

UNIT 2:

Data Representation and Computer Arithmetic

Number System

The number system or the numeral system is the system of naming or representing numbers. We know that a number is a mathematical value that helps to count or measure objects and it helps in performing various mathematical calculations. There are different types of number systems in Maths like decimal number system, binary number system, octal number system, and hexadecimal number system. In this article, we are going to learn what is a number system in Maths, different types, and conversion procedures with many number system examples in detail. Also, check [mathematics for grade 12](#) here.

What is a Number?

A number is a mathematical value used for counting or measuring or labelling objects. Numbers are used to performing arithmetic calculations. Examples of numbers are natural numbers, whole numbers, rational and irrational numbers, etc. 0 is also a number that represents a null value.

A number has many other variations such as even and odd numbers, prime and composite numbers. Even and odd terms are used when a number is divisible by 2 or not, whereas prime and composite differentiate between the numbers that have only two factors and more than two factors, respectively.

In a number system, these numbers are used as digits. 0 and 1 are the most common digits in the number system, that are used to represent binary numbers. On the other hand, 0 to 9 digits are also used for other number systems. Let us learn here the types of number systems.

Types of Number Systems

There are various types of number systems in mathematics. The four most common number system types are:

1. Decimal number system (Base- 10)
2. Binary number system (Base- 2)
3. Octal number system (Base-8)
4. Hexadecimal number system (Base- 16)

Now, let us discuss the different types of number systems with examples.

Decimal Number System (Base 10 Number System)

The decimal number system has a base of 10 because it uses ten digits from 0 to 9. In the decimal number system, the positions successive to the left of the decimal point represent units, tens, hundreds, thousands and so on. This system is expressed in [decimal numbers](#). Every position shows a particular power of the base (10).

Example of Decimal Number System:

The decimal number 1457 consists of the digit 7 in the units position, 5 in the tens place, 4 in the hundreds position, and 1 in the thousands place whose value can be written as:

$$(1 \times 10^3) + (4 \times 10^2) + (5 \times 10^1) + (7 \times 10^0)$$

$$(1 \times 1000) + (4 \times 100) + (5 \times 10) + (7 \times 1)$$

$$1000 + 400 + 50 + 7$$

$$1457$$

Binary Number System (Base 2 Number System)

The base 2 number system is also known as the [Binary number system](#) wherein, only two binary digits exist, i.e., 0 and 1. Specifically, the usual base-2 is a radix of 2. The figures described under this system are known as binary numbers which are the combination of 0 and 1. For example, 110101 is a binary number.

We can convert any system into binary and vice versa.

Example

Write $(14)_{10}$ as a binary number.

Solution:

2	14	
2	7	0
2	3	1
	1	1

© Byjus.com

Base 2 Number System Example

$$\therefore (14)_{10} = 1110_2$$

Octal Number System (Base 8 Number System)

In the [octal number system](#), the base is 8 and it uses numbers from 0 to 7 to represent numbers. Octal numbers are commonly used in computer applications. Converting an octal number to decimal is the same as decimal conversion and is explained below using an example.

Example: Convert 215_8 into decimal.

Solution:

$$215_8 = 2 \times 8^2 + 1 \times 8^1 + 5 \times 8^0$$

$$= 2 \times 64 + 1 \times 8 + 5 \times 1$$

$$= 128 + 8 + 5$$

$$= 141_{10}$$

Hexadecimal Number System (Base 16 Number System)


In the hexadecimal system, numbers are written or represented with base 16. In the hexadecimal system, the numbers are first represented just like in the decimal system, i.e. from

0 to 9. Then, the numbers are represented using the alphabet from A to F. The below-given table shows the representation of numbers in the [hexadecimal number system](#).

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Number System Chart

In the number system chart, the base values and the digits of different number systems can be found. Below is the chart of the numeral system.



Number System	Base value	Set of digits	Example
Base 3	3	0, 1, 2	$(123)_3$
Base 4	4	0, 1, 2, 3	$(145)_4$
Base 5	5	0, 1, 2, 3, 4	$(425)_5$
Base 6	6	0, 1, 2, 3, 4, 5	$(225)_6$
Base 7	7	0, 1, 2, 3, 4, 5, 6	$(1205)_7$
Base 8	8	0, 1, 2, 3, 4, 5, 6, 7	$(105)_8$
Base 9	9	0, 1, 2, 3, 4, 5, 6, 7, 8	$(25)_9$
Base 10	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	$(1125)_{10}$

Number System Chart

Number System Conversion

Numbers can be represented in any of the number system categories like binary, decimal, hexadecimal, etc. Also, any number which is represented in any of the number system types can be easily converted to another. Check the detailed lesson on the [conversions of number](#)

[systems](#) to learn how to convert numbers in decimal to binary and vice versa, hexadecimal to binary and vice versa, and octal to binary and vice versa using various examples.

