysts predominated over hardware types in defining standard Floating-point arithmetic is a way to represent and handle a large range of real numbers in a binary form: The C-64's built-in BASIC interpreter containsa set of subroutines that perform various tasks on numbers in floating-point format, allowing BASIC to use real numbers. These routines may also be called from the user's own machine code programs, to handle real numbers in the range $\pm 2.93873588 \cdot 10^{-38}$ to $\pm 1.70141183 \cdot 10^{38}$. A real number T in the floating-point format consists of a mantissa m and an integer exponent E, which are "selected" so that

$$T = m \cdot 2^E$$

The mantissa is always a number in the range from 1 to 2, so that $1 \le m < 2$,and it's stored as a fixed-decimal binary real; a number that begins with a one and the decimal point, followed by several binary decimals (31 of them, in the case of the 64's BASIC routines). The exponent is an integer with some special provisions for handling negative exponents (i.e. floating-point real numbers less than 1): The 64 stores the exponent as the numberE + 128, so that an exponent of 2 is stored as 130 (128 + 2), and an exponent of −2 as 126 (128 — 2).

Self-Assessment Questions – 6

13. Floating-point arithmetic is a way to represent and handle a large rangeof real numbers in a ____________ .
14. The mantissa is always a number in the range from_______________.

9. THE ACCUMULATOR

In a computer's central processing unit (CPU), an accumulator is a registerin which intermediate arithmetic and logic results are stored. Without aregister like an accumulator, it would be necessary to write the result ofeach calculation (addition, multiplication, shift, etc.) to the main memory, perhaps only to be read right back again for use in the next operation. Access to main memory is slower than access to a register like an accumulator because the technology used for the large main memory is slower (but cheaper) than that used for a register.

The canonical example for accumulator use is summing a list of numbers. The accumulator is initially set to zero, then each number in turn is read and added to the value in the accumulator. Only when all numbers have been added is the result held in the accumulator written to main memory or to another, non-accumulator, CPU register.

Modern CPUs are typically 2-operand or 3-operand machines — the additional operands specify which one of many general-purpose registers (also called "general purpose accumulators") are used as the source and destination for calculations. These CPUs are not considered "accumulator machines".

The characteristic which distinguishes one register as being the accumulatorof a computer architecture is that the accumulator (if the architecture were tohave one) would be used as an implicit operand for arithmetic instructions. For instance, a CPU might have an instruction like:

ADD memaddress

This instruction would add the value read from the memory location atmemaddress to the value from the accumulator, placing the result in the accumulator. The accumulator is not identified in the instruction by a registernumber; it is implicit in the instruction and no other register can be specified in the instruction. Some architectures use a particular register as an accumulator in some instructions, but other instructions use register numbers for explicit operand specification.

Self-Assessment Questions – 7

15. ____________ is a register in which intermediate arithmetic andlogic results are stored.
16. Modern CPUs are typically__________machines.

10. SHIFTS, CARRY AND OVERFLOW

The shift operation, with its various modes and applications, is discussed here. The ALU may have one or more shift registers in order to implement the different types of shifts. This section shows two designs for general shift registers, one using D flip-flops and the other using JK flip-flops. A shift register may be unidirectional or bidirectional. It may have serial or parallel inputs (or both). In a shift register with serial input, the bits to be shifted are input one by one. Each time a bit is shifted to one end.

Figure 2.5 shows a serial-serial (i.e., serial input and serial output) shift register based on D flip-flops. This register shifts because the output Q of aD flip-flop is set to the input D on a clock pulse (in the diagram, they are set on the trailing edge of the clock pulse). The register has serial output, but the parallel output can be obtained by connecting lines to the various Q outputs. The figure also shows how parallel input can be added to this shift registerby using flip-flops with a “present” input.

