UNIT 2 DATA REPRESENTATION

2.0   INTRODUCTION

In the previous Unit, you have been introduced to the basic configuration of the Computer system, its components and working. The concept of instructions and their execution was also explained. In this Unit, we will describe various types of binary notations that are used in contemporary computers for storage and processing of data. As far as instructions and their execution is concerned it will be discussed in detailed in the later blocks.

The Computer System is based on the binary system; therefore, we will be devoting this complete unit to the concepts of binary Data Representation in the Computer System. This unit will re-introduce you to the number system concepts. The number systems defined in this Unit include the Binary, Octal, and Hexadecimal notations. In addition, details of various number representations such as floating-point representation, BCD representation and character-based representations have been described in this Unit. Finally the Error detection and correction codes have been described in the Unit.

2.1   OBJECTIVES

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

*   Use binary, octal and hexadecimal numbers;
*   Convert decimal numbers to other systems and vice versa;
*   Describe the character representation in computers;
*   Create fixed and floating point number formats;
*   Demonstrate use of fixed and floating point numbers in performing arithmetic operations; and
*   Describe the data error checking mechanism and error detection and correction codes.

2.2   DATA REPRESENTATION

The basic nature of a Computer is as an information transformer. Thus, a computer must be able to take input, process it and produce output. The key questions here are:

How is the Information represented in a computer?

Well, it is in the form of **Binary Digit** popularly called **Bit**.

How is the input and output presented in a form that is understood by us?

One of the minimum requirements in this case may be to have a representation for characters. Thus, a mechanism that fulfils such requirement is needed. In Computers information is represented in digital form, therefore, to represent characters in computer we need codes. Some common character codes are ASCII, EBCDIC, ISCII etc. These character codes are discussed in the subsequent sections.

How are the arithmetic calculations performed through these bits?

We need to represent numbers in binary and should be able to perform operations on these numbers.

Let us try to answer these questions, in the following sections. Let us first recapitulate some of the age-old concepts of the number system.

2.3   NUMBER SYSTEMS: A LOOK BACK

Number system is used to represent information in quantitative form. Some of the common number systems are binary, octal, decimal and hexadecimal.

A number system of base (also called radix) r is a system, which has r distinct symbols for r digits. A string of these symbolic digits represents a number. To determine the value that a number represents, we multiply the number by its place value that is an integer power of r depending on the place it is located and then find the sum of weighted digits.

Decimal Numbers: Decimal number system has ten digits represented by 0,1,2,3,4,5,6,7,8 and 9. Any decimal number can be represented as a string of these digits and since there are ten decimal digits, therefore, the base or radix of this system is 10.

Thus, a string of number 234.5 can be represented as:

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

Binary Numbers: In binary numbers we have two digits 0 and 1 and they can also be represented, as a string of these two-digits called bits. The base of binary number system is 2.

For example, 101010 is a valid binary number.

Decimal equivalent of a binary number:

For converting the value of binary numbers to decimal equivalent we have to find its value, which is found by multiplying a digit by its place value. For example, binary number 101010 is equivalent to:

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$= 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$$
$$= 32 + 8 + 2$$
$$= 42 \text{ in decimal.}$$

Octal Numbers: An octal system has eight digits represented as 0,1,2,3,4,5,6,7. For finding equivalent decimal number of an octal number one has to find the quantity of the octal number which is again calculated as:

(Please note the subscript 8 indicates it is an octal number, similarly, a subscript 2 will indicate binary, 10 will indicate decimal and H will indicate Hexadecimal number, in case no subscript is specified then number should be treated as decimal number or else whatever number system is specified before it.)

Decimal equivalent of Octal Number:

$(23.4)_8$
$= 2 \times 8^1 + 3 \times 8^0 + 4 \times 8^{-1}$
$= 2 \times 8 + 3 \times 1 + 4 \times 1/8$
$= 16 + 3 + 0.5$
$= (19.5)_{10}$

Hexadecimal Numbers: The hexadecimal system has 16 digits, which are represented as 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. A number $(F2)_H$ is equivalent to

$F \times 16^1 + 2 \times 16^0$
$= (15 \times 16) + 2$ // (As F is equivalent to 15 for decimal)
$= 240 + 2$
$= (242)_{10}$

Conversion of Decimal Number to Binary Number: For converting a decimal number to binary number, the integer and fractional part are handled separately. Let us explain it with the help of an example:

Example 1: Convert the decimal number 43.125 to binary number.

Solution:

Integer Part = 43	Fraction 0.125
On dividing the quotient of integer part repeatedly by 2 and separating the remainder till we get 0 as the quotient	On multiplying the fraction repeatedly and separating the integer as you get it till you have all zeros in fraction

Integer Part	Quotient on division by 2	Remainder on division by 2
43	21	1
21	10	1
10	05	0
05	02	1
02	01	0
01	00	1

Read

Please note in the figure above that:

- The equivalent binary to the Integer part of the number is $(101011)_2$

- You will get the Integer part of the number, if you READ the remainder in the direction of the Arrow.

Fraction	On Multiplication by 2	Integer part after Multiplication	Read
0.125	0.250	0	
0.250	0.500	0	↓
0.500	1.000	1	

Please note in the figure above that:

- The equivalent binary to the Fractional part of the number is 001.

- You will get the fractional part of the number, if you READ the Integer part of the number in the direction of the Arrow.

Thus, the number $(101011.001)_2$ is equivalent to $(43.125)_{10}$.

You can cross check it as follows:

$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$
$= 32 + 0 + 8 + 0 + 2 + 1 + 0 + 0 + 1/8$

$= (43.125)_{10}$

One easy direct method in Decimal to binary conversion for integer part is to first write the place values as:

2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

Step 1:   Take the integer part e.g. 43, find the next lower or equal binary place value number, in this example it is 32. Place 1 at 32.
Step 2:   Subtract the place value from the number, in this case subtract 32 from 43, which is 11.
Step 3:   Repeat the two steps above till you get 0 at step 2.
Step 4:   On getting a 0 put 0 at all other place values.

These steps are shown as:

32	16	8	4	2	1	
32	16	8	4	2	1	
1	-	-	-	-	-1	43 -32 =11
1	-	1	-	-	-	11- 8 = 3
1	-	1	-	1	-	3-2 = 1
1	-	1	-	1	1	1-1= 0
1	0	1	0	1	1	is the required number.

You can extend this logic to fractional part also but in reverse order. Try this method with several numbers. It is fast and you will soon be accustomed to it and can do the whole operation in single iteration.

Conversion of Binary to Octal and Hexadecimal: The rules for these conversions are straightforward. For converting binary to octal, the binary number is divided into

groups of three, which are then combined by place value to generate equivalent octal. For example the binary number 1101011.00101 can be converted to Octal as:

1	101	011	.	001	01
001	101	011	.	001	010
1	5	3	.	1	2

(Please note the number is unchanged even though we have added 0 to complete the grouping. Also note the style of grouping before and after decimal. We count three numbers from right to left while after the decimal from left to right.)

Thus, the octal number equivalent to the binary number 1101011.00101 is $(153.12)_8$.

