

CHAPTER 2

NUMBER SYSTEMS AND CODES

2.1 INTRODUCTION

We all are familiar with the number system in which an ordered set of ten symbols—0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, known as *digits*—are used to specify any number. This number system is popularly known as the *decimal number system*. The *radix* or *base* of this number system is 10 (number of distinct digits). Any number is a collection of these digits. For example, 1982.365 signifies a number with an *integer* part equal to 1982 and a *fractional* part equal to 0.365, separated from the integer part with a *radix point* (.) also known as *decimal point*. There are some other systems also, used to represent numbers. Some of the other commonly used number systems are: *binary*, *octal* and *hexadecimal* number systems. These number systems are widely used in digital systems like microprocessors, logic circuits, computers, etc. and therefore, the knowledge of these number systems is very essential for understanding, analysing and designing digital systems.

As discussed in Chapter 1, computers and other digital circuits use binary signals but are required to handle data which may be numeric, alphabets or special characters. Therefore, the information available in any other form is required to be converted into suitable binary form before it can be processed by digital circuits. This means that the information available in the form of numerals, alphabets and special characters or in any combination of these must be converted into binary format. To achieve this, a process of coding is employed whereby each numeral, alphabet or special character is coded in a unique combination of 0s and 1s using a coding scheme, known as a *code*. The process of coding is known as *encoding*.

There can be a variety of coding schemes (codes) to serve different purposes, such as arithmetic operations, data entry, error detection and correction, etc. In digital systems, a large number of codes are in use. Selection of a particular code depends on its suitability for the purpose. In one digital system, different codes may be used for different operations and it may be necessary to convert data from one code to another code. For this purpose, code converter circuits are required which will be discussed later.

2.2 NUMBER SYSTEMS

In general, in any number system there is an ordered set of symbols known as digits with rules defined for performing arithmetic operations like addition, multiplication, etc. A collection of these digits makes a number which in general has two parts—integer and fractional, set apart by a radix point (.), that is

$$(N)_b = \underbrace{d_{n-1} d_{n-2} \dots d_i \dots d_1}_{\text{Integer portion}} \underbrace{d_0}_{\substack{\uparrow \\ \text{Radix} \\ \text{point}}} \underbrace{d_{-1} d_{-2} \dots d_{-f} \dots d_{-m}}_{\text{Fractional portion}} \quad (2.1)$$

where N = a number

b = radix or base of the number system

n = number of digits in integer portion

m = number of digits in fractional portion

d_{n-1} = most significant digit (msd)

d_{-m} = least significant digit (lsd)

and

$$0 \leq (d_i \text{ or } d_{-j}) \leq b - 1$$

The digits in a number are placed side by side and each position in the number is assigned a *weight* or *index* of importance by some predesigned rule. Table 2.1 gives the details of commonly used number systems.

Table 2.1 *Characteristics of Commonly Used Number Systems*

Number system	Base or radix (b)	Symbols used (d_i or d_{-j})	Weight assigned to position		Example
			i	$-j$	
Binary	2	0, 1	2^i	2^{-j}	1011.11
Octal	8	0, 1, 2, 3, 4, 5, 6, 7	8^i	8^{-j}	3567.25
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	10^i	10^{-j}	3974.57
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	16^i	16^{-j}	3F49.56

2.3 BINARY NUMBER SYSTEM

The number system with base (or radix) two is known as the *binary number system*. Only two symbols are used to represent numbers in this system and these are 0 and 1. These are known as bits. This system has the minimum base (0 is not possible and 1 is not useful). It is a positional system, that is every position is assigned a specific weight.

Table 2.2 illustrates counting in binary number system. The corresponding decimal numbers are given in the right-hand column. Similar to decimal number system, the left-most bit is known as the *most significant bit* (MSB) and the right-most bit is known as the *least significant bit* (LSB). Any number of 0s can be added to the left of the number without changing the value of the number. In the binary number system, a group of four bits is known as a *nibble*, and a group of eight bits is known as a *byte*.

Table 2.2 4-bit Binary Numbers and Their Corresponding Decimal Numbers

Binary number				Decimal number	
B_3	B_2	B_1	B_0	D_1	D_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	2
0	0	1	1	0	3
0	1	0	0	0	4
0	1	0	1	0	5
0	1	1	0	0	6
0	1	1	1	0	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	1	2
1	1	0	1	1	3
1	1	1	0	1	4
1	1	1	1	1	5

2.3.1 Binary-to-Decimal Conversion

Any binary number can be converted into its equivalent decimal number using the weights assigned to each bit position as given in Table 2.1.

