UNIT 2 DATA REPRESENTATION

Structure **Page Nos.**

2.0 Introduction
2.1 Objectives
2.2 Data Representation in Computer
2.3 Representation of Characters
2.4 Number Systems
2.5 Negative Number Representation Using Complements
 2.5.1 Fixed Point Representation
 2.5.2 Binary Arithmetic using Complement notation
 2.5.3 Decimal Fixed Point Representation
2.6 Floating Point Representation
2.7 Error Detection and Correction Codes
2.8 Summary
2.9 Solutions/ Answers

2.0 INTRODUCTION

In the first Unit of this Block, you have learnt the concepts relating to different architectures of a Computer System. It also explains the process of execution of an instruction highlighting the use of various components of a Computer system. This Unit explains about how the data is represented in a computer system.

The Unit first defines the concepts of number systems in brief, which is followed by discussion on conversion of numbers of different number systems. An important concept of signed complement notation, which is used for arithmetic operations on binary numbers, has been explained in this Unit. This is followed by discussion on the fixed point and floating point numbers, which are used to represent the numerical data in computer systems. This Unit also explains the error detection and correction codes and introduces you to basics of computer arithmetic operations.

2.1 OBJECTIVES

At the end of the unit you will be able to:

- Represent numeric data using binary, octal and hexadecimal numbers;
- Perform number conversions among various number bases;
- Define the ASCII and UNICODE representation for a character set;
- Explain the fixed and floating point number formats;
- Perform arithmetic operations using fixed and floating point numbers ; and
- Explain error detection and correction codes

2.2 DATA REPRESENTATION IN COMPUTER

A computer system is an electronic device that processes data. An electronic device, in general, consists of two stable states represented as 0 and 1. Therefore, the basic unit of data on a computer is called a **B**inary Dig**it** or a **Bit.** With the advances in quantum computing technology a new basic unit called Qubits has emerged, which also represent 0 and 1, but with the difference that it can also represent both the states at the same time. The concepts of Quantum computing is beyond the scope of this unit.

A computer performs three basic operation on data, viz. data input, processing and data output. The data input and information output, in general, is presented in text, graphics, audio or other human recognizable form. Therefore, all human readable characters, graphics, audio and video should be coded using bits such that computer is able to interpret them. The most common code to represents characters into computer are ASCII and UNICODE. Pictures and graphs can be represented using pixel (picture elements), digital sound and video are represented by coding the frames in digital formats. Since graphics, digital audio and digital video, which are stored on storage devices as files, are very large in size, therefore, a large number of storage formats that use data compression techniques are used for represent digital information. Some of these concepts are explained in Unit 8.

The numeric data is used for computation in computer. However, as computer is an electronic device, it can only process binary data. Thus, in general, numeric data is to be converted to binary for computation. Computer uses fixed point and floating point representation for representing numeric data. Data in computer is stored in random access memory (RAM) and is required to be transferred in or out of the RAM for the purpose of processing, therefore, an error detection mechanism may be employed to identify and correct simple errors while transfer of binary data. The subsequent sections of the Unit explains the character representation, representation of binary numbers and error detection mechanism.

2.3 REPRESENTATION OF CHARACTERS

A character can be presented in a computer using a binary code. This code should be same for different types of computers, else the information from one computer will not be transferable to other computers. Thus, there is need of a standard for character representation. A coding standard has to address two basic issues, viz. the length of code, and organisation of different types of characters, which include printable character set of different languages and special characters. Two important character representation schemes are ASCII and UNICODE, which are discussed next.

American Standard Code for Information Interchange (ASCII)

ASCII was among the first character encoding standard. The length of ASCII is 7-bit. Thus, it can represent 2^7=128 characters. It represents printable characters - English alphabets (both lower case and upper case), decimal digits, special characters as present on the present day keyboard, certain graphical characters etc.; and non-printable control characters. American National Standards Institute (ANSI) has designed a standard ISO 646 in 1964, which is based on ASCII.

However, as the basic unit of computer storage was 8, 16, 32 or 64 bits, the ASCII was extended to create an 8 bit code. This code could represent 2^8=256 characters, most of the additional characters in extended code were graphics characters. ANSI has designed a code ISO 8859 for extended ASCII. It may be noted that ASCII has many variants, which are based on characters used in different countries.

In ASCII, the coding sequence of characters is very interesting. Simple binary arithmetic operations can result in conversion of lower case to upper case characters. For example, character 'A' in ASCII is represented as the binary value 100 0001, which is equivalent to a value 65 in decimal notation, whereas the character 'a' is stored as binary value 110 0001, which is the decimal 97. Thus, conversion from lowercase to upper case and vice-versa may be performed by subtracting or adding 32, as the case may be.

ISO 8859 which is based on extended ASCII is a good representation; however, all the languages cannot be represented using ASCII as its length is very small. Therefore, a new standard that could represent almost all the characters of all the languages was developed. This is called the UNICODE.

Unicode

Unicode is a standard for character representation, which provides a unique code also called *code point*, for every character of almost all the languages of the world. The set of all the codes is called *code space*. The code space is divided into 17 continuous sequences of codes called *code planes*, with each code plane can represent 2^{16} codes. Thus, Unicode values ranges from U+0000_{16} to U+$10FFFF_{16}$. Here U+ represents the Unicode followed by the hexadecimal value of a code point. The code planes of the Unicode being U+00000_{16} to U+$0FFFF_{16}$; U+10000_{16} to U+$1FFFF_{16}$; U+20000_{16} to U+$2FFFF_{16}$; … , U+$F0000_{16}$ to U+$FFFFF_{16}$; and U+100000_{16} to U+$10FFFF_{16}$. You can learn about more details on Unicode from the further readings. Also read the hexadecimal number system given in the next section to learn about the hexadecimal values given above.

One of the major advantages of using Unicode is that it helps in seamless digital data transfer among the applications that use this character formatting, thus, not causing any compatibility problem.

Unicode code points may consist of about 24 binary digits, however, all of these code points may not be required for a given set of data. In addition, a digital system requires the data in the units of bytes. Thus, a number of encodings has been designed to represent Unicode code points in a digital format. Two of these popular encodings - Unicode Transformation Formats are UTF-8 and UTF-16. UTF-8 uses 1 to 4 bytes to represent the code points of Unicode. Most of the 1 byte UTF-8 code points are compatible to ASCII. UTF-16 represents code points as one or two 16-bit code units. The standard ISO 10646 represents various Unicode coding formats.

In general, if you are working with web pages having mostly English language, UTF-8 may be a good choice of character representation. However, if you are creating a multi-lingual web page, it may be a good idea to use UTF-16.

Indian Standard Code for information interchange (ISCII)

The ISCII is and ASCII compatible code consisting of eight-bits. The code for values 0 to 127 in ISCII is similar to ASCII; however, for the values 128 to 225 it represents the characters of Indian scripts. IS 13194:1991 BIS standard defines the details of ISCII. However, with the popularity of Unicode, its use has now been limited.

2.4 NUMBER SYSTEMS

A number system is used to represent the quantitative information. This section discusses the binary, octal and hexadecimal number systems

Formally, a number system is represented using a base or radix, which is equal to the distinct digits used by that system, and the position of a digit in a number. For example, the decimal number system has a base 10. It consists of ten decimal digits, viz. 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9; and a place value as shown in the following example:

$$9\times10^4 + 8\times10^3 + 7\times10^2 + 6\times10^1 + 5\times10^0 + 4\times10^{-1} + 3\times10^{-2} + 2\times10^{-3} + 1\times 10^{-4}$$
$$= 98765.4321$$

Binary Numbers: A binary number system has a base 2 and consists of only two digits 0 and 1, which are also called the bits. For example, 1001_2 represent a binary number with four binary digits. The subscript 2 represents that the number 1001 has a base 2 or in other words is a binary number.

Note: The subscript shown in the numbers represents the base of the number. In case a subscript is not given then please assume it as per the context of discussion.

Conversion of binary number to Decimal equivalent:

A binary number is converted to its decimal equivalent by multiplying each binary digit by its place value. For example, a seven digit binary number 1001001_2 can be converted to decimal equivalent value as follows:

Binary Digits of Number	1	0	0	1	0	0	1
The place value	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
Binary digit × Place value	1×64	0×32	0×16	1×8	0×4	0×2	1×1
Computed values	64	0	0	8	0	0	1
Sum of the computed values	64+0+0+8+0+0+1 = 73 in Decimal						

You may now try converting few more numbers. Try 0010001, which will be 16+1=17; 1111111 will be 64+32+16+8+4+2+1=127. So a 7 bit binary number can contain decimal values from 0 to 127.

Octal Numbers: An Octal number system has a base of 8, therefore, it has eight digits, which are 0,1,2,3,4,5,6,7. For example, 76543210_8 is an octal number.

Conversion of Octal number to Decimal equivalent:

An Octal number is converted to its decimal equivalent by multiplying each octal digit by its place value. For example, an octal number 5432_8 can be converted to decimal equivalent value as follows:

Octal Digits of Number	5	4	3	2
The place value	8^3 =512	8^2 =64	8^1 =8	8^0 =1
Octal digit × Place value	5×512	4×64	3×8	2×1
Computed values	2560	256	24	2
Sum of the computed values	$2560+256+24+2=2842_{10}$			

Hexadecimal Numbers: A hexadecimal number system has a base of 16, therefore, it uses sixteen digits, which are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A(10), B(11), C(12), D(13), E(14), F(15). For example, $FDA9_{16}$ is a hexadecimal number.