Similarly by grouping four binary digits and finding equivalent hexadecimal digits for it can make the hexadecimal conversion. For example the same number will be equivalent to $(6B.28)_H$.

110	1011	.	0010	1	
0110	1011	.	0010	1000	
6	11	.	2	8	
6	B	.	2	8	(11 in hexadecimal is B)
Thus equivalent hexadecimal number is $(6B.28)_H$					

Conversely, we can conclude that a hexadecimal digit can be broken down into a string of binary having 4 places and an octal can be broken down into string of binary having 3 place values. Figure 1 gives the binary equivalents of octal and hexadecimal numbers.

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

Figure 1: Binary equivalent of octal and hexadecimal digits

Check Your Progress 1

1) Convert the following binary numbers to decimal.

    i)   1100.1101

    ii) 10101010

    ....

    ....

    ....

    ....

2) Convert the following decimal numbers to binary.

    i)   23

    ii) 49.25

    iii) 892

    ....

    ....

    ....

    ....

3) Convert the numbers given in question 2 to hexadecimal from decimal or from the binary.

    ....

    ....

    ....

    ....

2.4   DECIMAL REPRESENTATION IN COMPUTERS

The binary number system is most natural for computer because of the two stable states of its components. But, unfortunately, this is not a very natural system for us as we work with decimal number system. So, how does the computer perform the arithmetic? One solution that is followed in most of the computers is to convert all input values to binary. Then the computer performs arithmetic operations and finally converts the results back to the decimal number so that we can interpret it easily. Is there any alternative to this scheme? Yes, there exists an alternative way of performing computation in decimal form but it requires that the decimal numbers should be coded suitably before performing these computations. Normally, the decimal digits are coded in 7-8 bits as alphanumeric characters but for the purpose of arithmetic calculations the decimal digits are treated as four bit binary code.
As we know 2 binary bits can represent $2^2 = 4$ different combinations, 3 bits can represent $2^3 = 8$ combinations, and similarly, 4 bits can represent $2^4 = 16$ combinations. To represent decimal digits into binary form we require 10 combinations, but we need to have a 4-digit code. One such simple representation may be to use first ten binary combinations to represent the ten decimal digits. These are popularly known as Binary Coded Decimals (BCD). Figure 2 shows the binary coded decimal numbers.

Decimal	Binary Coded Decimal
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0001 0000
11	0001 0001
12	0001 0010
13	0001 0011
..	....
20	0010 0000
..	....
30	0011 0000

Figure 2: Binary Coded Decimals (BCD)

Let us represent 43.125 in BCD.

4	3	.	1	2	5
0100	0011	.	0001	0010	0101

Compare the equivalent BCD with equivalent binary value. Both are different.

2.5   ALPHANUMERIC REPRESENTATION

But what about alphabets and special characters like +, -, * etc.? How do we represent these in a computer? A set containing alphabets (in both cases), the decimal digits (10 in number) and special characters (roughly 10-15 in numbers) consist of at least 70-80 elements.

ASCII

One such standard code that allows the language encoding that is popularly used is ASCII (American Standard Code for Information Interchange). This code uses 7 bits

to represent 128 characters, which include 32 non-printing control characters, alphabets in lower and upper case, decimal digits, and other printable characters that are available on your keyboard. Later as there was need for additional characters to be represented such as graphics characters, additional special characters etc., ASCII was extended to 8 bits to represent 256 characters (called Extended ASCII codes). There are many variants of ASCII, they follow different code pages for language encoding, however, having the same format. You can refer to the complete set of ASCII characters on the web. The extended ASCII codes are the codes used in most of the Microcomputers.

The major strength of ASCII is that it is quite elegant in the way it represents characters. It is easy to write a code to manipulate upper/lowercase ASCII characters and check for valid data ranges because of the way of representation of characters.

In the original ASCII the 8th bit (the most significant bit) was used for the purpose of error checking as a check bit. We will discuss more about the check bits later in the Unit.

EBCDIC

Extended Binary Coded Decimal Interchange Code (EBCDIC) is a character-encoding format used by IBM mainframes. It is an 8-bit code and is NOT Compatible to ASCII. It had been designed primarily for ease of use of punched cards. This was primarily used on IBM mainframes and midrange systems such as the AS/400. Another strength of EBCDIC was the availability of wider range of control characters for ASCII. The character coding in this set is based on binary coded decimal, that is, the contiguous characters in the alphanumeric range are represented in blocks of 10 starting from 0000 binary to 1001 binary. Other characters fill in the rest of the range. There are four main blocks in the EBCDIC code:

0000 0000 to 0011 1111	Used for control characters
0100 0000 to 0111 1111	Punctuation characters
1000 0000 to 1011 1111	Lowercase characters
1100 0000 to 1111 1111	Uppercase characters and numbers.

There are several different variants of EBCDIC. Most of these differ in the punctuation coding. More details on EBCDIC codes can be obtained from further reading and web pages on EBCDIC.

Comparison of ASCII and EBCDIC

EBCDIC is an easier to use code on punched cards because of BCD compatibility. However, ASCII has some of the major advantages on EBCDIC. These are:
While writing a code, since EDCDIC is not contiguous on alphabets, data comparison to continuous character blocks is not easy. For example, if you want to check whether a character is an uppercase alphabet, you need to test it in range A to Z for ASCII as they are contiguous, whereas, since they are not contiguous range in EDCDIC these may have to be compared in the ranges A to I, J to R, and S to Z which are the contiguous blocks in EDCDIC.

Some of the characters such as [] \{}^~| are missing in EBCDIC. In addition, missing control characters may cause some incompatibility problems.

UNICODE

This is a newer International standard for character representation. Unicode provides a unique code for every character, irrespective of the platform, Program and Language. Unicode Standard has been adopted by the Industry. The key players that have adopted Unicode include Apple, HP, IBM, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many other companies. Unicode has been implemented in most of the

latest client server software. Unicode is required by modern standards such as XML, Java, JavaScript, CORBA 3.0, etc. It is supported in many operating systems, and almost all modern web browsers. Unicode includes character set of Dev Nagari. The emergence of the Unicode Standard, and the availability of tools supporting it, is among the most significant recent global software technology trends.

One of the major advantages of Unicode in the client-server or multi-tiered applications and websites is the cost saving over the use of legacy character sets that results in targeting website and software products across multiple platforms, languages and countries without re-engineering. Thus, it helps in data transfer through many different systems without any compatibility problems. In India the suitability of Unicode to implement Indian languages is still being worked out.

Indian Standard Code for Information Interchange (ISCII)

The ISCII is an eight-bit code that contains the standard ASCII values till 127 from 128-225 it contains the characters required in the ten Brahmi-based Indian scripts. It is defined in IS 13194:1991 BIS standard. It supports INSCRIPT keyboard which provides a logical arrangement of vowels and consonants based on the phonetic properties and usage frequencies of the letters of Bramhi-scripts. Thus, allowing use of existing English keyboard for Indian language input. Any software that uses ISCII codes can be used in any Indian Script, enhancing its commercial viability. It also allows transliteration between different Indian scripts through change of display mode.