Example 2.1

Find the decimal equivalent of the binary number $(1\ 1\ 1\ 1)_2$.

Solution

The equivalent decimal number is

$$\begin{aligned}
 &= 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
 &= 16 + 8 + 4 + 2 + 1 \\
 &= (31)_{10}
 \end{aligned}$$

To differentiate between numbers represented in different number systems, either the corresponding number system may be specified along with the number or a small subscript at the end of the number may

be added signifying the number system. For example, $(1000)_2$ represents a binary number and is not one thousand.

Example 2.2

Determine the decimal numbers represented by the following binary numbers:

- (a) 110101 (b) 101101 (c) 11111111 (d) 00000000

Solution

- (a) $(110101)_2 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
 $= 32 + 16 + 0 + 4 + 0 + 1$
 $= (53)_{10}$
 (b) $(101101)_2 = 32 + 0 + 8 + 4 + 0 + 1$
 $= (45)_{10}$
 (c) $(11111111)_2 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$
 $= (255)_{10}$
 (d) $(00000000)_2 = (0)_{10}$

Example 2.3

Determine the decimal numbers represented by the following binary numbers:

- (a) 101101.10101 (b) 1100.1011 (c) 1001.0101 (d) 0.10101

Solution

- (a) $(101101.10101)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}$
 $= 32 + 0 + 8 + 4 + 0 + 1 + \frac{1}{2} + 0 + \frac{1}{8} + 0 + \frac{1}{32}$
 $= (45.65625)_{10}$
 (b) $(1100.1011)_2 = 8 + 4 + 0 + 0 + 0.5 + 0 + 0.125 + 0.0625$
 $= (12.6875)_{10}$
 (c) $(1001.0101)_2 = 8 + 0 + 0 + 1 + 0 + 0.25 + 0 + 0.0625$
 $= (9.3125)_{10}$
 (d) $(0.10101)_2 = 0.5 + 0 + 0.125 + 0 + 0.03125$
 $= (0.65625)_{10}$

2.3.2 Decimal-to-Binary Conversion

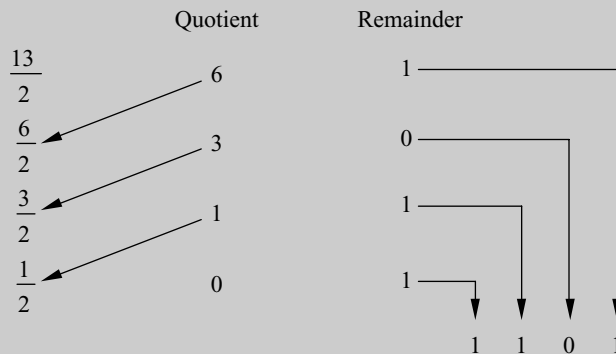
Any decimal number can be converted into its equivalent binary number. For integers, the conversion is obtained by continuous division by 2 and keeping track of the remainders, while for fractional parts, the

conversion is affected by continuous multiplication by 2 and keeping track of the integers generated. The conversion process is illustrated by the following examples.

Example 2.4

Convert $(13)_{10}$ to an equivalent base-2 number.

Solution

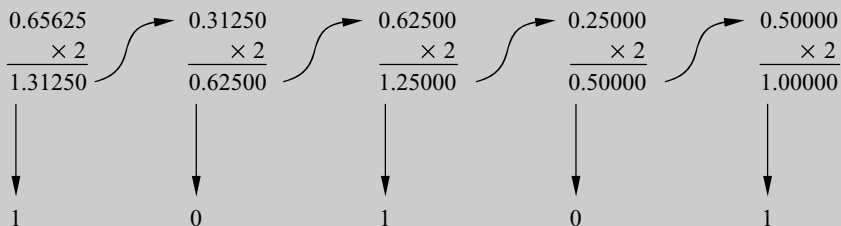


Thus, $(13)_{10} = (1101)_2$

Example 2.5

Convert $(0.65625)_{10}$ to an equivalent base-2 number.