Conversion of Hexadecimal number to Decimal equivalent:

A hexadecimal number is converted to its decimal equivalent by multiplying each hexadecimal digit by its place value. For example, a hexadecimal number $13AF_{16}$ can be converted to decimal equivalent value as follows:

Hexadecimal Digits of Number	1	3	A=10	F=15
The place value	16^3 =4096	16^2 =256	16^1 =16	16^0 =1
Hexadecimal digit × Place value	1×4096	3×256	10×16	15×1
Computed values	4096	768	160	15
Sum of the computed values	$4096+768+160+15=5039_{10}$			

Conversion of Decimal to Binary: A decimal number can consists of Integer and fraction part. Both are converted to binary separately.

Process:

For Integer part: Repetitively divide the quotient of integer part by 2 keeping remainder separate till quotient is 0. Collect all the remainders from last remainder to first to make equivalent binary

For Factional part: Repetitively multiply the fraction by 2 and maintain the list of integer value that is obtained till fraction becomes 0. Collect all the integer values.

The following example explains the process of Decimal to binary conversion.

Example 1: Convert the decimal number 22.25 to binary number.

Solution:

For Integer part: Repetitively divide the quotient of integer part by 2 keeping remainder separate till quotient is 0. Integer value of example: 22	**For Factional part:** Repetitively multiply the fraction by 2 and maintain the list of integer value that is obtained till fraction becomes 0. Fraction value of example:.25

Integer Part	**After Division by 2**		**Direction of Reading the Result**
	Quotient	**Remainder**	
22	11	0	↑
11	5	1	
5	2	1	
2	1	0	
1	0	1	
0	**STOP**		**Ans: 10110**

Fraction Part	**After multiplication by 2**		**Direction of Reading the Result**
	Result	**Integer part**	
.25	0.50	0	↓
.50	1.00	1	
.00	**STOP**		**Ans: 01**

22.25_{10} in binary is 10110.01_2

Verification

Binary Digits of Number	**1**	**0**	**1**	**1**	**0**	**.**	**0**	**1**
The place value	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1		2^{-1} =1/2	2^{-2} =1/4
Digit × Place value	1×16	0×8	1×4	1×2	0×1	.	0×0.5	1×0.25
Computed values	16	0	4	2	0		0	0.25
Sum of the computed values	22.25_{10}							

--

The method as shown above is a standard technique. You must start using this method for various problems. However, you may use the following simple technique of converting integer part of decimal numbers to binary.

Conversion from Decimal to Binary a simpler Process:

Assume a decimal number *N* is to be converted to binary. Now perform the following steps:

1. Is the decimal number *N* equal to a binary place value, then assign that place value to *P* and move to step 3.

2. Else, find the binary place values which is just lower to the decimal number *N*. Assign this place value to *P*. For example, for number 73 just lower place value is 64, as $2^6 = 64$ and 2^7=128.

3. Put 1 in the position of *P* and subtract the place value *P* from *N*.

4. If (N-P) ≠ 0 then Repeat the steps 1 to 3 by taking new N=N-P

5. Put 0 in all the remaining places, where 1 has not been put.

The following example demonstrate this technique showing conversion of 73 to binary.

Example 2: Convert decimal numbers 73, 39 and 20 into binary using the method as above.

The following table shows the process of the conversion.

The place value	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
***N* = 73**	128 > 73 > 64, therefore, *P*=64						
Step 2 and 3	1	*N* = *N-P* = 73-64 =9 (Not 0 so repeat)					
Step 2	16 > 9 > 8 so new *P*=8 and New *N*=9-8=1						
Step 3	1			1			
Step 1	Since 1 is a place value, new *P*=1 and New *N* =1-1=0						
Step 3	1			1			1
Step 4	**1**	**0**	**0**	**1**	**0**	**0**	**1**
Place values	64	32	16	8	4	2	1
N=39		1					
New *N=7*		1			1		
New *N=3*		1			1	1	
New *N=1*		1			1	1	1
Step 4	**0**	**1**	**0**	**0**	**1**	**1**	**1**
Place values	64	32	16	8	4	2	1
N=20			1				
New *N=4*			1		1		
Step 4	**0**	**0**	**1**	**0**	**1**	**0**	**0**

The logic as presented here can be extended to the fractional part, however, it is recommended that you may follow the repeated multiplication method as explained earlier for the fractions.

Conversion of Binary number to Octal Number

The base of a binary number is 2 and the base of octal number is 8. Interestingly, 2^3=8. Thus, if you simply group three binary digits, the equivalent value may form the octal digit. However, you may be wondering how to group binary numbers. This is explained with the help of following example.

Example 3: Convert the binary 11001101.00111_2 into equivalent Octal number.

Process: The process is to group three binary digits. The grouping before the binary point is done from right to left and after the binary point from left tonright. Each of the group then is converted to equivalent octal digit. The following table shows this conversion process.

Binary Number	-	1	1	0	0	1	1	0	1	.	0	0	1	1	1	-
Grouping Directions	←									.	→					
Grouped (- replaced by 0)	0 1 1			0 0 1			1 0 1			.	0 0 1			1 1 0		
Binary place values	4	2	1	4	2	1	4	2	1	.	4	2	1	4	2	1
Equivalent Octal Digit	0+2+1=3			0+0+1=1			4+0+1=5			.	0+0+1=1			4+2+0=6		
Octal Number	**3**			**1**			**5**			**.**	**1**			**6**		

Therefore, 11001101.00111_2 is equivalent to **315.16_8**

Conversion of Binary number to Hexadecimal Number

The base of a binary number is 2 and the base of hexadecimal number is 16. You may notice that $2^4=16$. Therefore, conversion of binary to hexadecimal notation may require grouping of 4 binary digits. This is explained with the help of following example.

Example 4: Convert the binary 11001101.00111_2 into equivalent hexadecimal number.

Process: The process is almost similar to binary number to octal number conversion expect now four binary digits are combined as given in the following table.

Binary Number	1	1	0	0	1	1	0	1	.	0	0	1	1	1	-	-	-
Grouping Direction	←								.	→							
Grouped	1 1 0 0				1 1 0 1				.	0 0 1 1				1 0 0 0			
Binary place values	8	4	2	1	8	4	2	1	.	8	4	2	1	8	4	2	1
Hexadecimal digit	8+4+0+0=12				8+4+0+1=13				.	0+0+2+1=3				8+0+0+0=8			
Hexadecimal	12 is C				13 is D				.	3				8			

Therefore, $\mathbf{11001101.00111_2}$ is equivalent to $\mathbf{315.16_8}$ and $\mathbf{CD.38_{16}}$

As computer is a binary device, therefore, all the numbers of different number systems may be represented in binary format. This is shown in the following table.

Decimal Number	Binary Coded Decimal	Equivalent Octal Number	Binary coded Octal	Hexadecimal Number	Binary-coded Hexadecimal
0	0000	0	000	0	0000
1	0001	1	001	1	0001
2	0010	2	010	2	0010
3	0011	3	011	3	0011
4	0100	4	100	4	0100
5	0101	5	101	5	0101
6	0110	6	110	6	0110
7	0111	7	111	7	0111
8	1000	10	001 000	8	1000
9	1001	11	001 001	9	1001
10	0001 0000	12	001 010	A	1010
11	0001 0001	13	001 011	B	1011
12	0001 0010	14	001 100	C	1100
13	0001 0011	15	001 101	D	1101
14	0001 0100	16	001 110	E	1110
15	0001 0101	17	001 111	F	1111
16	0001 0110	21	010 000	10	0001 0000
17	0001 0111	22	010 001	11	0001 0001
…					
49	0100 1001	61	110 001	31	0011 0001
…					
63	0110 0110	77	111 111	3F	0011 1111

Table 1: Decimal, Octal, Hexadecimal Numbers

Please note the following points in the Table 1 given above.

- The Binary coded decimal (BCD) is the representation of each decimal digit to a sequence of 4 bits. For example, a decimal number 12 in BCD is 0001 0010. This representation is used in several calculators for performing computation.
- It may be noted that BCD is not binary equivalent value. For example, the BCD value of decimal 49 is 0100 1001 but its binary equivalent value is 0011 0001.
- Please also note that binary coded hexadecimal values are equivalent to binary value of a number. For example, decimal value 63 in hexadecimal binary notation is 0011 1111, which is same as its binary value.

The conversion of decimal to octal and hexadecimal may be performed in the same way as done using repeated division or multiplication of binary. The process is exactly same except, in decimal number to octal or hexadecimal number conversion division is done by 8 or 16 respectively.

Check Your Progress 1

1) Perform the following conversions:

   i) 11100.01101_2 to Octal and Hexadecimal

   ii) 1101101010_2 to Octal and Hexadecimal

   ....

   ....

   ....

2) Convert the following numbers to binary.

   i) 119_{10}

   ii) 19.125_{10}

   iii) 325_{10}

   ....

   ....

   ....

3) Convert the numbers to hexadecimal and octal.

   i) 119_{10}

   ii) 19.125_{10}

   iii) 325_{10}

   ....

   ....

   ....

2.5 NEGATIVE NUMBER REPRESENTATION USING COMPLEMENTS

You have gone through the details of binary representation of character data and the number systems. In general, you use positive and negative integers and real numbers for computation. How these numbers can be represented in binary? This section describes how positive and negative numbers can be represented in binary for performing arithmetic operations.

In general, Integer numbers can be represented using the sign and magnitude of the number, whereas real numbers may be represented using a sign, decimal point and magnitude of integral and fractional part. Real numbers can also be represented using a scientific exponential notation. This section explains how integers can be represented as binary numbers in a computer system.