With the help of the different conversion procedures explained above, now let us discuss in brief about the conversion of one number system to the other number system by taking a random number.

Assume the number 349. Thus, the number 349 in different number systems is as follows:

The number 349 in the binary number system is 101011101

The number 349 in the decimal number system is 349.

The number 349 in the octal number system is 535.

The number 349 in the hexadecimal number system is 15D

Number System Solved Examples

Example 1:

Convert $(1056)_{16}$ to an octal number.

Solution:

Given, 1056_{16} is a hex number.

First we need to convert the given hexadecimal number into decimal number

$$(1056)_{16}$$

$$= 1 \times 16^3 + 0 \times 16^2 + 5 \times 16^1 + 6 \times 16^0$$

$$= 4096 + 0 + 80 + 6$$

$$= (4182)_{10}$$

Now we will convert this decimal number to the required octal number by repetitively dividing by 8.

8	4182	Remainder
8	522	6
8	65	2
8	8	1
8	1	0
	0	1

Therefore, taking the value of the remainder from bottom to top, we get;

$$(4182)_{10} = (10126)_8$$

Therefore,

$$(1056)_{16} = (10126)_8$$

Example 2:

Convert $(1001001100)_2$ to a decimal number.

Solution:

$$(1001001100)_2$$

$$= 1 \times 2^9 + 0 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= 512 + 64 + 8 + 4$$

$$= (588)_{10}$$

Example 3:

Convert 10101_2 into an octal number.

Solution:

Given,

10101_2 is the binary number

We can write the given binary number as,

010 101

Now as we know, in the octal number system,

$010 \rightarrow 2$

$101 \rightarrow 5$

Therefore, the required octal number is $(25)_8$

Example 4:

Convert hexadecimal 2C to decimal number.

Solution:

We need to convert $2C_{16}$ into binary numbers first.

$2C \rightarrow 00101100$

Now convert 00101100_2 into a decimal number.

$$101100 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= 32 + 8 + 4$$

$$= 44$$

ASCII Code

The ASCII stands for American Standard Code for Information Interchange. The ASCII code is an alphanumeric code used for data communication in digital computers. The ASCII is a 7-bit code capable of representing 2^7 or 128 number of different characters. The ASCII code is made up of a three-bit group, which is followed by a four-bit code.

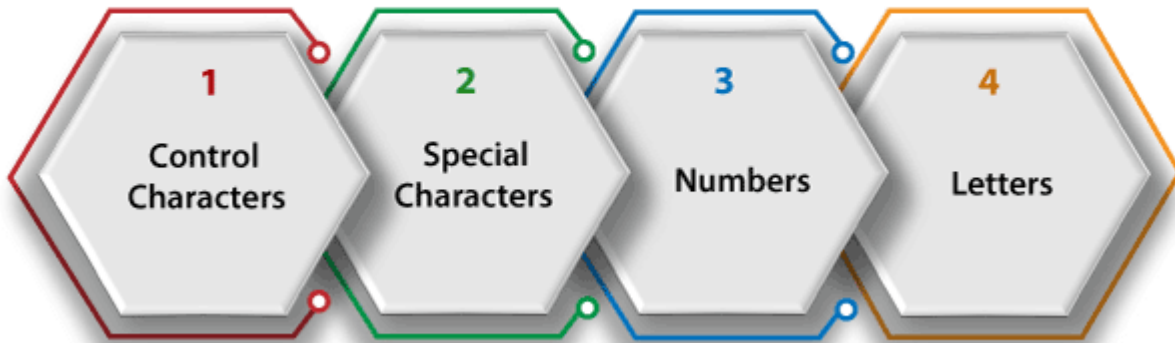
Representation of ASCII Code



- The ASCII Code is a 7 or 8-bit alphanumeric code.
- This code can represent 127 unique characters.
- The ASCII code starts from 00h to 7Fh. In this, the code from 00h to 1Fh is used for control characters, and the code from 20h to 7Fh is used for graphic symbols.
- The 8-bit code holds ASCII, which supports 256 symbols where math and graphic symbols are added.
- The range of the extended ASCII is 80h to FFh.

The ASCII characters are classified into the following groups:

ASCII Characters



Control Characters

The non-printable characters used for sending commands to the PC or printer are known as control characters. We can set tabs, and line breaks functionality by this code. The control characters are based on telex technology. Nowadays, it's not so much popular in use. The character from 0 to 31 and 127 comes under control characters.