Solution



Thus, $(0.65625)_{10} = (0.10101)_2$

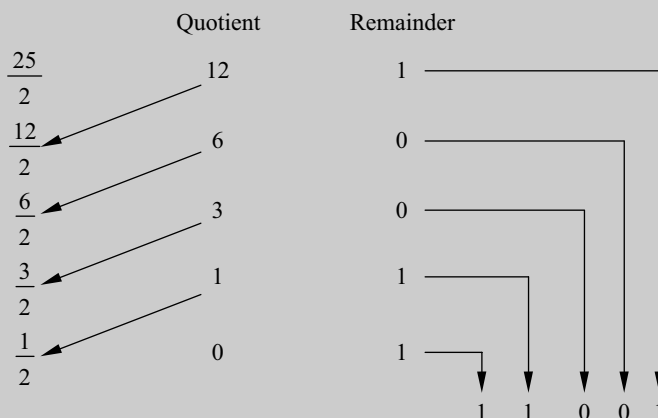
Example 2.6

Express the following decimal numbers in the binary form:

- (a) 25.5 (b) 10.625 (c) 0.6875

Solution

(a) *Integer part*



Therefore, $(25)_{10} = (11001)_2$

Fractional part

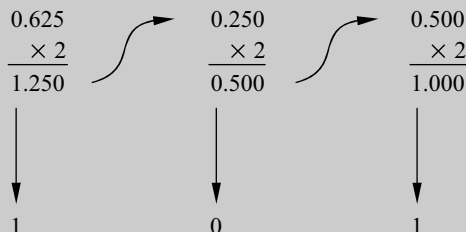
$$\begin{array}{r} 0.5 \\ \times 2 \\ \hline 1.0 \\ \downarrow \\ 1 \end{array}$$

i.e., $(0.5)_{10} = (0.1)_2$

Therefore, $(25.5)_{10} = (11001.1)_2$

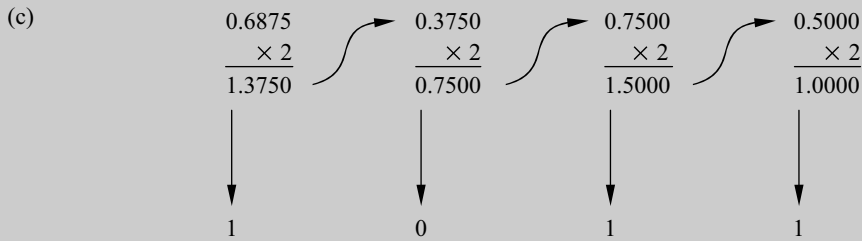
(b) *Integer part* $(10)_{10} = (1010)_2$

Fractional part



i.e., $(0.625)_{10} = (0.101)_2$

Therefore, $(10.625)_{10} = (1010.101)_2$



Therefore, $(0.6875)_{10} = (0.1011)_2$

2.4 SIGNED BINARY NUMBERS

2.4.1 Sign-Magnitude Representation

In the decimal number system a plus (+) sign is used to denote a positive number and a minus (−) sign for denoting a negative number. The plus sign is usually dropped, and the absence of any sign means that the number has positive value. This representation of numbers is known as *signed number*. As is well known, digital circuits can understand only two symbols, 0 and 1; therefore, we must use the same symbols to indicate the sign of the number also. Normally, an additional bit is used as the *sign bit* and it is placed as the most significant bit. A 0 is used to represent a positive number and a 1 to represent a negative number. For example, an 8-bit signed number 01000100 represents a positive number and its value (magnitude) is $(1000100)_2 = (68)_{10}$. The left most 0 (MSB) indicates that the number is positive. On the other hand, in the signed binary form, 11000100 represents a negative number with magnitude $(1000100)_2 = (68)_{10}$. The 1 in the left most position (MSB) indicates that the number is negative and the other seven bits give its magnitude. This kind of representation for signed numbers is known as *sign-magnitude representation*. The user must take care to see the representation used while dealing with the binary numbers.

Example 2.7

Find the decimal equivalent of the following binary numbers assuming sign-magnitude representation of the binary numbers.