Integer representation in binary:

An integer is represented in binary using fixed number of binary digits. One of the simplest representations for representing integer would be - to represent the sign using a bit; and magnitude may be represented by the remaining bits. Fortunately, the value of sign can be either positive or negative, therefore, it can easily be represented in binary. The + sign can be represented using 0 and – sign can be represented using 1.

For example, a decimal number 73 has a sign + (bit value 0) and magnitude 73 (binary equivalent 1001001). The following table shows some of the numbers using this *Sign-magnitude Representation*:

Number	Sign Bit	Magnitude						
+73	0	1	0	0	1	0	0	1
-73	1	1	0	0	1	0	0	1
+39	0	0	1	0	0	1	1	1
-39	1	0	1	0	0	1	1	1
+127	0	1	1	1	1	1	1	1
-127	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
-0	1	0	0	0	0	0	0	0

Table 2: 8-bit Sign-Magnitude representation

Please note the following points about this 8 bit sign-magnitude representation:

- It represents number in the range 0 to +127 and -0 to -127. Therefore, the range of the numbers that can be represented using these 8 bits is -127 to +127.
- It represents 255 different numbers viz. -127 to -1, ±0 and +1 to +127.
- The number of bits, which are used to represent the magnitude of the number, can be used to determine the maximum and minimum numbers that are representable.
- There are two representations of zero +0 and -0

Is this representation suitable for representing numbers for computation? It has one basic problem that the sequences of steps to perform arithmetic operations are not straight forward. For example, for adding +73 and -39, first you need to compare the sign of the numbers, and as the signs are different in this case, therefore, you should perform subtraction of smaller number from the bigger number and finally assigning the sign of bigger number to the result.

Is there any better representation? Yes, an interesting representation that uses complement of a number to represent negative numbers has been designed. What is a complement of a number?

Complement notation: A complement, by definition, is a number that makes a given number complete. For the decimal numbers, this completeness can be defined with respect to the highest value of the digit, i.e. 9 or the next higher value, i.e. 10. These are called 9's and 10's complement respectively for the decimal numbers.

For example, for a decimal digit 3, the 9's complement would be 9-3 =6 and 10's complement would be 10-3=7.

In general, for a number with base *B* two types of complements are defined –(*B*-1)'s complement and *B*'s complement. For example, for decimal system base value *B* is10. Therefore, for decimal numbers two complements, viz. 9's and 10's complements, are defined. Thus, for binary system where base is 2, the two complements, viz. 1's complement and 2's complement, are defined. The following example illustrates the steps of finding 9's and 10's complement for decimal numbers.

Example 5: Compute the 9's complement and 10's complement for a four digit decimal number 1095, 8567 and 0560.
Solution: Following table shows the process:

Complement	Operation	The Number			
9's Complement	Number	1	0	9	5
	Subtract each digit from 9	8	9	0	4
10's Complement	Add 1 in the 9's complement	-	-	-	1
	It results in 10's complement	8	9	0	5
9's Complement	Number	8	5	6	7
	Subtract each digit from 9	1	4	3	2
10's Complement	Add 1 in the 9's complement	-	-	-	1
	It results in 10's complement	1	4	3	3
9's Complement	Number	0	5	6	0
	Subtract each digit from 9	9	4	3	9
10's Complement	Add 1 in the 9's complement	-	-	-	1
	It results in 10's complement	9	4	4	0

Table 3: Computation of 9's and 10's complement

Please note that the sum of the number and its 9's complement for the numbers of 4 digits is 9999, and the sum of the number and its 10's complement is 10000. The 9's and 10's complement of the numbers can be used in a computer system when BCD numbers are used instead of binary numbers. Similarly, the 1's and 2's complement can be computed for binary numbers. The following example demonstrates the complement notation in binary.

Example 6: Compute the 1's and 2's complement for the binary numbers 1001_2, 1111_2 and 0000_2 using representation, which has four bits.

Solution:

Solution: Following table shows the process:

Complement	Operation	The Number			
1's Complement	Number	1	0	0	1
	Subtract each digit from 1	0	1	1	0
2's Complement	Add 1 in the 1's complement	-	-	-	1
	It results in 2's complement	0	1	1	1
1's Complement	Number	1	1	1	1
	Subtract each digit from 1	0	0	0	0

2's Complement	Add 1 in the 1's complement	-	-	-	1
	It results in 2's complement	0	0	0	1
1's Complement	Number	0	0	0	0
	Subtract each digit from 1	1	1	1	1
2's Complement	Add 1 in the 1's complement	-	-	-	1
	It results in 2's complement. There will be a carry bit from the last digit	0	0	0	0

Table 4: Computation of 1's and 2's complement

Please note the following in the table as above:

- On subtracting 1 from binary digit will result in change of bit from 0 to 1 OR 1 to 0.
- When you add binary digit 1 with 1, then it results in a sum bit of 0 and carry bit as 1.
- 1's complement of 0000 is 1111, when 1 is added to it, you will get 10000 as the 2's complement. Since, only 4 binary digits are used in the notation as above, the fifth digit, which is 1, is ignored while taking the complement.

An interesting observation from the Table 4 is that 1's complement can be obtained simply by changing 1 to 0 and 0 to 1. For obtaining 2's complement leave all the trailing zeros and the first 1 intact and after that complement the remaining bits. For example, for an eight bit binary number 10101100, the complement can be done as follows:

Number	1	0	1	0	1	1	0	0
1's Complement change every bit from 0 to 1 OR 1 to 0	0	1	0	1	0	0	1	1
Number	1	0	1	0	1	1	0	0
For 2's complement leave the trailing 0's till first 1						1	0	0
then complement remaining bits(change 0 to 1 or 1 to 0)	0	1	0	1	0			
2's Complement of the Number	0	1	0	1	0	1	0	0

Table 5: Computation of 1's and 2's complement

But, how are these complement notations used in a computer system to represent integers? The next sub-section explains the integral representation of computers.

2.5.1 Fixed Point Representation

A computer system uses registers or memory locations to store arithmetic data like numbers. The number stored in these locations are of fixed size, such as 8 or 16 or 32 or 64 or 128 bits etc. Interestingly, binary point is not represented in the numbers, rather its location is assumed. The fixed point number representation assumes that the binary point is at the end of all the binary digits, thus, can be used to represent integers. Since, Integers include positive and negative number both, therefore, fixed point number also use one bit as the sign bit as shown in Table 2. Fixed point numbers may use either signed magnitude notation or complement notation. However, as explained in the previous section signed magnitude notation is not a natural notation for binary arithmetic, the complement notation is used in computers. The complement notation works well for the digital binary numbers as they are of fixed length. For the sake of simplicity, in this unit we will use an complement notation having a length of 8-bits.

For the fixed point number representation signed 1's complement and signed 2's complement notation can be used. The signed complement notation is same as the complement notation as introduced in the previous section, except that it uses a sign bit, in addition to represent magnitude. In signed 1's and 2's complement notation the positive number has the same magnitude as that of binary number with the sign bit as zero, however, the negative numbers are represented in complement form. The following example explains the process of conversion of decimal numbers to signed 1's or signed 2's complement notation.

Example 7: Represent the +73, -73, +39, -39, +127, -127 and 0 using signed 1's complement notation.

Solution: The table 6 shows the values in signed 1's complement notation of length 8 bits (S is the sign bit). Please note that even in signed 1's complement notation there are two representations for 0. The number range for 1's complement for this 8 bit representation is -127 to -0 and +0 to +127. So it can represent 2^8-1 (as two representation of 0) =255 numbers.

Number	Process	S	7 bits						
+73	Sign is 0 (positive) and 7 bit magnitude is same as binary equivalent value of 73	0	1	0	0	1	0	0	1
-73	**Take 1's complement of all the 8 bits (including sign bit) to obtain -73**	1	0	1	1	0	1	1	0
+39	Follow same process as stated for +73	0	0	1	0	0	1	1	1
-39	Follow same process as stated for -73	1	1	0	1	1	0	0	0
+127	Follow same process as stated for +73	0	1	1	1	1	1	1	1
-127	Follow same process as stated for -73	1	0	0	0	0	0	0	0
0	Follow same process as stated for +73	0	0	0	0	0	0	0	0
-0	Follow same process as stated for -73	1	1	1	1	1	1	1	1

Table 6: 8-bit Signed 1's complement notation

Example 8: Represent the +73, -73, +39, -39, +127, -127, 0 and -128 using signed 2's complement notation.

Solution: The table 7 shows the values in signed 2's complement notation of length 8 bits (S is the sign bit). Please note that in signed 2's complement notation there is a unique representations for 0, therefore, -128 can also be represented. Thus, the range of the number that can be represented using signed 2's complement notation is -128 to +127. Thus, a total of 256 numbers can be represented using signed 2's complement notation.

Number	Process	S	7 bits						
+73	Sign is 0 (positive) and 7 bit magnitude is same as binary equivalent value of 73	0	1	0	0	1	0	0	1
-73	**Take 2's complement of the number (including sign bit) to obtain -73**	1	0	1	1	0	1	1	1
+39	Follow same process as stated for +73	0	0	1	0	0	1	1	1
-39	Follow same process as stated for -73	1	1	0	1	1	0	0	1
+127	Follow same process as stated for +73	0	1	1	1	1	1	1	1
-127	Follow same process as stated for -73	1	0	0	0	0	0	0	1
0	Follow same process as stated for +73	0	0	0	0	0	0	0	0
-0	Follow same process as stated for -73	0	0	0	0	0	0	0	0
-128	-127-1 is = -128	1	0	0	0	0	0	0	0

Table 7: 8-bit Signed 2's complement notation

In general, a signed 2's number representation of *n* bits can represent numbers in the range -2^{n-1} to + $(2^{n-1}-1)$. Therefore, an 8 bit representation can represent the numbers in the range -2^{8-1} to + $(2^{8-1}-1)$; i.e. -2^7 to $+(2^7-1)$, which is -128 to +127. For a 16 bit representation this range will be -2^{15} to $+(2^{15}-1)$, which is -32768 to +32767. Please relate these ranges to range given in programming languages like C.