2.6 DATA REPRESENTATION FOR COMPUTATION

As discussed earlier, binary codes exist for any basic representation. Binary codes can be formulated for any set of discrete elements e.g. colours, the spectrum, the musical notes, chessboard positions etc. In addition these binary codes are also used to formulate instructions, which are advanced form of data representation. We will discuss about instructions in more detail in the later blocks. But the basic question which remains to be answered is:

How are these codes actually used to represent data for scientific calculations?

The computer is a discrete digital device and stores information in flip-flops (see Unit 3, 4 of this Block for more details), which are two state devices, in binary form. Basic requirements of the computational data representation in binary form are:

- Representation of sign
- Representation of Magnitude
- If the number is fractional then binary or decimal point, and
- Exponent

The solution to sign representation is easy, because sign can be either positive or negative, therefore, one bit can be used to represent sign. By default it should be the left most bit (in most of the machines it is the Most Significant Bit).

Thus, a number of n bits can be represented as n+1 bit number, where $n+1^{th}$ bit is the sign bit and rest n bits represent its magnitude (Please refer to Figure 3).
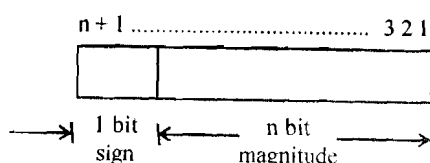

Figure 3: A (n + 1) bit number

The decimal position can be represented by a position between the flip-flops (storage cells in computer). But, how can one determine this decimal position? Well to simplify the representation aspect two methods were suggested: (1) Fixed point representation where the binary decimal position is assumed either at the beginning or at the end of a number; and (2) Floating point representation where a second register is used to keep the value of exponent that determines the position of the binary or decimal point in the number.

But before discussing these two representations let us first discuss the term "complement" of a number. These complements may be used to represent negative numbers in digital computers.

Complement: There are two types of complements for a number of base (also called radix) r. These are called r's complement and (r- 1)'s complement. For example, for decimal numbers the base is 10, therefore, complements will be 10's complement and (10-1) = 9's complement. For binary numbers we talk about 2's and 1's complements. But how to obtain complements and what do these complements means? Let us discuss these issues with the help of following example:

Example 2: Find the 9's complement and 10's complement for the decimal number 256.

Solution:

9's complement: The 9's complement is obtained by subtracting each digit of the number from 9 (the highest digit value). Let us assume that we want to represent a maximum of four decimal digit number range. 9's complement can be used for BCD numbers.

	9	9	9	9
9's complement of 256	-0	-2	-5	-6
	9	7	4	3

Similarly, for obtaining 1's complement for a binary number we have to subtract each binary digit of the number from the digit 1.

10's complement: Adding 1 in the 9's complement produces the 10's complement.
10's complement of 0256 = 9743+1 = 9744

Please note on adding the number and its 9's complement we get 9999 (the maximum possible number that can be represented in the four decimal digit number range) while on adding the number and its 10's complement we get 10000 (The number just higher than the range. This number cannot be represented in four digit representation.)
Example3: Find 1's and 2's complement of 1010 using only four-digit representation.

Solution:

1's complement: The 1's complement of 1010 is

1	1	1	1
-1	-0	-1	-0
0	1	0	1

The number is	1	0	1	0
The 1's complement is	0	1	0	1

Please note that wherever you have a digit 1 in number the complement contains 0 for that digit and vice versa. In other words to obtain 1's complement of a binary number, we only have to change all the 1's of the number to 0 and all the zeros to 1's. This can be done by complementing each bit of the binary number.

2's complement: Adding 1 in 1's complement will generate the 2's complement

The number is	1	0	1	0
The 1's complement is	0	1	0	1
For 2's complement add 1 in 1's complement	-	-	-	1
Please note that 1+1 = 1 0 in binary	0	1	1	0

⇧ Most Significant bit ⇧ Least significant bit

The number is	1	0	1	0
The 1's complement is	0	1	1	0

The 2's complement can also be obtained by not complementing the least significant zeros till the first 1 is encountered. This 1 is also not complemented. After this 1 the rest of all the bits are complemented on the left.

Therefore, 2's complement of the following number (using this method) should be (you can check it by finding 2's complement as we have done in the example).

The number is	0	0	1	0	0	1	0	0
The 2's complement is	1	1	0	1	1	1	0	0

No change in these bits

The number is	1	0	0	0	0	0	0	0
The 2's complement is	1	0	0	0	0	0	0	0

No change in number and its 2's Complement, a special case

The number is	0	0	1	0	1	0	0	1
The 2's complement is	1	1	0	1	0	1	1	1

No change in this bit only

2.6.1 Fixed Point Representation

The fixed-point numbers in binary uses a sign bit. A positive number has a sign bit 0, while the negative number has a sign bit 1. In the fixed-point numbers we assume that the position of the binary point is at the end, that is, after the least significant bit. It implies that all the represented numbers will be integers. A negative number can be represented in one of the following ways:

- Signed magnitude representation

- Signed 1's complement representation, or
- Signed 2's complement representation.

(Assumption: size of register = 8 bits including the sign bit)

Signed Magnitude Representation

Decimal Number	Representation (8 bits)	
	Sign Bit	Magnitude (7 bits)
+6	0	000 0110
-6	1	000 0110
No change in the Magnitude, only sign bit changes		

Signed 1's Complement Representation

Decimal Number	Representation (8 bits)	
	Sign Bit	Magnitude/ 1's complement for negative number (7 bits)
+6	0	000 0110
-6	1	111 1001
For negative number take 1's complement of all the bits (including sign bit) of the positive number		

Signed 2's Complement Representation

Decimal Number	Representation (8 bits)	
	Sign Bit	Magnitude/ 1's complement for negative number (7 bits)
+6	0	000 0110
-6	1	111 1010
For negative number take 2's complement of all the bits (including sign bit) of the positive number		

Arithmetic addition

The complexity of arithmetic addition is dependent on the representation, which has been followed. Let us discuss this with the help of following example.

Example 4: Add 25 and -30 in binary using 8 bit registers, using:

- Signed magnitude representation
- Signed 1's complement
- Signed 2's complement

Solution:

Number,	Signed Magnitude Representation	
	Sign Bit	Magnitude
+25	0	001 1001
-25	1	001 1001
+30	0	001 1110
-30	1	001 1110

To do the arithmetic addition with one negative number only, we have to check the magnitude of the numbers. The number having smaller magnitude is then subtracted from the bigger number and the sign of bigger number is selected. The implementation of such a scheme in digital hardware will require a long sequence of control decisions as well as circuits that will add, compare and subtract numbers. Is there a better alternative than this scheme? Let us first try the signed 2's complement.