- (a) 101100 (b) 001000 (c) 0111 (d) 1111

Solution

- (a) Sign bit is 1, which means the number is negative.

$$\therefore \begin{array}{l} \text{Magnitude} = 01100 = (12)_{10} \\ (101100)_2 = (-12)_{10} \end{array}$$

- (b) Sign bit is 0, which means the number is positive.

$$\therefore \begin{array}{l} \text{Magnitude} = 01000 = 8 \\ (001000)_2 = (+8)_{10} \end{array}$$

(c) $(0111)_2 = (+7)_2$

(d) $(1111)_2 = (-7)_2$

2.4.2 One's Complement Representation

In a binary number, if each 1 is replaced by 0 and each 0 by 1, the resulting number is known as the *one's complement* of the first number. In fact, both the numbers are complement of each other. If one of these numbers is positive, then the other number will be negative with the same magnitude and vice-versa. For example, $(0101)_2$ represents $(+5)_{10}$, whereas $(1010)_2$ represents $(-5)_{10}$ in this representation. This method is widely used for representing signed numbers. In this representation also, MSB is 0 for positive numbers and 1 for negative numbers.

Example 2.8

Find the one's complement of the following binary numbers.

- (a) 0100111001 (b) 11011010

Solution

- (a) 1011000110 (b) 00100101

Example 2.9

Represent the following numbers in one's complement form.

- (a) +7 and -7 (b) +8 and -8 (c) +15 and -15

Solution

In one's complement representation,

- (a) $(+7)_{10} = (0111)_2$ and $(-7)_{10} = (1000)_2$
 (b) $(+8)_{10} = (01000)_2$ and $(-8)_{10} = (10111)_2$
 (c) $(+15)_{10} = (01111)_2$ and $(-15)_{10} = (10000)_2$

From the above examples, it can be observed that for an n -bit number, the maximum positive number which can be represented in 1's complement representation is $(2^{n-1} - 1)$ and the maximum negative number is $-(2^{n-1} - 1)$.

2.4.3 Two's Complement Representation

If 1 is added to 1's complement of a binary number, the resulting number is known as the *two's complement* of the binary number. For example, 2's complement of 0101 is 1011. Since 0101 represents $(+5)_{10}$, therefore, 1011 represents $(-5)_{10}$ in 2's complement representation. In this representation also, if the MSB is 0 the number is positive, whereas if the MSB is 1 the number is negative. For an n -bit number, the maximum positive number which can be represented in 2's complement form is $(2^{n-1} - 1)$ and the maximum negative number is -2^{n-1} . Table 2.3 gives sign-magnitude, 1's and 2's complement numbers represented by 4-bit binary numbers. From the table, it is observed that the maximum positive number is $0111 = +7$, whereas the maximum negative number is $1000 = -8$ using four bits in 2's complement format.

It is also observed that the 2's complement of the 2's complement of a number is the number itself.

Table 2.3 *Sign-Magnitude, 1's and 2's Complement Representation Using Four Bits*

Decimal number	Binary number		
	Sign-magnitude	One's complement	Two's complement
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	0101	0101
6	0110	0110	0110
7	0111	0111	0111
-8	—	—	1000
-7	1111	1000	1001
-6	1110	1001	1010
-5	1101	1010	1011
-4	1100	1011	1100
-3	1011	1100	1101
-2	1010	1101	1110
-1	1001	1110	1111
-0	1000	1111	—

Example 2.10

Find the 2's complement of the numbers:

- (i) 01001110 (ii) 00110101

Solution

(i) Number	0 1 0 0 1 1 1 0
1's complement	1 0 1 1 0 0 0 1
Add 1	<div> <div></div> <div>1</div> </div>
	1 0 1 1 0 0 1 0
(ii) Number	0 0 1 1 0 1 0 1
1's complement	1 1 0 0 1 0 1 0
Add 1	<div> <div></div> <div>1</div> </div>
	1 1 0 0 1 0 1 1

From the above example, we observe the following:

1. If the LSB of the number is 1, its 2's complement is obtained by changing each 0 to 1 and 1 to 0 except the least-significant bit.
2. If the LSB of the number is 0, its 2's complement is obtained by scanning the number from the LSB to MSB bit by bit and retaining the bits as they are up to and including the occurrence of the first 1 and complement all other bits.

Example 2.11

Find two's complement of the numbers:

- (i) 01100100 (iii) 11011000
(ii) 10010010 (iv) 01100111

Solution

Using the rules of conversion given above, we obtain

		↓ ↓ ↓
(i) Number	0 1 1 0 0 1 0 0	
2's Complement	1 0 0 1 1 1 0 0	
		↓ ↓ ↓
(ii) Number	1 0 0 1 0 0 1 0	
2's Complement	0 1 1 0 1 1 1 0	
		↓ ↓ ↓ ↓
(iii) Number	1 1 0 1 1 0 0 0	
2's Complement	0 0 1 0 1 0 0 0	
		↓
(iv) Number	0 1 1 0 0 1 1 1	
2's Complement	1 0 0 1 1 0 0 1	

Example 2.12

Represent $(-17)_{10}$ in

- (i) Sign-magnitude,
- (ii) one's complement,
- (iii) two's complement representation.