Signed 2's complement notation is one of the best notation to perform arithmetic on numbers. Next, we explain the process of performing arithmetic using fixed point numbers.

2.5.2 Binary Arithmetic using Complement notation

In this section, we discuss about binary arithmetic using fixed point complement notation.

Arithmetic addition: Arithmetic addition operation can be performed using any of the signed-magnitude, singed 1's complement and signed 2's complement notation.

Addition using signed-magnitude notation:

The process of addition of two numbers using signed magnitude notation will require the following steps:

Step 1: Check if the numbers have similar or different signs.

Step 2: If signs are same then just add the two numbers, otherwise identify the number having bigger magnitude (in case both numbers have same magnitude then first number may be assumed as bigger number) and subtract the smaller number from bigger number.

Step 3: If signs of the number are same then check if the result exceeds the size of number of bits of the representation, if that happens then report overflow of number. For the numbers with different signs overflow cannot occur.

Step 4: The sign of the final number is the sign of any operand, if signs are same; or the sign of the bigger number, if signs are different.

The following example explains the process of addition using signed-magnitude notation. The example, uses an 8 bit representation.

Example 9: Add the decimal numbers 75 and -80 using signed magnitude notation, assuming the 8-bit length of the notations.

Solution: The numbers are (The left most bit is the Sign bit):

Number	Signed Magnitude								Signed 1's Complement								Signed 2's Complement							
+75	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1
+80	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0
-80	1	1	0	1	0	0	0	0	1	0	1	0	1	1	1	1	1	0	1	1	0	0	0	0

Table 8: 8-bit representation of +75, +80 and -80

For the present example, signs of two numbers are different and the magnitude of -80 is higher than 75, therefore, 75 is subtracted from 80 as shown in the following table and the sign of the -80, which is minus is selected for the output. Please note that during the subtraction, carry is required to be taken as done in the decimal numbers. In binary, if a carry is taken then the number changes to two digit number as shown in the table. Please note that 10 of binary is a value 2 in decimal.

Number	Signed Magnitude							
Carry taken out from the bit	No	No	No	yes	yes	yes	yes	No
Updated bit value				0	~~10~~ 1	~~10~~ 1	~~10~~ 1	10
-80	1	1	0	~~1~~	~~0~~	~~0~~	~~0~~	~~0~~
+75	0	1	0	0	1	0	1	1
Subtraction	Sign bit	1-1	0-0	0-0	1-1	1-0	1-1	10-1
Result =-5	1	0	0	0	0	1	0	1

Table 9: 8-bit addition using Signed magnitude notation

Example 10: Add the decimal numbers 75 and 80 using signed magnitude notation, assuming the 8-bit length of the notations.

Solution: The process of addition of +75 and +80 in signed magnitude representation is shown below:

Number	Signed Magnitude							
Carry from previous bit addition	No	Yes	No	No	No	No	No	No
Carry bit value	1							
+75	0	1	0	0	1	0	1	1
+80	0	1	0	1	0	0	0	0
	1	1+1=10	0+0	0+1	1+0	0+0	1+0	1+0
Result is -27 (error)	1	0	0	1	1	0	1	1

Table 10: 8-bit addition using Signed magnitude notation having overflow

The addition of 7 bit magnitude has resulted in 8 bit output, which cannot be stored in this notation as this notation has a length of 8 bits with 1 sign bit. The last bit will be lost and you will obtain an incorrect result -27. This problem has occurred as the size of the number is fixed. This is called the overflow. You may please note that the actual addition of the 75 and 80 is 155, which is beyond the range of 8 bit signed magnitude representation, which is -127 to +127. This is why you should be careful while selecting the integral data types in a programming languages. For example, in case you have selected small unsigned integer of byte size for a variable, then you can store a only the number values in the range 0 to 255 in that variable.

Addition using signed-1's complement notation:

In signed 1's complement notation the addition process is simpler than signed-magnitude representation. An interesting fact is that in this notation you do not have to check the sign, just add the numbers, why? This is due to the fact that complement of a number as defined makes it complete, the binary digits are complement of each other and even the sign bits are complement. Therefore, the process of addition of two signed 1's complement notation just requires addition of the two numbers, irrespective of the sign. The process of addition in signed 1's complement representation will require the following steps:

Step 1: Just add the numbers, irrespective of sign.

Step 2: Now check the following conditions:

Carry in to the Sign Bit	**Carry out of the Sign bit**	**Comments**
No	No	Result is fine
Yes	Yes	Add 1 to result and it is fine
No	Yes	Overflow, incorrect result
Yes	No	Overflow, incorrect result

Table 11: The conditions of 1's complement notation, while addition

The following example demonstrates the process of addition .

Example 11: Add the decimal numbers 75 and -80 using signed 1's complement notation, assuming the 8-bit length of the notations.

Solution: The numbers are (The left most bit is the Sign bit). The Table 8 shows the values of +75 and -80 in signed 1's complement notation.

Number	Signed 1's Complement Notation							
Carry from previous bit	No	No	No	yes	yes	yes	yes	-

addition								
Updated bit value	-	-	-	1	1	1	1	-
+75	0	1	0	0	1	0	1	1
-80	1	0	1	0	1	1	1	1
Addition	Sign bit (0+1=1)	1+0 =1	0+1 =1	1+0+0 =1	1+1+1 =11	1+0+1 =10	1+1+1 =11	1+1 =10
Result (-5)	1	1	1	1	1	0	1	0
Since, the result is negative, so taking a 1's complement to verify the magnitude								
-Result=+5	0	0	0	0	0	1	0	1

Table12: 8-bit addition using Signed 1's complement notation

Example 12: Add the decimal numbers 75 and 80 using signed 1's complemt notation, assuming the 8-bit length of the notations.
Solution: The process of addition of +75 and +80 in signed magnitude representation is shown below:

Number	Carry out (9th bit)	Signed 1's Complement Notation							
Carry from previous bit addition	No	Yes	No	No	No	No	No	No	-
Updated bit value	No Carry out of the sign bit addition	Carry in to Sign Bit 1	-	-	-	-	-	-	-
+75		0	1	0	0	1	0	1	1
+80		0	1	0	1	0	0	0	0
Addition		Sign bit (0+0+1=1)	1+1 =10	0+0 =0	0+1 =1	1+0 =1	0+0 =0	1+0 =1	1+0 =1
Result (A negative number)		1	0	0	1	1	0	1	1
There is carry in the sign bit (1) and no carry out of the sign bit. Therefore, as per Table 11, there is an overflow and the result is incorrect. You can observe that the result is negative for addition of two positive numbers, which is NOT possible.									

Table 13: 8-bit addition using Signed 1's complement notation having overflow

Once again, please observe that the range of numbers in 8-bit 2's complement notation is -127 to +127, and the addition of two numbers 155 cannot be represented in 8-bits. Hence, there is an overflow.

Addition using signed-2's complement notation:

In signed 2's complement notation the addition process is simplest of these three representations. In this notation also, you do not have to check the sign, just add the numbers including the sign bit. The process of addition in signed 2's complement representation will use the following steps:
Step 1: Just add the numbers, irrespective of sign.
Step 2: Now check the following conditions:

Carry in to the Sign Bit	Carry out of the Sign bit	Comments
No	No	Result is fine
Yes	Yes	Result is fine
No	Yes	Overflow, incorrect result
Yes	No	Overflow, incorrect result

Table 14: The conditions of 2's complement notation, while addition

The following example demonstrates the process of addition using signed 2's complement notation.

Example 13: Add the decimal numbers (i) -69 -59 (ii) -69+59 (iii) +69-59 and (iv) +69=59

Solution: The numbers are (The left most bit is the Sign bit). The Table 14 shows the numbers in signed 2's complement notation. The left most bit is the sign bit

Number	Signed 2's Complement							
+69	0	1	0	0	0	1	0	1
- 69	1	0	1	1	1	0	1	1
+59	0	0	1	1	1	0	1	1
-59	1	1	0	0	0	1	0	1

Table 15: Numbers of example 13 in 2's complement notation

(i) -69-59

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry from previous bit addition	yes	Carry in to Sign bit yes	yes	yes	yes	yes	yes	yes	-
Carry for addition	1	1	1	1	1	1	1	1	-
-69		1	0	1	1	1	0	1	1
-59		1	1	0	0	0	1	0	1
Addition of bits given above		1+1+1 =11	1+0+1 =10	1+1+0 =10	1+1+0 =10	1+1+0 =10	1+0+1 =10	1+1+0 =10	1+1 =10
Result	1	1	0	0	0	0	0	0	0
There is carry in to the sign bit (1) and there is a carry out of the sign bit (1). Therefore, as per Table 14, there is NO overflow and the result is correct and equal to -128. Discard the carry out bit (the 9th bit).									