Special Characters

All printable characters that are neither numbers nor letters come under the special characters. These characters contain technical, punctuation, and mathematical characters with space also. The character from 32 to 47, 58 to 64, 91 to 96, and 123 to 126 comes under this category.

Numbers Characters

This category of ASCII code contains ten Arabic numerals from 0 to 9.

Letters Characters

In this category, two groups of letters are contained, i.e., the group of uppercase letters and the group of lowercase letters. The range from 65 to 90 and 97 to 122 comes under this category.

Radix and Diminished Radix complement

The mostly used complements are 1's, 2's, 9's, and 10's complement. Apart from these complements, there are many more complements from which mostly peoples are not familiar. For finding the subtraction of the number base system, the complements are used. If r is the base of the number system, then there are two types of complements that are possible, i.e., r 's and $(r-1)$'s. We can find the r 's complement, and $(r-1)$'s complement of the number, here r is the radix. The r 's complement is also known as **Radix complement** $(r-1)$'s complement, is known as **Diminished Radix complement**.

If the base of the number is 2, then we can find 1's and 2's complement of the number. Similarly, if the number is the octal number, then we can find 7's and 8's complement of the number.

There is the following formula for finding the r 's and $(r-1)$'s complement:

$$\begin{aligned} r' \text{ s complement} &= (r^n)_{10} - N \\ (r-1)' \text{ s complement} &= \{(r^n)_{10} - 1\} - N \end{aligned}$$

In the above formulas,

- The n is the number of digits in the number.
- The N is the given number.

- The r is the radix or base of the number.

Advantages of r's complement

These are the following advantages of using r's complement:

- In r's complement, we can further use existing addition circuitry means there is no special circuitry.
- There is no need to determine whether the minuend and subtrahend are larger or not because the result has the right sign automatically.
- The negative zeros are eliminated by r's complement.

Let's take some examples to understand how we can calculate the r's and (r-1)'s complement of binary, decimal, octal, and hexadecimal numbers.

Example 1: $(1011000)_2$

This number has a base of 2, which means it is a binary number. So, for the binary numbers, the value of r is 2, and r-1 is 2-1=1. So, we can calculate the 1's and 2's complement of the number.

1's complement of the number 1011000 is calculated as:

$$\begin{aligned}
 &= \{(2^7)_{10} - 1\} - (1011000)_2 \\
 &= \{(128)_{10} - 1\} - (1011000)_2 \\
 &= \{(127)_{10}\} - (1011000)_2 \\
 &= 1111111_2 - 1011000_2 \\
 &= 0100111
 \end{aligned}$$

2's complement of the number 1011000 is calculated as:

$$\begin{aligned}
 &= (2^7)_{10} - (1011000)_2 \\
 &= (128)_{10} - (1011000)_2 \\
 &= 10000000_2 - 1011000_2 \\
 &= 0101000_2
 \end{aligned}$$

Example 2: $(155)_{10}$

This number has a base of 10, which means it is a decimal number. So, for the decimal numbers, the value of r is 10, and $r-1$ is $10-1=9$. So, we can calculate the 10's and 9's complement of the number.

9's complement of the number 155 is calculated as:

$$\begin{aligned} &= \{(10^3)_{10} - 1\} - (155)_{10} \\ &= (1000 - 1) - 155 \\ &= 999 - 155 \\ &= (844)_{10} \end{aligned}$$

10's complement of the number 1011000 is calculated as:

$$\begin{aligned} &= (10^3)_{10} - (155)_{10} \\ &= 1000 - 155 \\ &= (845)_{10} \end{aligned}$$

Example 3: $(172)_8$

This number has a base of 8, which means it is an octal number. So, for the octal numbers, the value of r is 8, and $r-1$ is $8-1=7$. So, we can calculate the 8's and 7's complement of the number.

7's complement of the number 172 is calculated as:

$$\begin{aligned} &= \{(8^3)_{10} - 1\} - (172)_8 \\ &= ((512)_{10} - 1) - (132)_8 \\ &= (511)_{10} - (122)_{10} \\ &= (389)_{10} \\ &= (605)_8 \end{aligned}$$

8's complement of the number 172 is calculated as:

$$\begin{aligned} &= (8^3)_{10} - (172)_8 \\ &= (512)_{10} - 172_8 \\ &= 512_{10} - 122_{10} \\ &= 390_{10} \\ &= 606_8 \end{aligned}$$

Example 4: $(F9)_{16}$

This number has a base of 16, which means it is a hexadecimal number. So, for the hexadecimal numbers, the value of r is 16, and $r-1$ is $16-1=15$. So, we can calculate the 16's and 15's complement of the number.