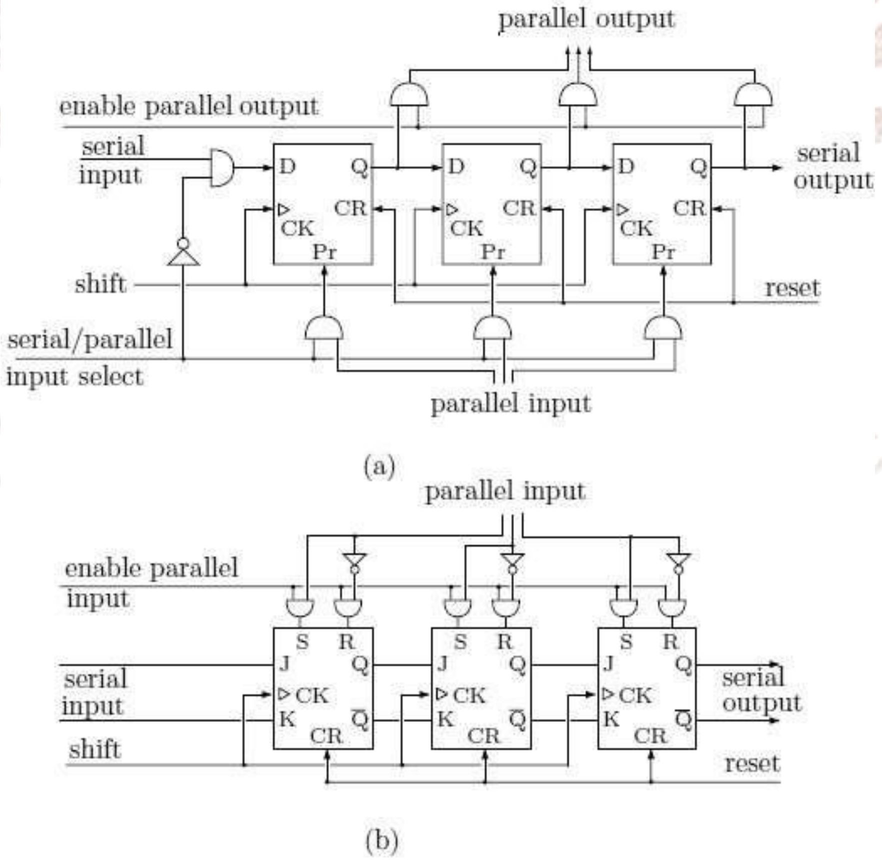


Fig 2.3: Shift registers output

The figure illustrates a similar shift register based on JK flip-flops connected in amaster-slave configuration. These flip-flops have synchronous JK inputs andasynchronous SR inputs. The former performs the Shifts and the latter is used for the parallel input. These diagrams also suggest that a typical shift register can easily be extended to perform circular shifts by feeding the serial output into the serial input of the register.

Carry and Overflow

These two concepts are associated with arithmetic operations. Certain operations may result in a carry, in an overflow, in none, or in both. There is much confusion and misunderstanding among computer users concerning carry and overflow, and this section attempts to clear up this confusion by defining and illustrating these concepts.

Definitions:

Given two N-bit numbers used in an addition or subtraction, we would like,of course, to end up with a result that has the same size, and fits in the same registers or memory words where the original operands came from. We consequently say that the operation results in a carry if the sum (or difference) has N + 1 bit. Similarly, the operation results in an overflow if the result is too large to fit in a register or a computer word.

Carry Algorithm

- C (Carry)=1 carry generated
- For n-bit arithmetic operations, the carry contains an 'extra' most-significant bit.
- Addition (carry)
- So, for a byte operation, the carry flag will contain the ninth bit

Example 1:

0xFF + 0x1 = 0x100.

But, since the destination is a 8-bit byte, the result will be 0x0Carry will be 1

Example 2:

0x03 + 0x04 = 0x07

Result is 7

Carry will be 0

Implications for relational operators

Subtraction (borrow) A-B:

- If B > A, then C=1.
- If B <= A, then C=0

Subtract in opposite order to determine B < A or B >=A.

- Rotate right instruction: rotates through carry
- The previous value of carry becomes the new value for the mostsignificant bit
- The previous value of the least significant bit becomes the new valuefor carry
- Useful for shifting multi-word integers to the right.

Overflow Algorithm

Indicates if the result of a signed 2's complement addition or subtraction isout-of-range,

How detected:

Sign bit is "wrong"

Addition

Sum of two non-negative values should be non-negative

Sum of two negative values should be negative

Subtraction

A - B:

if A is negative and B is positive, result should be negative

if A is positive and B is negative, result should be positive

Subtraction of same sign or addition of different signs cannot overflow

- If overflow is set, the sign bit is inverted (work this out)

For example

Take-away message: hear “V” indicates overflow

- ➢ If V == N, then the result should have been non-negative
  - N=V=0: If V=0, then N (0) is correct
  - N=V=1: If V=0, then N (1) is wrong
- ➢ if V != N, then the result should have been negative
  - V=0, N=1: If V=0, then N (1) is correct

- V=1, N=0: If V=0, then N (0) is incorrect

Self-Assessment Questions – 8

17. Two designs for general shift registers, one using__________and the other using ___________.
18. A shift register may be___________.

11. SUMMARY

In this unit, you learned the basic arithmetic concepts, signed numbers, and complements, and other related concepts like Booth's Algorithm, Floating-point arithmetic, etc. A binary digit is called a *bit*. Addition and subtraction are two familiar arithmetic operations of decimal numbers. Floating-point describes a method of representing real numbers in a way that can supporta wide range of values. In computing, signed number representations are required to encode negative numbers in binary number systems. Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. Floating-point arithmetic is a way to represent and handle a large range of real numbers in a binary form. V (overflow): Indicates if the result of a signed 2's complement addition or subtraction is out-of-range

12. TERMINAL QUESTIONS

1. How to find Integer Addition and Subtraction?
2. Discuss on fixed point and floating-point numbers
3. Explain how you can get 1's complement of a given binary number?Also explain 1's complement subtraction.
4. What is the use of Booth's Algorithm?
5. What we are learning by maintain IEEE standards using Floating-pointarithmetic, the accumulator and shifts, carry and overflow?

13. ANSWERS

Self-Assessment Questions:

1. Bit
2. positive
3. real
4. Floating point
5. Negative
6. True
7. Andrew Donald Booth
8. Calculators
9. Division
10. one digit
11. 32 bits
12. Intel 8087
13. binary form
14. 1 to 2
15. Accumulator
16. 2-operand or 3-operand
17. D flip-flops and JK flip-flops
18. unidirectional or bidirectional

Terminal Questions:

1. Addition is the basic operation of arithmetic. In its simplest form, additioncombines two numbers. For more details Refer section 2 for Integer Addition and Subtraction.
2. The term 'fixed point' refers to the corresponding manner in which numbers are represented, with a fixed number of digits after, and sometimes before, the decimal point. Refer section 3.
3. 1's complement of a given binary number can be obtained by replacing all 0s by 1s and 1s by 0s. Refer sub-section 4.2.
4. Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. For more details. Refer section 6.
5. IEEE 754-1985 was an industry standard for representing floating-point numbers in computers. For more details Refer section 9 to 12 for learning IEEE standards.