Table 16: Addition of two negative numbers without overflow

(ii) -69+59

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry from previous bit addition	No	Carry in to Sign bit No	yes	yes	yes	No	yes	yes	-
Carry for addition	-	-	1	1	1	-	1	1	-
-69		1	0	1	1	1	0	1	1
+59		0	0	1	1	1	0	1	1
Addition of bits given above		1+0 =1	1+0+0 =1	1+1+1 =11	1+1+1 =11	1+1 =10	1+0+0 =1	1+1+1 =11	1+1 =10
Result	-	1	1	1	1	0	1	1	0
There is No carry in to the sign bit and there is No carry out of the sign bit. Therefore, as per Table 14, there is NO overflow and the result is correct and equal to -10. Verify the result yourself.									

Table 17: Addition of bigger negative number and smaller positive numbers. No overflow is possible.

(iii) +69-59

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry from previous bit addition	yes	Carry in to Sign bit yes	No	No	No	yes	No	yes	-
Carry for addition	1	1				1		1	-
+69		0	1	0	0	0	1	0	1
-59		1	1	0	0	0	1	0	1
Addition of bits given above		1+0+1 =10	1+1 =10	0+0 =0	0+0 =0	1+0+0 =1	1+1 =10	1+0+0 =1	1+1 =10
Result	1	0	0	0	0	1	0	1	0
There is a carry in to the sign bit (1) and there is a carry out of the sign bit (1). Therefore, as per Table 14, there is NO overflow and the result is correct and equal to +10. Discard the carry out bit (the 9th bit). Verify the result yourself.									

Table 18: Addition of smaller negative number and bigger positive numbers. No overflow is possible.

(iv) +69+59

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry from previous bit addition	No	Carry in to Sign bit yes	yes	yes	yes	yes	yes	yes	-
Carry for addition	-	1	1	1	1	1	1	1	-
+69		0	1	0	0	0	1	0	1
+59		0	0	1	1	1	0	1	1
Addition of bits given above		1+0+0 =1	1+1+0 =10	1+0+1 =10	1+0+1 =10	1+0+1 =10	1+1+0 =10	1+0+1 =10	1+1 =10
Result	-	1	0	0	0	0	0	0	0
There is a carry in to the sign bit (1) but there is NO carry out of the sign bit. Therefore, as per Table 14, there is an *overflow* and the result is incorrect. Verify the result yourself. Overflow has occurred as the addition of the two numbers is +128, which is out of the range of numbers that can be represented using 8-bit signed 2's complement notation.									

Table 19: Addition of two positive numbers.

It may be noted that for the signed 2's complement notation, which is using 8 bits representation, is -128 to +127, which can be checked from table 16 and table 19.

Overflow formally is defined as the situation where the result of operation on two or more numbers, each of size n digits, exceeds the size n.

Overflow may cause even your correct programs to output incorrect results, therefore, is a very risky error. One of the ways of avoiding overflow in programs is to select appropriate data types and verifying the results range.

Arithmetic Subtraction: In general, a computer system uses the signed 2's complement notation, which simplifies the process of addition and subtraction as well as has a single representation for 0. You can perform subtraction by just taking the 2's complement of the number that is to be subtracted, and thereafter just adding the two numbers just like it has been shown in this section.

Multiplication and division: Multiplication and division operations using signed 2's complement notations are not straight forward. One of the simplest approach to multiply two signed 2's complement numbers is by multiplying the positive numbers and then adjusting the result based on the sign. However, this approach is time consuming as well as not used for implementation of multiplication operation. There are a number of algorithms for performing multiplication and division. One such algorithm is the Booth's algorithm. A detailed discussion on these topics is beyond the scope of this course.

In several arithmetic computations binary representation of decimal number is used for performing arithmetic operations. The next subsection briefly explains this representation.

2.5.3 Decimal Fixed Point Representation

Decimal digits can be represented in binary directly using four bits, as there are only 10 decimal digits, whereas 2^4=16 different values can be expressed using 4 bits. Thus, a BCD may be represented as 0000 (for decimal digit 0) to 1001 (for decimal digit 9). In addition, the sign can be represented using a single bit; however, it may change the format of representation. Thus, in decimal fixed point representation even sign is represented as four bits. Interestingly, the positive sign is represented using 1100 and negative sign is represented using 1101. Please note these two combinations are different from the representation of decimal digits, which is 0000 to 1001.
Example14: Represent +125 as BCD and a binary number.

+125 in BCD is given below:

Sign Digit	1	2	5
1100	0001	0010	0101

+125 in Binary:

S	7-bit magnitude						
-	64	32	16	8	4	2	1
0	1	1	1	1	1	0	1

Why is this representation needed? In several computing devices the computations are performed on binary coded decimals directly, without conversion to binary. One such device was old calculator. You may refer to further readings for more details on BCD arithmetic.

Check Your Progress 2

1) Write the BCD for the following decimal numbers:
   i) -23456
   ii) 17.89
   iii) 299

   ....

   ....

....

....

2) Compute the 1's and 2's complement of the following binary numbers. Also find the decimal equivalent of the number.

   i) 1110 0010

   ii) 0111 1110

   iii) 0000 0000

   ....

   ....

   ....

   ……………………………………………………………………………….

3) Add the following decimal numbers by converting them to 8-bit signed 2's complement notation.

   i) +56 and – 56

   ii) +65 and –75

   iii) +121 and +8

   Identify, if there is overflow.

   ....

   ………………...

   ……...

   …………………………………………………………………………………

2.6 FLOATING POINT REPRESENTATION

In most real numbers, you may use a decimal point to distinguish integer and fraction part. However, in a computer system the position of binary point is assumed in the numbers. Fixed point representation, in general, fixes the location of the point towards the right most. Thus, integer values are represented using fixed point representation. What about the real numbers. A real number can be represented using an exponential notation. This forms the basis of binary number representation called floating point representation. For example, a decimal real number 29.25 can be represented as 0.2925×10^{2} or 2925×10^{-2}.

The first part of the number is called the "mantissa" of "significand" and second part of the number is called the exponent. You may please note that the mantissa can either be integer or fraction as shown in the example; the exponent value is adjusted accordingly. In computer the mantissa and exponent both are represented as binary numbers and the location of binary point is assumed. The following example explains the binary floating point representation. This representation is IEEE 754 standard for 32-bit floating point number.

It has the following format:

Bit Positions from the left	**1**	**2 to 9**	**10 to 32**

Length of Field	1 bit	8 bits	23 bits
Purpose	To store the *Sign* bit	To store the *Exponent*	Stores the fractional *Significand* of the Number
Comment	The *Sign* bit is for the *Significand*	The exponent is stored in *biased* form with a *bias* of 127	The *Significand* is stored as a *normalized* binary number

(a) Basic details

***Exponent* (8 bits) so possible values 0 to 255.** A bias of 127 is assumed. Let the exponent be *exp*	***Significand* values (23 Bits)** Assume that *Significand* be *M*, which is 23 bit long	**The Number Represented**
For *Exponent* value (*exp*) 0	All the bits of *M* are zeros.	The number is ±0 depending on the sign bit.
	M is NOT zero (*M* may not be normalized)	The Number is $\pm 0.M \times 2^{-126}$
For *Exponent* values (*exp*) from 1 to 254	Normalized representation is used, therefore, *the first bit is assumed to be 1.*	The Number is $\pm 1.M \times 2^{exp-127}$
For *Exponent* value (*exp*) 255	All bits of *M* are zeros	The number is ±∞ depending on the sign bit
	M is NOT zero.	It does NOT represent a valid Number

(b) Single Precision 32-bit IEEE-754 Standard

Table 20: IEEE 754 Floating Point 32-bit Number Representation

The three terms in Table 20 are *fractional Significand*, *bias* and *normalized*. They are explained below:

fractional Significand: Floating point number assumes that the position of binary point is prior to the *Significand*, therefore, *Significand* is a fraction (Refer to example 15).

Bias: It is an interesting way to store signed numbers without using any sign bit. It stores the number by adding a value in the exponent. For example, a 4 bit binary number can store values 0000 to 1111, i.e. values 0 to 15. A bias of 8 will allow values -8 to +7 to be stored in this range by adding the bias. In other words exponent value -8 will be coded as (-8+8) 0, -7 will be coded as (-7+8) 1, and so on till +7, which will be coded as (+7+8) 15. But, why is biasing used for exponent? The basic reason here is that biased numbers simplify the floating point arithmetic. This is explained later with the help of an example.

Normalized: A fraction is called normalized if it starts with a bit value 1 and not with bit value 0. For example, the values .1001, .1111, .1000, .1010 are normalized, but the values .0100, .0001, .0010, .0011 are not normalized.

The following example explains the process of converting a decimal real number to a floating point number representation using IEEE-754 standard (32-bit representation). You may solve similar problems using double precision representation also, where only the size of exponent (and bias) and significand is different.

Example 15: Represent the number -29.25 using IEEE 754 (32 bit) representation as shown in Table 20.

Solution:

Step 1: Convert the number to binary

The number should first be converted to binary as follows:

Sign bit = 1 as number is negative

29 can be represented in 7 bits as 001 1101

.25 can be represented in 4 bits as .0100

Thus, 29.25 without sign is 001 1101 . 0100

Step 2: Normalize the number

Normalizing the number requires binary point to be moved before the most significant 1, it requires point to be shifted to left by 5 spaces. Thus, the normalized number now is: $0.1\ 1101\ 0100 \times 2^5$.

Step 3: Adjust the normalized number

It may be noted from Table 20(b), that in IEEE-754 representation, when the exponent is between 1 and 254, the first bit is assumed to be 1, therefore, the *Significand* whose size is 23 bits, actually represents 24 bit *Significand.* In addition, please note that as the number is assume to be $\pm 1.M \times 2^{exp\text{-}127}$, therefore, the value to be represented ($0.1\ 1101\ 0100 \times 2^5$) must be adjusted to this format by shifting the binary point one place to the right and adjusting the exponent. Thus, the adjusted number is 1.11010100×2^4.