Number	Signed Magnitude Representation	
	Sign Bit	Magnitude
+25	0	001 1001
-25	1	110 0111
+30	0	001 1110
-30	1	110 0010

Now let us perform addition using signed 2's complement notation:
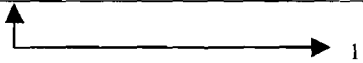
Operation	Decimal equivalent number	Signed 2's complement representation			Comments
		Carry out	Sign out	Magnitude	
Addition of two positive number	+25	-	0	001	1001	Simple binary addition. There is no carry out of sign bit
	+30	-	0	001	1110	
	+55	0	0	011	0111	
Addition of smaller Positive and larger negative Number	+25	-	0	001	1001	Perform simple binary addition. No carry in to the sign bit and no carry out of the sign bit
	-30	-	1	110	0010	
	-05	0	1	111	1011	
Positive value of result	+05	0	1	000	0101	2's complement of above result
Addition of larger Positive and smaller negative Number	-25	-	1	110	0111	Perform simple binary addition. No carry in to the sign bit and carry out of the sign bit
	+30	-	1	001	1110	
	+05	1	0	000	0101	
		↑ Discard the carry out bit				
Addition of two negative Numbers	-25	-	1	110	0111	Perform simple binary addition. There is carry in to the sign bit and carry out of the sign bit No overflow
	-30	-	1	110	0010	
	-55	1	1	110	1001	
		↑ Discard the carry out bit				
Positive value of result	+55	-	0	011	0111	2's complemnt of above result

Please note how easy it is to add two numbers using signed 2's Complement. This procedure requires only one control decision and only one circuit for adding the two numbers. But it puts on additional condition that the negative numbers should be stored in signed 2's complement notation in the registers. This can be achieved by

complementing the positive number bit by bit and then incrementing the resultant by 1 to get signed 2's complement.

Signed 1's complement representation

Another possibility, which also is simple, is use of signed 1's complement. Signed 1's complement has a rule. Add the two numbers, including the sign bit. If carry of the most significant bit or sign bit is one, then increment the result by 1 and discard the carry over. Let us repeat all the operations with 1's complement.

Operation	Decimal equivalent number	Signed 1's complement representation			Comments
		Carry out	Sign out	Magnitude	
Addition of two positive number	+25	-	0	001 1001	Simple binary addition. There is no carry out of sign bit
	+30	-	0	001 1110	
Addition of smaller Positive and larger negative Number	+55	0	0	001 0111	
	+25	-	0	001 1001	Perform simple binary addition. No carry in to the sign bit and no carry out of the sign bit
	-30	-	1	110 0001	
	-05	0	1	111 1011	
Positive value of result	+05	-	0	000 0101	1's complement of above result
Addition of larger Positive and smaller negative Number	-25	-	1	110 0111	There is carry in to the sign bit and carry out of the sign bit. The carry out is added it to the Sum bit and then discard no overflow.
	+30	-	0	001 1110	
		1	0	000 0101	
	Add carry to Sum and discard it	⬆ ➔ 1			
	+05	-	0	000 0101	
Addition of two negative Numbers	-25	-	1	110 0111	Perform simple binary addition. There is carry in to the sign bit and carry out of the sign bit No overflow
	-30	-	1	110 0010	
	-55	1	1	100 0111	
	Add carry to sum and discard it	⬆ ➔ 1			
		-	1	100 1000	
Positive value of result	+55	-	0	011 0111	1's complemnt of above result

Another interesting feature about these representations is the representation of 0. In signed magnitude and 1's complement there are two representations for zero as:

Representation	+ 0		-0	
Signed magnitude	0	000 0000	1	000 0000
Signed 1's complement	0	000 0000	1	111 1111

But, in signed 2's complement there is just one zero and there is no positive or negative zero.

+0 in 2's Complement Notation:0 000 0000

-0 in 1's complement notation: 1 111 1111

Add 1 for 2's complement: _____1___

Discard the Carry Out 1   0 000 0000

Thus, -0 in 2's complement notation is same as +0 and is equal to 0 000 0000. Thus, both +0 and -0 are same in 2's complement notation. This is an added advantage in favour of 2's complement notation.

The highest number that can be accommodated in a register, also depends on the type of representation. In general in an 8 bit register 1 bit is used as sign, therefore, the rest 7 bits can be used for representing the value. The highest and the lowest numbers that can be represented are:

For signed magnitude representation $(2^7 - 1)$ to $- (2^7 - 1)$

$$= (128-1) \text{ to } - (128-1)$$

$$= 127 \text{ to } -127$$

For signed 1's complement   127 to −127

But, for signed 2's complement we can represent +127 to −128. The − 128 is represented in signed 2's complement notation as 10000000.

Arithmetic Subtraction: The subtraction can be easily done using the 2's complement by taking the 2's complement of the value that is to be subtracted (inclusive of sign bit) and then adding the two numbers.

Signed 2's complement provides a very simple way for adding and subtracting two numbers. Thus, many computers (including IBM PC) adopt signed 2's complement notation. The reason why signed 2's complement is preferred over signed 1's complement is because it has only one representation for zero.

Overflow: An overflow is said to have occurred when the sum of two n digits number occupies n+ 1 digits. This definition is valid for both binary as well as decimal digits.

What is the significance of overflow for binary numbers?

Well, the overflow results in errors during binary arithmetic as the numbers are represented using a fixed number of digits also called the size of the number. Any value that results from computation must be less than the maximum of the allowed value as per the size of the number. In case, a result of computation exceeds the maximum size, the computer will not be able to represent the number correctly, or in other words the number has overflowed. Every computer employs a limit for representing numbers e.g. in our examples we are using 8 bit registers for calculating the sum. But what will happen if the sum of the two numbers can be accommodated in 9 bits? Where are we going to store the 9th bit, The problem will be better understood by the following example.

Example: Add the numbers 65 and 75 in 8 bit register in signed 2's complement notation.

65	0	100 0001
75	0	100 1011
140	1	000 1100

The expected result is +140 but the binary sum is a negative number and is equal to −116, which obviously is a wrong result. This has occurred because of overflow.

How does the computer know that overflow has occurred?

If the **carry into the sign bit is not equal to the carry out of the sign bit** then overflow must have occurred.

Another simple test of overflow is: if the sign of both the operands is same during addition, then overflow must have occurred if the sign of resultant is different than that of sign of any operand.

For example

Decimal	Carry out	Sign bit	2's Complement Mantissa		Decimal	Carry out	Sign bit	2's Complement Mantissa	
-65		1	011	1111	-65		1	011	1111
-15		1	111	0001	-75		1	111	0001
-80	1	1	011	0000	-140	1	0	111	0100

Carry into Sign bit = 1   Carry into Sign bit = 0
Carry out of sign bit = 1   Carry out of Sign bit = 1
Therefore, NO OVERFLOW   Therefore, OVERFLOW

Thus, overflow has occurred, i.e. the arithmetic results so calculated have exceeded the capacity of the representation. This overflow also implies that the calculated results will be erroneous.

2.6.2 Decimal Fixed Point Representation

The purpose of this representation is to keep the number in decimal equivalent form and not binary as above. A decimal digit is represented as a combination of four bits; thus, a four digit decimal number will require 16 bits for decimal digits representation and additional 1 bit for sign. Normally to keep the convention of one decimal digit to 4 bits, the sign sometimes is also assigned a 4-bit code. This code can be the bit combination which has not been used to represent decimal digit e.g. 1100 may represent plus and 1101 can represent minus.