15's complement of the number F9 is calculated as:

$$\begin{aligned} & \{(16^2)_{10}-1\}-(F9)_{16} \\ & (256-1)_{10}-F9_{16} \\ & 255_{10}-249_{10} \\ & (6)_{10} \\ & (6)_{16} \end{aligned}$$

16's complement of the number F9 is calculated as:

$$\begin{aligned} & \{(16^2)_{10}\}-(F9)_{16} \\ & 256_{10}-249_{10} \\ & (7)_{10} \\ & (7)_{16} \end{aligned}$$

Binary Addition and Subtraction

The addition and subtraction of the binary number system are similar to that of the decimal number system. The only difference is that the decimal number system consists the digit from 0-9 and their base is 10 whereas the binary number system consists only two digits (0 and 1) which make their operation easier.

Binary Addition

The binary number system uses only two digits 0 and 1 due to which their addition is simple. There are four basic operations for binary addition, as mentioned above.

$$\begin{aligned} 0+0 &= 0 \\ 0+1 &= 1 \\ 1+0 &= 1 \\ 1+1 &= 10 \end{aligned}$$

The above first three equations are very identical to the binary digit number. The column by column addition of binary is applied below in details. Let us consider the addition of 11101 and 11011.

$$\begin{array}{r}
 \\
 \\
 (+) \\
 \hline
 1
 \end{array}$$

Circuit Globe

The above sum is carried out by following step

$$1 + 1 = 10 = 0 \text{ with a carry of } 1.$$

$$1+0+1 = 10 = 0 \text{ with a carry of } 1$$

$$1+1+0 = 10 = 10 = 0 \text{ with a carry of } 1$$

$$1+1+1 = 10+1 = 11 = 1 \text{ with a carry of } 1$$

$$1 + 1 + 1 = 11$$

Note carefully that $10 + 1 = 11$, which is equivalent to two + one = three (the next binary number after 10)

Thus the required result is 111000.

Binary Subtraction

The subtraction of the binary digit depends on the four basic operations

$$0 - 0 = 0$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

$$10 - 1 = 1$$

The above first three operations are easy to understand as they are identical to decimal subtraction. The fourth operation can be understood with the logic two minus one is one.

For a binary number with two or more digits, the subtraction is carried out column by column as in decimal subtraction. Also, sometimes one has to borrow from the next higher column. Consider the following example.

$$\begin{array}{r}
 \leftarrow \text{borrow} \\
 1 1 0 \\
 (-) 1 0 0 \\
 \hline
 0 1 0 \\
 \hline
 \end{array}$$

Circuit Globe

The above subtraction is carried out through the following steps.

$$0 - 0 = 0$$

For $0 - 1 = 1$, taking borrow 1 and then $10 - 1 = 1$

For $1 - 0$, since 1 has already been given, it becomes $0 - 0 = 0$

$$1 - 1 = 0$$

Therefore the result is 0010.

Overflow in Arithmetic Addition in Binary number System

To explain the overflow of arithmetic addition, we are going to mostly use 2's complement system, which is a type of widely used number system in computer architecture.

- If the 2's complement number system has N-bit, it is able to represent a number from -2^{n-1} to $2^{n-1}-1$.
- If the 2's complement system has 4 bit, it will represent the number from (-8 to 7).
- If the 2's complement system has 5 bit, it will represent the number from (-16 to 15).

When we are trying to add 2 N-bit 2's complement numbers, and the generated output is so large that it is not fitted into that N-bit group, in this case, overflow will occur with respect to addition. There are usually N bit fixed registers into a system or computer.

When we add the two N-bit numbers, it will generate the output in max N+1 bit number. With the help of a carry flag, this extra bit is stored. The problem is that the carry does not always use to show an overflow.

For example: In this example, we are going to add 7 and 1 with the help of 2's complement.

Solution: The addition of binary number 7(0001) and 7(0111) is described as follows:

1. $0001 (1) + 0111 (7) = 1000 (-8)$

As we can see that adding $7 + 1$ with the help of 4 bit is equal to 8. But we cannot represent 8 with the help of 4 bit 2's complement number because the number 8 is out of range. When we add two positive numbers, we just get a negative number (-8). In this example, 0 is also the carry. Normally the problem related to an overflow is left to the programmer, and he has to deal with these situations.