Step 4: Compute the exponent using the bias

Finally add bias to the exponent value to obtain *exp* value of IEEE-754. In this case, *exp* = 4+127 (127 is the bias value)=131.

Step 5: Represent the final number

Represent the sign bit (S), *exp* in 8 bits and *Significand* in 23 bits, as follows:

S	*exp* of length 8 bits (value 131)	*Significand* of length 23 bits (value 1.11010100) Represented as ~~1~~. 110 1010 0
1	1 0 0 0 0 0 1 1	1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Example 16: A number using IEEE 754 (32 bit) is given below, what is the equivalent decimal value.

S	*exp* of length 8 bits	*Significand* of length 23 bits (*M*)
1	1 0 0 0 1 0 0 1	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Solution:

The number is represented as: $\pm 1.M \times 2^{exp\text{-}127}$

The sign bit states it is a negative number

M is 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Exp is 1 0 0 0 1 0 0 1 = 137 in decimal.

The number is $-1.11110000000000000000000 \times 2^{137\text{-}127}$

= $-1.11110000000000000000000 \times 2^{10}$

= -11111000000.0000000000000

= -11111000000

= -1984 in decimal

In floating point numbers a term *precision* is very important. What is precision? The precision defines the correctness of representation. For example, suppose you *just use 2 decimal digits* in a fractional decimal numbers, then you can represent the numbers 0.10, 0.11, 0.99 etc precisely. The number 0.985 may be either truncated to 0.98 or rounded off to 0.99. This introduces an error in number, which is due to the fact that the size of *Significand* is limited. For scientific computations such errors may lead to failure. Therefore, IEEE-754 defines many different precision of numbers, few such popular precisions are single precession IEEE-754 number, which is a 32-bit representation, explained above; **IEEE-754 double precision number**, which is a 64 bit representation with 1 sign bit, 11 bit exponent and 52 bit *Significand*; and IEEE-754 quadruple precision number, which is a 128 bit representation with 1 sign bit, 15 bit exponent and 112 bit *Significand*. It may be noted that in programming languages you use data types *float* and *double*, which corresponds to the IEEE-754 single and double representation respectively.

Finally, what is the range of the numbers that can be represented using the IEEE-754 representation? As stated in Table 20, the minimum exponent value for a normalized number is 1 and maximum is 254. Therefore, the minimum (negative) number will be:

S	*exp* of length 8 bits	*Significand* of length 23 bits (*M*)
1	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

This will be equal to $\pm 1.M \times 2^{exp-127}$

$$= -1.000\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{1-127}$$
$$= -1 \times 2^{-126}$$

The maximum (positive) number will be:

S	*exp* of length 8 bits	*Significand* of length 23 bits (*M*)
1	1 1 1 1 1 1 1 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

This will be equal to $\pm 1.M \times 2^{exp-127}$

$$= +1.111\ 1111\ 1111\ 1111\ 1111\ 1111 \times 2^{254-127}$$
$$= +(1.111\ 1111\ 1111\ 1111\ 1111\ 1111 + 0.000\ 0000\ 0000\ 0000\ 0000\ 0001 - 0.000\ 0000\ 0000\ 0000\ 0000\ 0001) \times 2^{127}$$
$$= +(10.\ 000\ 0000\ 0000\ 0000\ 0000\ 0000 - 0.000\ 0000\ 0000\ 0000\ 0000\ 0001) \times 2^{127}$$
$$= +(2 - 1 \times 2^{-23}) \times 2^{127}$$

You may please note that IEEE-754 has a representation for *0 and infinite*.

Arithmetic Using Floating Point Numbers:

As you have noticed that addition and subtraction using 2's complement notation was direct, but addition and subtraction of floating point number requires several steps. These steps are explained with the help of the following example.

Example 17: Add the following floating point numbers

Equivalent Numbers		**IEEE 754 32 bit representation**		
Decimal	*Binary*	*S*	*exp*	*Significand* (*M*)
-7	$-1.11 \times 2^{129-127}$ $= -1.11 \times 2^{2}$ $= -111.0$	1	1000 0001	110 0000 0000 0000 0000
+24	$+1.1 \times 2^{131-127}$ $= +1.1 \times 2^{4}$ $= +11000.0$	0	1000 0011	100 0000 0000 0000 0000

Solution:

Step 1: Find the difference in exponents of the numbers

1000 0011 - 1000 0001 = 0000 0010 = 2 in decimal

Step 2: Align the *Significand* of the smaller number by denormalizing it

Shift the smaller number to right by the difference of exponents as shown:

	The value of 1.*M*
Significand of First Number (Smaller)	1.110 0000 0000 0000 0000
Shift it to right twice (denormalized)	0.011 1000 0000 0000 0000

Step 3: Check the sign of the two numbers, if same add else subtract smaller number from bigger number.

The signs are different; therefore, subtract smaller number from larger number

	The value of 1.*M*
Significand of Second Number (Larger)	1.100 0000 0000 0000 0000
Denormalized first number (smaller)	0.011 1000 0000 0000 0000
Result of Subtraction	1.000 1000 0000 0000 0000

Step 4: Select the sign and exponent of the bigger number as sign and exponent of the result and Normalize the *Significand* by adjusting the exponent
The result is shown below. Please note that in this case, there is no need to normalize the result as it is already normalized.

Result of addition operation (verification)		**IEEE 754 32 bit representation**		
Decimal	*Binary*	*S*	*exp*	*1.M*
+17	$+1.0001 \times 2^{131-127}$ $= +1.0001 \times 2^4$ $= +10001.0$	0	1000 0011	1.000 1000 0000 0000 0000

Likewise, subtraction operation can be performed.

Multiplication and division operations in floating point number require multiplication or division of significands as well as addition or subtraction of exponents. In addition, these operations may require normalizing the result. The following example shows the multiplication of two floating point numbers.

Example 18: Multiply the following floating point numbers

Equivalent Numbers		**IEEE 754 32 bit representation**		
Decimal	*Binary*	*S*	*exp*	*Significand (M)*
-7	- 111.0	1	1000 0001	110 0000 0000 0000 0000
+24	+11000.0	0	1000 0011	100 0000 0000 0000 0000

Solution:

Step 1: Multiply the *Significand* values of the two numbers and truncate to 23 bits (plus one assumed bit)

	The value of 1.*M*
Significand of First Number	1.110 0000 0000 0000 0000
Significand of Second Number	1.100 1000 0000 0000 0000
Significand after multiplication	10.101 0000 0000 0000 0000

Step 2: If multiplication, add the exponents and subtract the bias as both the numbers have biased exponents; if division, subtract the exponent of divisor from the exponent of dividend and add the bias as it get canceled in subtraction.

Exponent of First Number	1000 0001
Exponent of Second Number	1000 0011
Multiplication operation, so add and subtract bias *-127* *The new exponent*	1 0000 0100 -0111 1111 1000 0101

Step 3: Check the sign of the two numbers, if same result has + sign else - sign
The signs are different; therefore, result has negative sign. Also, normalize the *Significand* of the result

Result of Multiplication	**IEEE 754 32 bit representation**

operation				
Decimal	*Binary*	*S*	*exp*	*1.M*
Normalize the result		1	1000 0101	10.10 1000 0000 0000 0000
Normalized result		1	1000 0110	1.010 1000 0000 0000 0000
-168	$-1.0101\times2^{134-127}$ $= -1.0101\times2^{7}$ $= -10101000.0$	1	1000 0110	1.010 1000 0000 0000 0000

Likewise, division operation can be performed. You may refer to further readings for finding more details on floating point numbers and arithmetic.

2.7 ERROR DETECTION AND CORRECTION CODES

In the previous sections you have gone through various binary codes and representations. A computer works on these binary numbers and during the operations of the computer data is transferred from a source to one or more destinations. During this process of transmission of data, there is a possibility of transmission errors. The purpose of error detection and correction codes is to identify those data transmission errors and correct the data, as far as possible. As the data in computer consists of binary digits, therefore, an error in a bit can result in change of its value from 0 to 1 or vice versa. This section explains one error detection code called Parity and one error detection and correction code called Hamming Error correction code.

Parity bit: The purpose of a parity bit is to detect an error in a group of bits. But how does it perform the task of checking error? It is explained with the help of following process:

Steps	Source Side	Destination Side
1	A parity bit is generated at source, which ensures that: Number of 1's in sources data and source parity bit is odd (called Odd parity) OR Number of 1's in sources data and source parity bit is even (called Even parity)	
2	The source data and source parity bits are sent to the designation.	
3		The source data and source parity bits are received at the designation and a destination parity bit is generated using only the data received (not source parity) by using the same process i.e. Even of Odd parity used at source.
4		Source parity bit and destination parity bit are compared. If they are same, then no error in data is detected, else either the data or parity bit is in error, which is reported.

Table 21: Error detection using parity bit

Example 19 explains the process as given above.

Example 19: 7-bit data 010 1001 is sent from a source, such as CPU register, to a destination, such as RAM. The data is received at the destination as 010 1000 having error in one bit. How does this error be detected by parity bit?

Solution:

Steps	*The Process*
Step 1: At Source	Data to be sent: 010 1001 Odd Parity bit is computed as: The data has 3 bits having value 1. So odd parity bit =*0*
Step 2: At Source	The source parity + source data is sent as: *0* 010 1001
Step 3: At Destination	As per the statement of the example the data is received as: *0* 010 1000 Source Parity = *0* is received correctly as error is in one bit only Destination parity is computed on data received, which is 010 1000. It has 2 bits as 1, therefore, Odd parity bit at destination=*1*
Step 4: At Destination	Source parity bit (*0*) ≠ Destination parity bit(*1*) ERROR in data

It may be noted that parity bit can detect errors in case 1 bit is in error. In case 2 bits are in error, then it will fail to detect the error.