For example, a simple decimal number – 2156 can be represented as:

1101 0010 0001 0101 0110

Sign

Although this scheme wastes considerable amount of storage space yet it does not require conversion of a decimal number to binary. Thus, it can be used at places where the amount of computer arithmetic is less than that of the amount of input/output of data e.g. calculators or business data processing situations. The arithmetic in decimal can also be performed as in binary except that instead of signed complement, signed nine's complement is used and instead of signed 2's complement signed 9's complement is used. More details on decimal arithmetic are available in further readings.

Check Your Progress 2

1)   Write the BCD equivalent for the three numbers given below:

i)   23

ii) 49.25

iii) 892

....

....

....

....

2) Find the 1's and 2's complement of the following fixed-point numbers.

   i)   10100010

   ii) 00000000

   iii) 11001100

   ....

   ....

   ....

   ....

3) Add the following numbers in 8-bit register using signed 2's complement notation

   i)   +50 and – 5

   ii) +45 and – 65

   iii) +75 and +85

   Also indicate the overflow if any.

   ....

   ....

   ....

   ....

2.6.3   Floating Point Representation

Floating-point number representation consists of two parts. The first part of the number is a signed fixed-point number, which is termed as mantissa, and the second part specifies the decimal or binary point position and is termed as an Exponent. The mantissa can be an integer or a fraction. Please note that the position of decimal or binary point is assumed and it is not a physical point, therefore, wherever we are representing a point it is only the assumed position.

Example 1: A decimal + 12.34 in a typical floating point notation can be represented in any of the following two forms:

Sign   Sign
↓   ↓

| 0   1234 | | 0   02 |
Mantissa (fraction)   Exponent

Sign   Sign
↓   ↓

| 0   1234 | | 1   02 |
Mantissa (Integer)   Exponent
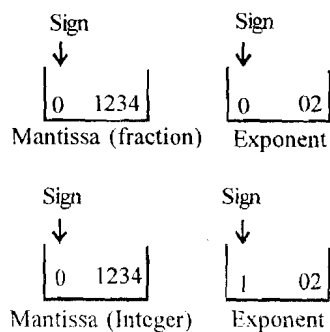
This number in any of the above forms (if represented in BCD) requires 17 bits for mantissa (1 for sign and 4 each decimal digit as BCD) and 9 bits for exponent (1 for sign and 4 for each decimal digit as BCD). Please note that the exponent indicates the correct decimal location. In the first case where exponent is +2, indicates that actual position of the decimal point is two places to the right of the assumed position, while exponent– 2 indicates that the assumed position of the point is two places towards the left of assumed position. The assumption of the position of point is normally the same in a computer resulting in a consistent computational environment.

Floating-point numbers are often represented in normalised forms. A floating point number whose mantissa does not contain zero as the most significant digit of the number is considered to be in normalised form. For example, a BCD mantissa + 370 which is 0 0011 0111 0000 is in normalised form because these leading zero's are not part of a zero digit. On the other hand a binary number 0 01100 is not in a normalised form. The normalised form of this number is:

0	1100	0100
Sign	Normalised Mantissa	Exponent (assuming fractional Mantissa

A floating binary number +1010.001 in a 16-bit register can be represented in normalised form (assuming 10 bits for mantissa and 6 bits for exponent).
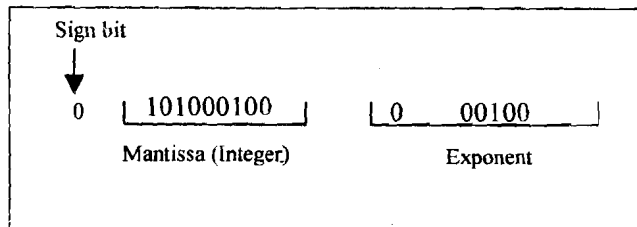
A zero cannot be normalised as all the digits in mantissa in this case have to be zero.

Arithmetic operations involved with floating point numbers are more complex in nature, take longer time for execution and require complex hardware. Yet the floating-point representation is a must as it is useful in scientific calculations. Real numbers are normally represented as floating point numbers.

The following figure shows a format of a 32-bit floating-point number.

0	1	8	9	31
Sign	Biased Exponent = 8 bits		Significand = 23 bits	

Figure 4: Floating Point Number Representation

The characteristics of a typical floating-point representation of 32 bits in the above figure are:

• Left-most bit is the sign bit of the number;
• Mantissa or signific and should be in,normalised form;
• The base of the number is 2, and
• A value of 128 is added to the exponent. (Why?) This is called a bias.

A normal exponent of 8 bits normally can represent exponent values as 0 to 255. However, as we are adding 128 for getting the biased exponent from the actual exponent, the actual exponent values represented in the range will be – 128 to 127.

Now, let us define the range that a normalised mantissa can represent. Let us assume that our present representations has the normalised mantissa, thus, the left most bit

cannot be zero, therefore, it has to be 1. Thus, it is not necessary to store this first bit and it is being assumed **implicitly** for the number. Therefore, a 23-bit mantissa can represent $23 + 1 = 24$ bit mantissa in our representation.

Thus, the smallest mantissa value may be:

The implicit first bit as 1 followed by 23 zero's, that is,

0.1000 0000 0000 0000 0000 0000

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

The Maximum value of the mantissa:
The implicit first bit 1 followed by 23 one's, that is,

0.1111 1111 1111 1111 1111 1111

Decimal equivalent:
For finding binary equivalent let us add 2^{-24} to above mantissa as follows:

Binary: 0.1111 1111 1111 1111 1111 1111

+ <u>0.0000 0000 0000 0000 0000 0001</u> $= 2^{-24}$

  1.0000 0000 0000 0000 0000 0000 $= 1$

$= (1 - 2^{-24})$

Therefore, in normalised mantissa and biased exponent form, the floating-point number format as per the above figure, can represent binary floating-point numbers in the range:

Smallest Negative number

      Maximum mantissa and maximum exponent

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

Largest negative number

      Minimum mantissa and Minimum exponent

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

Smallest positive number

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

Largest positive number

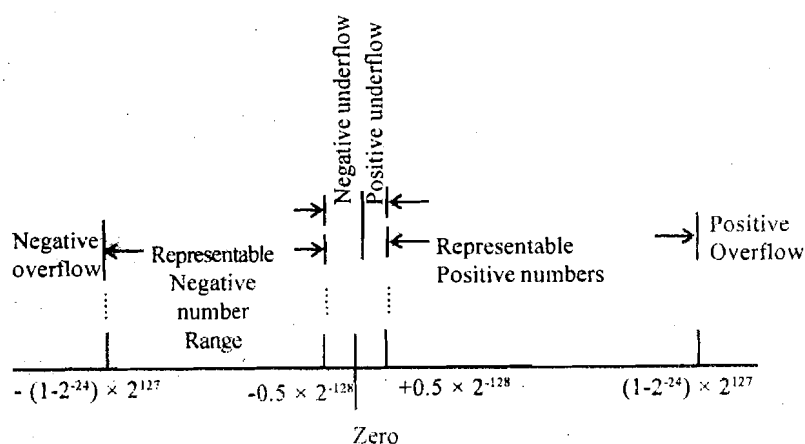$$= (1 - 2^{-24}) \times 2^{127}$$

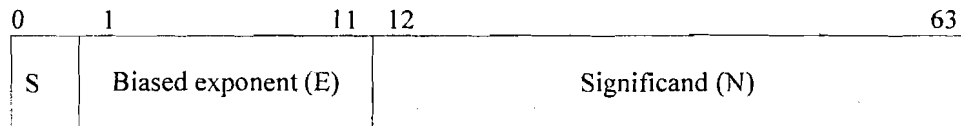Figure 5: Binary floating-point number range for given 32 bit format