Overflow Detection

At the time of showing the result of arithmetic operation, if the bits are insufficient to represent the binary number, the overflow will occur. The computer arithmetic is closed with respect to division, subtraction, multiplication, and subtraction; due to this, an overflow occurs. If the signs of the operands are different(resp. identical), in this case, the overflow will not occur.

There are two cases when the overflow occurs, which are described as follows:

1. When we try to add the two negative numbers and the generated result is a positive number.
2. When we try to add the two positive numbers and the generated result is a negative number.

For example:

INPUTS			OUTPUTS		
A _{sign}	B _{sign}	CARRY IN	CARRY OUT	SUM _{Sign}	OVERFLOW

0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

When the two positive numbers are added, and the generated result is a positive number, and when the two negative numbers are added, and the generated result is a negative number, then there will be no overflow, which is described as follows:

INPUTS		OUTPUTS			
A _{sign}	B _{sign}	CARRY IN	CARRY OUT	SUM _{Sign}	OVERFLOW
0	0	0	0	0	0
0	0	1	0	1	?
0	1	0	0	1	0
0	1	1	1	0	0

1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	?
1	1	1	1	1	0

If we have n bit number representation, then we will need $n + 1$ bit to detect and compensate for an overflow. For example: Suppose we have 32-bit arithmetic, we can detect and compensate for overflow with the help of 33 bits. We can take a carry (borrow) which is occurred into the sign bit, to implement this in addition. To explain it more deeply, we are going to provide four different combinations of a sign with the help of four-bit long binary representations of numbers 7 and 6.

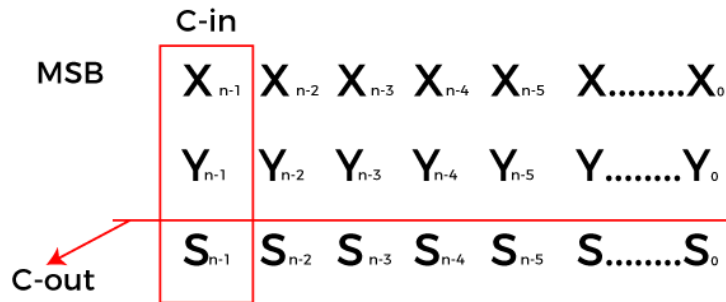
$$\begin{array}{r}
 \text{7 + 6} \\
 \begin{array}{cccc}
 0 & 1 & 1 & 1 \\
 + & 0 & 1 & 1 & 0 \\
 \hline
 1 & 1 & 0 & 1
 \end{array}
 \end{array}
 \begin{array}{l}
 (7_{\text{ten}}) \\
 (6_{\text{ten}}) \\
 (13_{\text{ten}})
 \end{array}$$

$$\begin{array}{r}
 \text{-7 + -6} \\
 \begin{array}{cccc}
 1 & 0 & 0 & 1 \\
 + & 1 & 0 & 1 & 0 \\
 \hline
 0 & 0 & 1 & 1
 \end{array}
 \end{array}
 \begin{array}{l}
 (-7_{\text{ten}}) \\
 (-6_{\text{ten}}) \\
 (-13_{\text{ten}})
 \end{array}$$

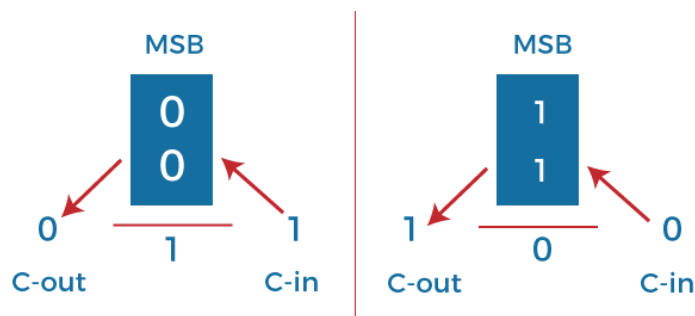
$$\begin{array}{r}
 \text{-7 - 6} \\
 \begin{array}{cccc}
 1 & 0 & 0 & 1 \\
 - & 0 & 1 & 1 & 0 \\
 \hline
 0 & 0 & 1 & 1
 \end{array}
 \end{array}
 \begin{array}{l}
 (-7_{\text{ten}}) \\
 (6_{\text{ten}}) \\
 (-13_{\text{ten}})
 \end{array}$$

$$\begin{array}{r}
 \text{-7 - 6 = -7 + -6} \\
 \begin{array}{cccc}
 1 & 0 & 0 & 1 \\
 + & 1 & 0 & 1 & 0 \\
 \hline
 0 & 0 & 1 & 1
 \end{array}
 \end{array}
 \begin{array}{l}
 (-7_{\text{ten}}) \\
 (-6_{\text{ten}}) \\
 (-13_{\text{ten}})
 \end{array}$$