Hamming Error-Correcting Code: The Hamming code was conceptualized by Richard Hamming at Bell Laboratories. This code is used to identify and correct the error in 1 bit. Thus, unlike parity bit, which just identifies the existence of error, this code also identifies the bit that is in error. The idea of Hamming's code is to divide the data bits into a number of groups; and using the parity bit to identify, which groups are in error; and based on the groups in error, identify the bit which has caused the error. Thus, the grouping process has to be very special, which is explained below:

How to Group data bits? Before grouping, you may assume the placement of data and parity bits using the following considerations.

A bit position that is exact power of 2 will be used for storing parity bit. For example, $2^0=1$, that is 1st bit position will be used to store parity bit, likewise $2^1=2$, $2^2=4$, and $2^3=8$, i.e. 2nd , 4th and 8th bit positions will also be used to store parity bit. Thus, you have now 7 bit data and 4 parity bits, so a total of 11 bit positions. (*p* indicates parity bit and *d* indicates data bit)

Bit Position	12	11	10	9	8	7	6	5	4	3	2	1
Stores	*d8*	*d7*	*d6*	*d5*	***p*4**	*d*4	*d*3	*d*2	***p*3**	*d*1	***p*2**	***p*1**
For grouping the data bit number is used to identify the parity bit to which data should be member of												
Bit position 12 (8+4) contains (d8)					8				4		-	-
Bit position 11 (8+2+1) contains (d7)					8				-		2	1
Bit position 10(8+2) contains (d6)					8				-		2	-
Bit position 9(8+1) contains (d5)					8				-		-	1
Bit position 8 contains (p4)					*p*4							
Bit position 7(4+2+1) contains (d4)					-				4		2	1
Bit position 6(4+2)					-				4		2	-

contains (d3)												
Bit position 5(4+1) contains (d2)					-				4		-	1
Bit position 4 contains (p3)									*p3*			
Bit position 3(2+1) contains (d1)					-				-		2	1
Bit position 2 contains (p2)											*p2*	
Bit position 1 contains (p1)												*p1*

Table 22: Placement of data and parity bits for Hamming's error detection and correction code

Groups for parity bits: The groups are made for each on the basis of bit positions, on the basis of above Table. A bit position, which includes a parity bit position is included in the group of that parity bit. For example, the bit at bit position 12 will be included in group of parity bit *p*4 and *p*3; similarly, bit position 7 will be included in group of parity bit *p*3, *p*2 and *p*1. But why these grouping? You may please note that each data bit is part of unique combination of groups, so if it is in error, it will cause errors in all those groups to which it is a part of. Thus, by identifying all the groups, which has parity mismatch, will identify the bit which is in error. The following table shows these groups for 8 bit data.

Group for Parity bits	Bit positions and data bit
*p*4	*Bit position 12 data bit d8, Bit position 11 data bit d7, Bit position 10 data bit d6 and Bit position 9 data bit d5*
*p*3	*Bit position 12 data bit d8, Bit position 7 data bit d4, Bit position 6 data bit d3 and Bit position 5 data bit d2*
*p*2	*Bit position 11 data bit d7, Bit position 10 data bit d6, Bit position 7data bit d4, Bit position 6 data bit d3 and Bit position 3data bit d1*
p1	*Bit position 11 data bit d7, Bit position 9 data bit d5, Bit position 7data bit d4, Bit position 5 data bit d2 and Bit position 3data bit d1*

Therefore, the parity bits will be generated using the following data bits:

Parity bit	Compute Odd parity of Data bits
*p*4	*d8, d7, d6 and d5*
*p*3	*d8, d4, d3 and d2*
*p*2	*d7, d6, d4, d3 and d1*
p1	*d7, d5, d4, d2 and d1*

So, how the data bit in error be recognised? It is illustrated with the help of following example

Example 20: 8-bit data 1010 1001 is sent from a source to a destination. The data is received at the destination as 1000 1001 having error in only one bit. How does this error be detected and corrected by Hamming's error detection and correction code?
Solution:
Step 1: Place the bits as shown in Table 22 and generate parity bits at the source, for example, the odd parity bit *p4* is computed using *d8, d7, d6* and *d5* (shown as shaded cells in the following table). Their values are 1, 0, 1, 0 as shown in the table, as there are only two bits containing 1, therefore, the odd parity value for *p4* is 1. Likewise compute the other parity bits as shown in Table 23.
Step 2: Data and the associated parity bits in the sequence as shown below are sent to the destination, where once again parity bits are computed for the received data.

Step 3: Compare the source parity bits and destination parity bits as shown in Table 23. Please note when two parity bit match, a 0 is put in the compare word else a 1 is put. The magnitude of comparison word, indicates the bit position that is in error.

Step 4: If there is an error, then the data at bit position that is in error is complemented.

Step 5: The data is used at the destination after omitting the parity bits.

Bit Position	12	11	10	9	8	7	6	5	4	3	2	1
Stores	*d8*	*d7*	*d6*	*d5*	***p4***	*d4*	*d3*	*d2*	***p3***	*d1*	***p2***	***p1***
Data Bits	1	0	1	0		1	0	0		1		
Compute Odd parity bit p4 using d8, d7, d6 and d5	▨	▨	▨	▨	1							
Compute Odd parity bit p3 using d8, d4, d3 and d2	▨					▨	▨	▨	1			
Compute Odd parity bit p2 using d7, d6, d4, d3 and d1		▨	▨			▨	▨			▨	0	
Compute Odd parity bit p1 using d7, d5, d4, d2 and d1		▨		▨		▨		▨		▨		1
Data and Parity bits at Source	1	0	1	0	1	1	0	0	1	1	0	1
Data is sent to the destination, where data is received with 1 bit in error (given), therefore all the source parity bits are received without any error												
Data received at destination including parity bits	1	0	0	0	1	1	0	0	1	1	0	1
***Step 2**: Compute the parity bits using the data received at the destination*												
Data Bits Received	1	0	0	0		1	0	0		1		
Compute Odd parity bit p4 using d8, d7, d6 and d5	▨	▨	▨	▨	0							
Compute Odd parity bit p3 using d8, d4, d3 and d2	▨					▨	▨	▨	1			
Compute Odd parity bit p2 using d7, d6, d4, d3 and d1		▨	▨			▨	▨			▨	1	
Compute Odd parity bit p1 using d7, d5, d4, d2 and d1		▨		▨		▨		▨		▨		1
***Step 3:** Compare the source parity bits and destination parity bits.*												
Source Parity bits					1				1		0	1
Destination Parity bits					0				1		1	1
Parity Comparison word (0 if source and destination parity match else 1)					1				0		1	0
The comparison word is 1010 = 10 in decimal, i.e. bit position 10 is in error. The error in this bit can be corrected by complementing the bit position 10.												
Corrected Data	1	0	1	0		1	0	0		1		

Table 23: Example of Hamming's error detection and correction code

It may be noted in Table 23 that the value of comparison word 0000 would mean that there is no error in transmission of data. In addition, the values 1000, 0100, 0010 and 0001 would mean that one bit error has occurred in the transmission of source parity bits *p4, p3, p2* and *p1* respectively. Thus, no change would be needed in the received data bits at the destination in such cases.

It may please be noted that Hamming's code presented in this section can detect and correct errors in a single bit ONLY. It will not work, in case two or more bits are in error. One final question is about the size of the code needed to correct single bit error. The size will be dependent on the size of data. A simple rule is that the size of code and the data should be less than the possible bit positions that can be flagged by the comparison word. If the data to be transmitted is of size *D* bits and *P* is the number of parity bits needed for the given Hamming's code, then size of the code is the smallest value of *P*, which satisfies that following equation:

$$D + P < 2^P$$

For example, for a *D*=4 bits, the value of *P* would be 3 as:

$$4 + 3 < 2^3 \quad \text{as } 7<8$$

and for a *D*=8 bits, the value of *P* would be 4 as:

$$8 + 4 < 2^4 \quad \text{as } 12<16$$

Check Your Progress 3

1) Represent the following numbers using the IEEE-754 32bit standard:

   i) 39.125
   ii) -0.000011000_2

2) Compute the Odd and Even parity bits for the following data:

   i) 0111110
   ii) 0110000
   iii) 1110111
   iv) 1001100

   ....

   ....

3) A 4 bit data 1011 is received at the destination as 1111, assuming single bit is in error, illustrate how Hamming's single error correction code will detect and correct the error

   ....

   ....

   ....

2.7 SUMMARY

This Unit has introduced you to the basic aspects of data representation. It introduces the character representing including ASCII and Unicode. In addition, the Unit explains number conversion and fixed point representation of binary number. The Unit also highlights the arithmetic operations. This was followed by a detailed discussion on the floating point numbers. Though only IEEE 754 32-bit single precision numbers are explained, however, the logic discussed is applicable to double precision numbers too. The Unit finally introduces you to error detection code -parity bit and error detection and correction code. You must practice the data conversions and these codes as they would be useful, when you deal with binary numbers.

You should refer to the further readings for more detailed information on these topics. You are advised to take the help of further readings, Massive Open Online Courses (MOOCs), and other online resources as Computer Science is a dynamic area.