In floating point numbers, the basic trade-off is between the range of the numbers and accuracy, also called the precision of numbers. If we increase the exponent bits in 32-bit format, the range can be increased, however, the accuracy of numbers will go down, as size of mantissa will become smaller. Let us take an example, which will clarify the term precision. Suppose we have one bit binary mantissa then we can represent only 0.10 and 0.11 in the normalised form as given in above example (having an implicit 1). The values such as 0.101, 0.1011 and so on cannot be represented as complete numbers. Either they have to be approximated or truncated and will be represented as either 0.10 or 0.11. Thus, it will create a truncation or round off error. The higher the number of bits in mantissa better will be the precision.

In floating point numbers, for increasing both precision and range more number of bits are needed. This can be achieved by using double precision numbers. A double precision format is normally of 64 bits.

Institute of Electrical and Electronics Engineers (IEEE) is a society, which has created lot of standards regarding various aspects of computer, has created IEEE standard 754 for floating-point representation and arithmetic. The basic objective of developing this standard was to facilitate the portability of programs from one to another computer. This standard has resulted in development of standard numerical capabilities in various microprocessors. This representation is shown in figure 6.

0	1	8	9	31

S	Biased exponent (E)	Significand (N)
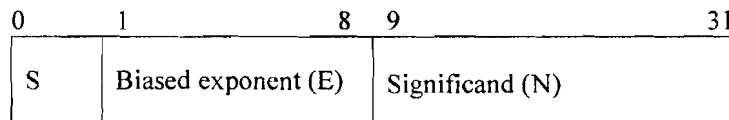
Single Precision = 32 bits

0	1	11	12	63

S	Biased exponent (E)	Significand (N)

Double Precision = 64 bits

Figure 6: IEEE Standard 754 format

Figure 7 gives the floating-point numbers specified by the IEEE Standard 754.

Single Precision Numbers (32 bits)

Exponent (E)	Significand (N)	Value / Comments
255	Not equal to 0	Do represent a number
255	0	- or +∞ depending on sign bit
0<E<255	Any	$\pm (1.N)\, 2^{E-127}$ For example, if S is zero that is positive number. N=101 (rest 20 zeros) and E=207 Then the number is $= +(1.101)\, 2^{207-127}$ $= + 1.101 \times 2^{80}$
0	Not equal to 0	$\pm (0.N)\, 2^{-126}$
0	0	$\pm\ 0$ depending on the sign bit.
Double precision Numbers (64 bits)		
Exponent (E)	Significand (N)	Value / Comments
2047	Not equal to 0	Do not represent a number

2047	0	- or $+\infty$ depending on the sign bit
0<E<2047	Any	$\pm (1.N) 2^{E-1023}$
0	Not equal to 0	$\pm (0.N) 2^{-1022}$
0	0	± 0 depending on the sign bit

Figure 7: Values of floating point numbers as per IEEE standard 754

Please note that IEEE standard 754 specifies plus zero and minus zero and plus infinity and minus infinity. Floating point arithmetic is more sticky than fixed point arithmetic. For floating point addition and subtraction we have to follow the following steps:

- Check whether a typical operand is zero
- Align the significand such that both the significands have same exponent
- Add or subtract the significand only and finally
- The significand is normalised again

These operations can be represented as

$$x + y = \left(N_x \times 2^{Ex-Ey} + N_y\right) \times 2^{Ey}$$

$$\text{and } x - y = \left(N_x \times 2^{Ex-Ey} - N_y\right) \times 2^{Ey}$$

Here, the assumption is that exponent of x (Ex) is greater than exponent of y (Ey), Nx and Ny represent significand of x and y respectively.

While for multiplication and division operations the significand need to be multiplied or divided respectively, however, the exponents are to be added or to be subtracted respectively. In case we are using bias of 128 or any other bias for exponents then on addition of exponents since both the exponents have bias, the bias gets doubled. Therefore, we must subtract the bias from the exponent on addition of exponents. However, bias is to be added if we are subtracting the exponents. The division and multiplication operation can be represented as:

$$x \times y = (N_x \times N_y) \times 2^{Ex+Ey}$$

$$x \div y = (N_x \div N_y) \times 2^{Ex-Ey}$$

For more details on floating point arithmetic you can refer to the further readings.

2.6.4 Error Detection and Correction Codes

Before we wind up the data representation in the context of today's computers one must discuss about the code, which helps in recognition and correction of errors. Computer is an electronic media; therefore, there is a possibility of errors during data transmission. Such errors may result from disturbances in transmission media or external environment. But what is an error in binary bit? An error bit changes from 0 to 1 or 1 to 0. One of the simplest error detection codes is called parity bit.

Parity bit: A parity bit is an error detection bit added to binary data such that it makes the total number of 1's in the data either odd or even. For example, in a seven bit data 0110101 an 8th bit, which is a parity bit may be added. If the added parity bit is even parity bit then the value of this parity bit should be zero, as already four 1's exists in the 7-bit number. If we are adding an odd parity bit then it will be 1, since we already have four 1 bits in the number and on adding 8th bit (which is a parity bit) as 1 we are making total number of 1's in the number (which now includes parity bit also) as 5, an odd number.

Similarly in data 0010101   Parity bit for even parity is 1
                                     Parity bit for odd parity is 0

But how does the parity bit detect an error? We will discuss this issue in general as an error detection and correction system (Refer figure 8).

But how does the parity bit detect an error? We will discuss this issue in general as an error detection and correction system (Refer figure 8).

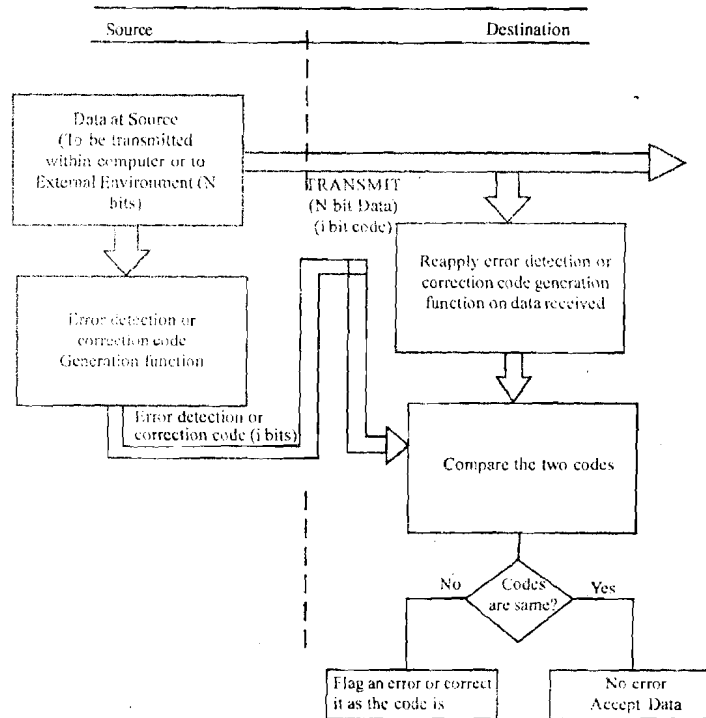The error detection mechanism can be defined as follows:

Figure 8: Error detection and correction

The Objective : Data should be transmitted between a source data pair reliably, indicating error, or even correcting it, if possible.