In place of using the 3-bit comparator, we can use the 2-bit comparator to detect the overflow. We can also detect the overflow with the help of checking the MSB of two numbers and their result. For this, we need to just check the Carry-in (C-in) and Carry-out (C-out) bits from Most Significant Bits. Suppose we are going to perform N bit addition of 2's complement number, which is described as follows:



Overflow will occur when the C-in and C-out are equals to each other. With the help of below analysis, we can explain the above expression for an overflow.



In the first image, 0 shows the MSB of two numbers that indicate that they are positive. If we get 1 as Carry-in, the result of MSB will also be 1, which means that result is negative (Overflow). In this case, Carry-out will also be 0. It is proved that the Carry-in and Carry-out are not equal to each other, hence overflow.

In the second image, 1 shows the MSB of two numbers that indicate that they are negative. If we get 0 as Carry-in, the result of MSB will also be 0, which means that result is positive (Overflow). In this case, Carry-out will also be 1. It is proved that the Carry-in is not equal to Carry-out hence overflow.

With the help of above explanation, we can say that the overflow can be detected by the C-in and C-out at MSB (Most Significant Bit), which is described as follows:



With the help of above XOR gate, we can easily detect the overflow.

Conditions of Overflow

There are various conditions of an overflow, which are described as follows:

- The arithmetic operation has the capability that it can run into the **overflow** condition.
- On the basis of the size of data type, which will accommodate the result, the overflow is dependent.
- If the result's data type is too small or too large that it is not fitted into the original data type, in this case, an overflow will occur.
- When we try to add the two signed 2's complement numbers, the condition of an overflow will occur if both the numbers are positive and the generated output is negative, or if both the numbers are negative and the generated output is positive.
- When we try to add the two unsigned numbers, the condition of an overflow will occur if the left-most bit contains the carry-out.

Floating Point Representation

There are posts on representation of floating point format. The objective of this article is to provide a brief introduction to floating point format.

The following description explains terminology and primary details of IEEE 754 binary floating-point representation. The discussion confines to single and double precision formats.

Usually, a real number in binary will be represented in the following format,

$$I_m I_{m-1} \dots I_2 I_1 I_0 . F_1 F_2 \dots F_n F_{n-1}$$

Where I_m and F_n will be either 0 or 1 of integer and fraction parts respectively.

A finite number can also be represented by four integer components, a sign (s), a base (b), a significant (m), and an exponent (e). Then the numerical value of the number is evaluated as

$$(-1)^s \times m \times b^e \text{ ——— Where } m < |b|$$

Depending on base and the number of bits used to encode various components, the [IEEE 754](#) standard defines five basic formats. Among the five formats, the binary32 and the binary64 formats are single precision and double precision formats respectively in which the base is 2.

Table – 1 Precision Representation

Precision	Base	Sign	Exponent	Significant
Single precision	2	1	8	23+1
Double precision	2	1	11	52+1

Single Precision Format:

As mentioned in Table 1 the single precision format has 23 bits for significant (1 represents implied bit, details below), 8 bits for exponent and 1 bit for sign.

For example, the rational number $9 \div 2$ can be converted to single precision float format as following,

$$9_{(10)} \div 2_{(10)} = 4.5_{(10)} = 100.1_{(2)}$$

The result said to be **normalized**, if it is represented with leading 1 bit, i.e. $1.001_{(2)} \times 2^2$. (Similarly when the number $0.000000001101_{(2)} \times 2^3$ is normalized, it appears as $1.101_{(2)} \times 2^{-6}$). Omitting this implied 1 on left extreme gives us the **mantissa** of float number. A normalized number provides more accuracy than corresponding **de-normalized** number. The implied most significant bit can be used to represent even more accurate significant (23 + 1 = 24 bits) which is called **subnormal** representation. *The floating point numbers are to be represented in normalized form.*

The subnormal numbers fall into the category of de-normalized numbers. The subnormal representation slightly reduces the exponent range and can't be normalized since that would result in an exponent which doesn't fit in the field. Subnormal numbers are less accurate, i.e. they have less room for nonzero bits in the fraction field, than normalized numbers. Indeed, the accuracy drops as the size of the subnormal number decreases. However, the subnormal representation is useful in filling gaps of floating point scale near zero.