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) i) 11100.01101_2 to Octal and Hexadecimal

Binary Number	0	1	1	1	0	0	.	0	1	1	0	1	0
Grouping Directions	←							→					
Grouped (- replaced by 0)	0 1 1			1 0 0			.	0 1 1			0 1 0		
Binary place values	4	2	1	4	2	1	.	4	2	1	4	2	1
Equivalent Octal Digit	0+2+1=3			4+0+0=4			.	0+2+1=3			0+2+0=2		
Octal Number	**3**			**4**			**.**	**3**			**2**		

Binary Number	0	0	0	1	1	1	0	0	.	0	1	1	0	1	0	0	0
Grouping Directions	←								.	→							
Grouped	0 0 0 1				1 1 0 0				.	0 1 1 0				1 0 0 0			
Binary place values	8	4	2	1	8	4	2	1	.	8	4	2	1	8	4	2	1
Equivalent Hexadecimal Digit	0+0+0+1=1				8+4+0+0=C				.	0+4+2+0=6				8+0+0+0=8			
Hexadecimal Number	**1**				**C**				**.**	**6**				**8**			

ii) 1101101010_2 to Octal and Hexadecimal

Binary Number	0	0	1	1	0	1	1	0	1	0	1	0
Grouping Directions	←											
Grouped (- replaced by 0)	0 0 1			1 0 1			1 0 1			0 1 0		
Binary place values	4	2	1	4	2	1	4	2	1	4	2	1
Equivalent Octal Digit	0+0+1=1			4+0+1=5			4+0+1=5			0+2+0=2		
Octal Number	**1**			**5**			**5**			**2**		

Binary Number	0	0	1	1	0	1	1	0	1	0	1	0
Grouping Directions	←											
Grouped	0 0 1 1				0 1 1 0				1 0 1 0			
Binary place values	8	4	2	1	8	4	2	1	8	4	2	1
Equivalent Hexadecimal Digit	0+0+2+1=3				0+4+2+0=6				8+0+2+0=A			
Hexadecimal Number	**3**				**6**				**A**			

2) i) 119_{10} to binary

The place value	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
***N* = 119**	119-**64**=55; 55-**32**=23; 23-**16**=7; 7-**4**=3; 3-**2**=1; 1-**1**=0						
Equivalent Binary	**1**	**1**	**1**	**0**	**1**	**1**	**1**

ii) 19.125_{10}

The place	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}

value	=16	=8	=4	=2	=1		=0.5	=0.25	=0.125
***N* = 19.125**	19-**16**=3; 3-**2**=1; 1-**1**=0; and 2^{-3} =**0.125**								
Equivalent Binary	**1**	**0**	**0**	**1**	**1**	**.**	**0**	**0**	**1**

iii) 325_{10}

The place value	2^8 =256	2^7 =128	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
***N* = 325**	325-**256**=69-**64**=5; 5-**4**=1; 1-**1**=0								
Equivalent Binary	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**

3 i) 119_{10}

The place value	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
Equivalent Binary	**1**	**1**	**1**	**0**	**1**	**1**	**1**
Equivalent Octal	**1**	**6**			**7**		
Equivalent Hexadecimal	**7**				**7**		

ii) 19.125_{10}

The place value	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1	.	2^{-1} =0.5	2^{-2} =0.25	2^{-3} =0.125
Equivalent Binary	**1**	**0**	**0**	**1**	**1**	**.**	**0**	**0**	**1**
Equivalent Octal	**2**		**3**			**.**	**1**		
Equivalent Hexadecimal	**1**	**3**				**.**	**0010=2**		

iii) 325_{10}

The place value	2^8 =256	2^7 =128	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
Equivalent Binary	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**
Equivalent Octal	**5**			**0**			**5**		
Equivalent Hexadecimal	**1**	**4**				**5**			

Check Your Progress 2

Check Your Progress 2

1) i) -23456 to BCD

Sign Digit	2	3	4	5	6
1101	0010	0011	0100	0101	0110

ii) 17.89

Sign Digit	1	7	.	8	9
1100	0001	0111	.	1000	1001

iii) 299

Sign Digit	2	9	9
1100	0010	1001	1001

2)

Deci	The Number	Signed 1's Complement	Signed 2's Complement

mal																								
-30	1	1	1	0	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	1	1	1	1	0
+126	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

3) i) +56 and – 56

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry for addition	1	1	1	1	1				-
+56		0	0	1	1	1	0	0	0
-56		1	1	0	0	1	0	0	0
Addition of bits given above		1+0+1 =10	1+0+1 =10	1+1+0 =10	1+1+0 =10	1+1 =10	0+0 =0	0+0 =0	0+0 =0
Result	~~1~~	0	0	0	0	0	0	0	0

There is a carry in to the sign bit (1) and there is a carry out of the sign bit (1). Therefore, as per Table 14, there is NO overflow and the result is correct and equal to 0. Discard the carry out bit (the 9th bit). Verify the result yourself.

ii) +65 and –75

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry for addition								1	-
+65		0	1	0	0	0	0	0	1
-75		1	0	1	1	0	1	0	1
Addition of bits given above		0+1 =1	1+0 =1	0+1 =1	0+1 =1	0+0 =0	0+1 =1	1+0+0 =1	1+1 =10
Result		1	1	1	1	0	1	1	0

There is No carry in to the sign bit and there is No carry out of the sign bit. Therefore, as per Table 14, there is NO overflow and the result is correct and equal to -10. Verify the result yourself.

iii) +121 and +8

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry for addition		1	1	1	1				-
+121		0	1	1	1	1	0	0	1
+8		0	0	0	0	1	0	0	0
Addition of bits given above		1+0+0 =1	1+1+0 =10	1+1+0 =10	1+1+0 =10	1+1 =10	0+0 =0	0+0 =0	0+1 =1
Result		1	0	0	0	0	0	0	1

There is a carry in to the sign bit (1) and there is NO carry out of the sign bit. Therefore, as per Table 14, there is OVERFLOW and the result is incorrect.

Check Your Progress 3

1) i) 39.125

Step 1: Convert the number to binary

Sign bit = 0 as number is positive

39 can be represented in 7 bits as 010 0111

.125 can be represented in 4 bits as .0010

Thus, 39.125 without sign is 0100111.0010

Step 2: Normalize the number

Normalizing the number requires binary point to be moved before the most significant 1, it requires point to be shifted to left by 6 spaces. Thus, the normalized number now is: 0.1001110010×2^6.

Step 3: Adjust the normalized number

The number is assume to be $\pm1.M\times2^{exp-127}$, therefore, the value to be represented (0.1001110010×2^6) must be adjusted. The adjusted number is 1.001110010×2^5.

Step 4: Compute the exponent using the bias

Add bias to the exponent value.

exp = 5+127 (127 is the bias value)=132.

Step 5: Represent the final number

Represent the sign bit (S), *exp* in 8 bits and *Significand* in 23 bits, as follows:

S	*exp* of length 8 bits (value 132)	*Significand* of length 23 bits (value *1.001110010*) Represented as ~~1~~.*001110010*
0	1000 0100	001 1100 1000 0000 0000 0000

ii) -0.000011000_2

Sign bit = 1 as number is negative

The number is of the format $\pm1.M\times2^{exp-127}$,

1.1000×2^{-5}

$exp = -5+127=122$

S	*exp* of length 8 bits (value 122)	*Significand* of length 23 bits (value *1.1000*) Represented as ~~1~~.*1000*
0	0111 1010	100 0000 0000 0000 0000 0000

2)

The Number	Even Parity	Odd Parity
0111110	1	0
0110000	0	1
1110111	0	1
1001100	1	0

3) The size of data (*D*)=4 bits, the value of *P* would be 3 as:

$4 + 3 < 2^3$ as 7<8

The bit positions for this code would be:

Bit Position	7	6	5	4	3	2	1
Stores	*d*4	*d*3	*d*2	***p*3**	*d*1	***p*2**	***p*1**
For grouping the data bit number is used to identify the parity bit to which data should be member of							
Bit position 7(4+2+1) contains (d4)				4		2	1
Bit position 6(4+2) contains (d3)				4		2	-
Bit position 5(4+1) contains (d2)				4		-	1
Bit position 4 contains (p3)				*p3*			
Bit position 3(2+1) contains (d1)				-		2	1
Bit position 2 contains (p2)						*p2*	
Bit position 1 contains (p1)							*p1*

Parity bit	Compute Odd parity of Data bits
*p*3	*d4, d3 and d2*
*p*2	*d4, d3 and d1*
p1	*d4, d2 and d1*

Bit Position	7	6	5	4	3	2	1
Stores	*d*4	*d*3	*d*2	***p*3**	*d*1	***p*2**	***p*1**
Data Bits	**1**	**0**	**1**		**1**		
Compute Odd parity bit p3 using d4, d3 and d2	▨	▨	▨	1			
Compute Odd parity bit p2 using d4, d3 and d1	▨	▨			▨	1	
Compute Odd parity bit p1 using d4, d2 and d1	▨		▨		▨		0
Data and Parity bits at Source	1	0	1	1	1	1	0
Data received at destination including parity bits	1	1	1	1	1	1	0
Data Bits Received	1	1	1		1		
Compute Odd parity bit p3 using d4, d3 and d2	▨	▨	▨	0			
Compute Odd parity bit p2 using d4, d3 and d1	▨	▨			▨	0	
Compute Odd parity bit p1 using d4, d2 and d1	▨		▨		▨		0
Source Parity bits				1		1	0
Destination Parity bits				0		0	0
Parity Comparison word (0 if source and destination parity match else 1)				**1**		**1**	**0**
Location 6 is in error, which is decimal equivalent of 110 the comparison word. So Corrected Data	1	0	1		1		