The Process:

- An error detection function is applied on the data available at the source end an error detection code is generated.
- The data and error detection or correction code are stored together at source.
- On receiving the data transmission request, the stored data along with stored error detection or correction code are transmitted to the unit requesting data (Destination).
- On receiving the data and error detection/correction code from source, the destination once again applies same error detection/correction function as has been applied at source on the data received (but not on error detection/ correction code received from source) and generates destination error detection/correction code.
- Source and destination error codes are compared to flag or correct an error as the case may be.

The parity bit is only an error detection code. The concept of error detection and correction code has been developed using more than one parity bits. One such code is Hamming error correcting code.

Hamming Error-Correcting Code: Richard Hamming at Bell Laboratories devised this code. We will just introduce this code with the help of an example for 4 bit data.

Let us assume a four bit number b4, b3, b2, b1. In order to build a simple error detection code that detects error in one bit only, we may just add an odd parity bit. However, if we want to find which bit is in error then we may have to use parity bits

for various combinations of these 4 bits such that a bit error can be identified uniquely. For example, we may create four parity sets as

	Source Parity	Destination Parity
b1, b2, b3	P1	D1
b2, b3, b4	P2	D2
b3, b4, b1	P3	D3
b1, b2, b3, b4	P4	D4

Now, a very interesting phenomena can be noticed in the above parity pairs. Suppose data bit b1 is in error on transmission then, it will cause change in destination parity D1, D3, D4.

ERROR IN (one bit only)	Cause change in Destination Parity
b1	D1, D3, D4
b2	D1, D2, D4
b3	D1, D2, D3, D4
b4	D2, D3, D4

Figure 9 : The error detection parity code mismatch

Thus, by simply comparing parity bits of source and destination we can identify that which of the four bits is in error. This bit then can be complemented to remove error. Please note that, even the source parity bit can be in error on transmission, however, under the assumption that only one bit (irrespective of data or parity) is in error, it will be detected as only one destination parity will differ.

What should be the length of the error detection code that detects error in one bit? Before answering this question we have to look into the comparison logic of error detection. The error detection is done by comparing the two 'i' bit error detection and correction codes fed to the comparison logic bit by bit (refer to figure 8). Let us have comparison logic, which produces a zero if the compared bits are same or else it produces a one.

Therefore, if similar Position bits are same then we get zero at that bit Position, but if they are different, that is, this bit position may point to some error, then this Particular bit position will be marked as one. This way a matching word is constructed. This matching word is 'i' bit long, therefore, can represent 2^i values or combinations.

For example, a 4-bit matching word can represent $2^4=16$ values, which range from 0 to 15 as:

0000,   0001,   0010,   0011,   0100,   0101,   0110,   0111

1000,   1001,   1010,   1011,   1100,   1101,   1110,   1111

The value 0000 or 0 represent no error while the other values i.e. 2^i-1 (for 4 bits $2^4-1=15$, that is from 1 to 15) represent an error condition. Each of these 2^i-1 (or 15 for 4 bits) values can be used to represent an error of a particular bit. Since, the error can occur during the transmission of 'N' bit data plus 'i' bit error correction code, therefore, we need to have at least 'N+i' error values to represent them. Therefore, the number of error correction bits should be found from the following equation:

$$2^i - 1 \quad >= N+i$$

If we are assuming 8-bit word then we need to have

$$2^i - 1 \quad >= 8+i$$

Say at i=3   LHS $= 2^3 - 1 = 7$; RHS $= 8+3 = 11$

i=4 $2^i\text{-}1 = 2^4 - 1 = 15$; RHS $= 8+4 = 12$

Therefore, for an eight-bit word we need to have at least four-bit error correction code for detecting and correcting errors in a single bit during transmission.

Similarly for 16 bit word we need to have i = 5

$2^5 - 1 = 31$ and $16+i = 16+5 = 21$

For 16-bit word we need to have five error correcting bits.

Let us explain this with the help of an example:

Let us assume 4 bit data as 1010

The logic is shown in the following table:

Source:

Source Data   **Odd parity bits at source**

b4	b3	b2	b1	P1 (b1, b2, b3)	P2 (b2, b3, b4)	P3 (b3, b4, b1)	P4 (b1, b2, b3,b4)
1	0	1	0	0	1	0	1

This whole information, that is (data and P1 to P4), is transmitted.

Assuming one bit error in data.

Case 1:   Data received as 1011 (Error in b1)

b4	b3	b2	b1	D1 (b1, b2, b3)	D2 (b2, b3, b4)	D3 (b3, b4, b1)	D4 (b1, b2, b3,b4)
1	0	1	0	0	1	0	1

Thus, P1 – D1, P3 – D3, P4 –D4 pair differ, thus, as per Figure 9, b1 is in error, so correct it by completing b1 to get correct data 1010.

Case 2:   Data Received as 1000 (Error in b2)

b4	b3	b2	b1	D1 (b1, b2, b3)	D2 (b2, b3, b4)	D3 (b3, b4, b1)	D4 (b1, b2, b3,b4)
1	0	0	0	0	1	0	0

Thus, P1 – D1, P2 – D2, P4 – D4 pair differ, thus, as per figure 9, bit b2 is in error. So correct it by complementing it to get correct data 1010.

Case 3:

Now let us take a case when data received is correct but on receipt one of the parity bit, let us say P4 become 0. Please note in this case since data is 1010 the destination parity bits will be D1=0, D2=1, D3=0, D4=1. Thus, P1– D1, P2 – D2, P3 – D3, will be same but P4 –D4 differs. This does not belong to any of the combinations in figure 9. Thus we conclude that P4 received is wrong.

Please not that all these above cases will fail in case error is in more than one bits. Let us see by extending the above example.

Normally, Single Error Correction (SEC) code is used in semiconductor memories for correction of single bit errors, however, it is supplemented with an added feature for detection of errors in two bits. This is called a SEC-DED (Single Error Correction-Double Error Detecting) code. This code requires an additional check bit in comparison to SEC code. We will only illustrate the working principle of SEC-DED code with the help of an example for a 4-bit data word. Basically, the SEC-DED code guards against the errors of two bits in SEC codes.