In other words, the above result can be written as $(-1)^0 \times 1.001_{(2)} \times 2^2$ which yields the integer components as $s = 0$, $b = 2$, significant (m) = 1.001, mantissa = 001 and $e = 2$. The corresponding single precision floating number can be represented in binary as shown below,



Where the exponent field is supposed to be 2, yet encoded as 129 (127+2) called **biased exponent**. The exponent field is in plain binary format which also represents negative exponents with an encoding (like sign magnitude, 1's complement, 2's complement, etc.). The biased exponent is used for the representation of negative exponents. The biased exponent has advantages over other negative representations in performing bitwise comparing of two floating point numbers for equality.

A **bias** of $(2^{n-1} - 1)$, where **n** is # of bits used in exponent, is added to the exponent (**e**) to get biased exponent (**E**). So, the biased exponent (**E**) of *single precision* number can be obtained as

$$E = e + 127$$

The range of exponent in single precision format is -128 to +127. Other values are used for special symbols.

Note: When we unpack a floating point number the exponent obtained is the biased exponent. Subtracting 127 from the biased exponent we can extract unbiased exponent.

Double Precision Format:

As mentioned in Table – 1 the double precision format has 52 bits for significant (1 represents implied bit), 11 bits for exponent and 1 bit for sign. All other definitions are same for double precision format, except for the size of various components.

Precision:

The smallest change that can be represented in floating point representation is called as precision. The fractional part of a single precision normalized number has exactly 23 bits of resolution, (24 bits with the implied bit). This corresponds to $\log_{(10)} (2^{23}) = 6.924 = 7$ (the characteristic of logarithm) decimal digits of accuracy. Similarly, in case of double precision numbers the precision is $\log_{(10)} (2^{52}) = 15.654 = 16$ decimal digits.

Accuracy:

Accuracy in floating point representation is governed by number of significant bits, whereas range is limited by exponent. Not all real numbers can exactly be represented in floating point format. For any number which is not floating point number, there are two options for floating point approximation, say, the closest floating point number *less than* **x** as **x₋** and the closest floating point number *greater than* **x** as **x₊**.

A **rounding** operation is performed on number of significant bits in the mantissa field based on the selected mode. The **round down** mode causes x set to x_- , the **round up** mode causes x set to x_+ , the **round towards zero** mode causes x is either x_- or x_+ whichever is between zero and. The **round to nearest** mode sets x to x_- or x_+ whichever is nearest to x . Usually **round to nearest** is most used mode. The closeness of floating point representation to the actual value is called as **accuracy**.

Special Bit Patterns:

The standard defines few special floating point bit patterns. Zero can't have most significant 1 bit, hence can't be normalized. The hidden bit representation requires a special technique for storing zero. We will have two different bit patterns +0 and -0 for the same numerical value zero. For single precision floating point representation, these patterns are given below,

0 00000000 000000000000000000000000 = +0

1 00000000 000000000000000000000000 = -0

Similarly, the standard represents two different bit patterns for +INF and -INF. The same are given below,

0 11111111 000000000000000000000000 = +INF

1 11111111 000000000000000000000000 = -INF

All of these special numbers, as well as other special numbers (below) are subnormal numbers, represented through the use of a special bit pattern in the exponent field. This slightly reduces the exponent range, but this is quite acceptable since the range is so large.

An attempt to compute expressions like $0 \times \text{INF}$, $0 \div \text{INF}$, etc. make no mathematical sense. The standard calls the result of such expressions as Not a Number (NaN). Any subsequent expression with NaN yields NaN. The representation of NaN has non-zero significant and all 1s in the exponent field. These are shown below for single precision format (x is don't care bits),

x 11111111 **m**000000000000000000000000

Where **m** can be 0 or 1. This gives us two different representations of NaN.

0 11111111 **0**000000000000000000000001 _____ Signaling NaN (SNaN)

0 11111111 **1**000000000000000000000001 _____ Quiet NaN (QNaN)

Usually QNaN and SNaN are used for error handling. QNaN do not raise any exceptions as they propagate through most operations. Whereas SNaN are which when consumed by most operations will raise an invalid exception.

Overflow and Underflow:

Overflow is said to occur when the true result of an arithmetic operation is finite but larger in magnitude than the largest floating point number which can be stored using the given precision. **Underflow** is said to occur when the true result of an arithmetic operation is smaller in magnitude (infinitesimal) than the smallest normalized floating point number which can be stored. Overflow can't be ignored in calculations whereas underflow can effectively be replaced by zero.