Case: 4

Let us assume now that two bit errors occur in data.
Data received:

b4	b3	b2	b1
1	1	0	0

b4	b3	b2	b1	D1 (b1, b2, b3)	D2 (b2, b3, b4)	D3 (b3, b4, b1)	D4 (b1, b2, b3,b4)
1	0	0	0	0	1	0	0

Thus, on -matching we find P3-D3 pair does not match.
However, this information is wrong. Such problems can be identified by adding one more bit to this Single Error Detection Code. This is called Double Error Detection bit (P5, D5).

So our data now is

b4 b3 b2 b1 P1 P2 P3 P4 P5 (Overall parity of whole data)

1   0   1   0   0   1   0   1   1

Data receiving end.

b4 b3 b2 b1 D1 D2 D3 D4 D5

1   1   0   0   0   1   1   1   0

D5–P5 mismatch indicates that there is double bit error, so do not try to correct error, instead asks the sender to send the data again. Thus, the name single error correction, but double error detection, as this code corrects single bit errors but only detects error in two bit.

Check Your Progress 3

1) Represent the following numbers in IEEE-754 floating point single precision number format:

   i)   1010. 0001
   ii) –0.0000111

2) Find the even and odd parity bits for the following 7-bit data:

   i)   0101010
   ii)   0000000
   iii) 1111111
   iv)   1000100

3) Find the length of SEC code and SEC-DED code for a 16-bit word data transfer.

....

....

....

2.7 SUMMARY

This unit provides an in-depth coverage of the data representation in a computer system. We have also covered aspects relating to error detection mechanism. The unit covers number system, conversion of number system, conversion of numbers to a different number system. It introduces the concept of computer arithmetic using 2's complement notation and provides introduction to information representation codes like ASCII, EBCDIC, etc. The concept of floating point numbers has also been covered with the help of a design example and IEEE-754 standard. Finally error detection and correction mechanism is detailed along with an example of SEC & SEC-DED code.

The information given on various topics such as data representation, error detection codes etc. although exhaustive yet can be supplemented with additional reading. In fact, a course in an area of computer must be supplemented by further reading to keep your knowledge up to date, as the computer world is changing with by leaps and bounds. In addition to further reading the student is advised to study several Indian Journals on computers to enhance his knowledge.

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1.

(i)

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	1	0	0	1	1	0	1

thus; Integer $= (1\times2^3+1\times2^2+0\times2^1+0\times2^0) = (2^3+2^2) = (8+4) = 12$
Fraction $= (1\times2^{-1}+1\times2^{-2}+0\times2^{-3}+1\times2^{-4}) = 2^{-1}+2^{-2}+2^{-4} = 0.5+0.125 + 0.0625 = 0.6875$

ii) 10101010

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	1	0	1	0
128	64	32	16	8	4	2	1
=1	0	1	0	1	0	1	0

The decimal equivalent is

$= 1\times 128 + 0\times64 + 1\times32 + 0\times16 + 1\times8 + 0\times4 + 1\times2 + 0\times1$

$= 128 + 32 + 8 + 2 = 170$

2.

i)

16	8	4	2	1
1	0	1	1	1

ii) Integer is 49.

32	16	8	4	2	1
1	1	0	0	0	1

Fraction is 0.25

1/2	1/4	1/8	1/16
0	1	0	0

The decimal number 49.25 is 110001.010

iii)

512	256	128	64	32	16	8	4	2	1
1	1	0	1	1	1	1	1	0	0

The decimal number 892 in binary is 1101111100

3)

i) Decimal to Hexadecimal

```
        16) 23 (1
             -16
            ____

              7
```
Hexadecimal is 17
Binary to Hexadecimal (hex)

```
=  1     0111   (from answer of 2 (i)

  0001   0111
_____
    1      7
```

ii)

49.25   or 110001.010

Decimal to hex

Integer part = 49

```
16 ) 49 ( 3
     -48
    ____
       1
```

Integer part =   31

Fraction part   = $.25 \times 16$

              =   4.000   So fraction part = 4

Hex number is 31.4

```
Binary to hex    11    0001 .   010
          =    0011   0001 .   0100
          =    3      1   .   4
          =      31.4
```

iii) 892 or 1101111100

```
   0011   0111   1100
 = 3       7      C
 = 37C
```

Number	Quotient on division by 16	Remainder
892	55	12=C
55	3	7
3	0	3

So the hex number is : 37C

Check Your Progress 2

1.

i)   23 in BCD is 0010 0011

ii)   49.25 in BCD is 0100 1001.0010 0101

iii)   892 in BCD is 1000 1001 0010

2.   1's complement is obtained by complementing each bit while 2's complement is obtained by leaving the number unchanged till first 1 starting from least significant bit after that complement each bit.

	(i)	(ii)	(iii)
Number	10100010	00000000	11001100
1's complement	01011101	11111111	00110011
2's complement	01011110	00000000	00110100

3.   We are using signed 2's complement notation

(i)   +50   is   0   0110010
             +5   is   0   0000101
therefore   −5   is   1   1111011

Add +50   =   0   0110010

    − 5   1   1111011
    _____

            1   0   0101101

carry out (discard the carry)

Carry in to sign bit   = 1

Carry out of sign bit   = 1   Therefore, no overflow

The solution is 0010 1101 = +45

ii)   +45   is   0   0101101
             +65   is   0   1000001
Therefore,   -65   is   1   0111111
             +45   0   0101101
             − 65   1   0111111
             _____
                       1   1101100

No carry into sign bit, no carry out of sign bit. Therefore, no overflow.

+20 is   0   0010100

Therefore, -20 is   1   1101100

which is the given sum

(iii)

+75	is	0	1001011	
+85	is	0	1010101	
		1	0100000	

Carry into sign bit = 1
Carry out of sign bit = 0
Overflow.

Check Your Progress 3

1.

i) 1010.0001
   = 1.0100001 $\times 2^3$
   So, the single precision number is :
   Significand = 010 0001 000 0000 0000 0000
   Exponent = 3+127 = 130 = 10000010
                                    Sign=0
   So the number is = 0 1000 0010 010 0001 0000 0000 0000 0000

ii) -0.0000111
    -1.11 $\times 2^{-5}$
Significand = 110 0000 0000 0000 0000 0000

Exponent = 127-5 = 122 = 0111 1010
Sign   = - ≡ 1
So the number is
1 0111 1010   110 0000 0000 0000 0000 0000

2.
Data	Even parity bit	Odd parity bit
0101010	1	0
0000000	0	1
1111111	1	0
1000100	0	1

3. The equation for SEC code is
    $2^i - 1 >= N + i$
    i — Number of bits in SEC code
    N — Number of bits in data word

In, this case   N = 16
                  i = ?

so the equation is

    $2^i - 1 >= 16 + i$

at   i   = 4
        $2^4 - 1 >= 16 + 4$
       15 >= 20 Not true.

at   i   = 5
        $2^5 - 1 >= 16 + 5$
        31 >= 21 True the condition is satisfied.

Although, this condition will be true for i > 5 also but we want to use only minimum essential correction bits which are 5.

For SEC-DED code we require an additional bit as overall parity. Therefore, the SEC-DED code will be of 6 bits.