Endianness:

The IEEE 754 standard defines a binary floating point format. The architecture details are left to the hardware manufacturers. The storage order of individual bytes in binary floating point numbers varies from architecture to architecture

Addition and subtraction with signed magnitude data

A signed-magnitude method is used by computers to implement floating-point operations. Signed-2's complement method is used by most computers for arithmetic operations executed on integers. In this approach, the leftmost bit in the number is used for signifying the sign; 0 indicates a positive integer, and 1 indicates a negative integer. The remaining bits in the number supported the magnitude of the number.

Example: -2410 is defined as –

10011000

In this example, the leftmost bit 1 defines negative, and the magnitude is 24.

The magnitude for both positive and negative values is the same, but they change only with their signs.

The range of values for the sign and magnitude representation is from **-127** to **127**.

There are eight conditions to consider while adding or subtracting signed numbers. These conditions are based on the operations implemented and the sign of the numbers.

The table displays the algorithm for addition and subtraction. The first column in the table displays these conditions. The other columns of the table define the actual operations to be implemented with the magnitude of numbers. The last column of the table is needed to avoid a negative zero. This defines that when two same numbers are subtracted, the output must not be - 0. It should consistently be +0.

In the table, the magnitude of the two numbers is defined by P and Q.

Addition and Subtraction of Signed Magnitude Numbers

Operations	Addition of Magnitudes	Subtraction of Magnitudes		
$(+P) + (+Q)$	$+(P+Q)$	$P > Q$	$P < Q$	$P = Q$
$(+P) + (-Q)$		$+(P-Q)$	$-(Q-P)$	$+(P-Q)$
$(-P) + (+Q)$		$-(P-Q)$	$+(Q-P)$	$+(P-Q)$
$(-P) + (-Q)$	$-(P+Q)$			
$(+P) - (+Q)$		$+(P-Q)$	$-(Q-P)$	$+(P-Q)$
$(+P) - (-Q)$	$+(P+Q)$			
$(-P) - (+Q)$	$-(P+Q)$			
$(-P) - (-Q)$		$-(P-Q)$	$+(Q-P)$	$+(P-Q)$

As display in the table, the addition algorithm states that –

- When the signs of P and Q are equal, add the two magnitudes and connect the sign of P to the output.
- When the signs of P and Q are different, compare the magnitudes and subtract the smaller number from the greater number.
- The signs of the output have to be equal as P in case $P > Q$ or the complement of the sign of P in case $P < Q$.
- When the two magnitudes are equal, subtract Q from P and modify the sign of the output to positive.

The subtraction algorithm states that –

- When the signs of P and Q are different, add the two magnitudes and connect the signs of P to the output.
- When the signs of P and Q are the same, compare the magnitudes and subtract the smaller number from the greater number.
- The signs of the output have to be equal as P in case $P > Q$ or the complement of the sign of P in case $P < Q$.
- When the two magnitudes are equal, subtract Q from P and modify the sign of the output to positive.

Multiplication Algorithm & Division Algorithm

Introduction

Arithmetic instructions in digital computers work on data to produce the results necessary for solving computational problems. These instructions are responsible for processing data on a computer. There are four basic arithmetic operations, addition, subtraction, multiplication, and division.

Multiplication Algorithm

Multiplication of fixed-point binary numbers in signed-magnitude representation is done by successive shift and add operations. For example, multiplication of numbers 10111(23) and 10011(19).

```
23   10111
19   x 10011
```

```
      10111
      10111x
      00000xx   +(adding all)
      00000xxx
```

110110101 Product 437

The process consists of looking at successive Multiplier, least significant bit first. If the Multiplier is 1, the multiplicand is copied down; otherwise, zero is copied. And like we do in standard multiplication, the numbers copied down in successive lines are shifted one position to the left. Finally, all binaries are added, and the total sum is the result. The sign of the product(result) is determined from the signs of multiplicand and Multiplier. If they are alike, the final product sign is positive. If they are unlike, the sign of the product is negative.

Division Algorithm

Division to two fixed-point binary numbers in signed-magnitude representation is done by the process of successive compare, shift, and subtract operations. The binary division is simpler than decimal because the quotient is either 0 or 1. There is no need to calculate how many times the dividend or partial remainder fits into the divisor. You can follow the following steps for binary division.

Step 1: Compare the divisor with the dividend; if the divisor is greater, place 0 as the quotient, then bring down the second bit of the dividend. If the divisor is smaller, multiply it by one, and the result must be subtracted. Then, subtract the result from the above to get the remainder.

Step 2: Bring down the next number bit from the dividend and perform step1.

Step 3: Repeat the whole process until the remainder becomes 0, or the whole dividend is divided.