Solution

The minimum number of bits required to represent $(+17)_{10}$ in signed number format is six.

$$\therefore (+17)_{10} = (010001)_2$$

Therefore, $(-17)_{10}$ is represented by

- (i) 110001 in sign-magnitude representation.
- (ii) 101110 in 1's complement representation.
- (iii) 101111 in 2's complement representation.

2.5 BINARY ARITHMETIC

We all are familiar with the arithmetic operations such as addition, subtraction, multiplication, and division of decimal numbers. Similar operations can be performed on binary numbers; infact, binary arithmetic is much simpler than decimal arithmetic because here only two digits, 0 and 1 are involved.

2.5.1 Binary Addition

The rules of binary addition are given in Table 2.4.

Table 2.4 Rules of Binary Addition

Augend	Addend	Sum	Carry	Result
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	10

In the first three rows above, there is no *carry*, that is, carry = 0, whereas in the fourth row a carry is produced (since the largest digit possible is 1), that is, carry = 1, and similar to decimal addition it is added to the next higher binary position.

Example 2.13

Add the binary numbers:

- (i) 1011 and 1100 (ii) 0101 and 1111

Solution

$$\begin{array}{r}
 \text{(i)} \quad \begin{array}{r}
 1011 \\
 (+) 1100 \\
 \hline
 10011 \\
 \uparrow \\
 \text{carry}
 \end{array}
 \qquad
 \text{(ii)} \quad \begin{array}{r}
 (1)0(1)1(1) \leftarrow \text{carry} \\
 0101 \\
 (+) 1111 \\
 \hline
 10101 \\
 \uparrow \\
 \text{carry}
 \end{array}
 \end{array}$$

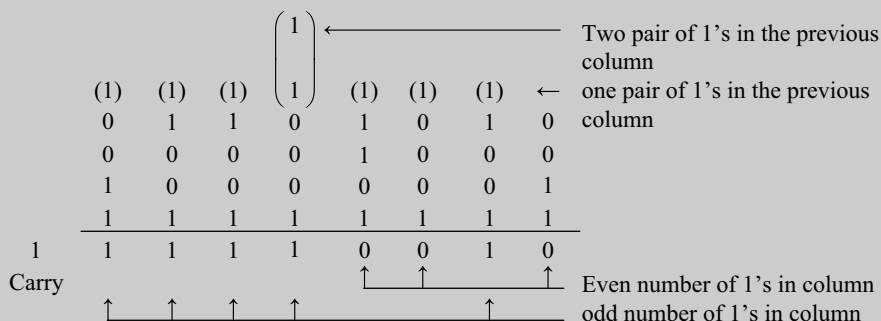
Example 2.14

Add the binary numbers:

```

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

```

Solution

∴ The sum = 1 1 1 1 1 0 0 1 0

From the above example, we observe the following:

- if the number of 1's to be added in a column is even then the sum bit is 0, and if the number of 1's to be added in a column is odd then the sum bit is 1.
- Every pair of 1's in a column produces a carry (1) to be added to the next higher bit column.

2.5.2 Binary Subtraction

The rules of binary subtraction are given in Table 2.5.

Table 2.5 Rules of Binary Subtraction

Minuend	Subtrahend	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Except in the second row above, the *borrow* = 0. When the *borrow* = 1, as in the second row, this is to be subtracted from the next higher binary bit as it is done in decimal subtraction.

Example 2.15

Perform the following subtraction:

$$\begin{array}{r} 1011 \\ - 0110 \\ \hline \end{array}$$

Solution

				Column 4
				3
				2
				1
	1	0	1	1
(-)	0	1	1	0
	0	1	0	1
				Minuend
				Subtrahend
				Difference

Here, in columns 1 and 2, borrow = 0 and in column 3 it is 1. Therefore, in column 4 first subtract 0 from 1 and from this result obtained subtract the borrow bit.

2.5.3 Binary Multiplication

Binary multiplication is similar to decimal multiplication. In binary, each partial product is either zero (multiplication by 0) or exactly same as the multiplicand (multiplication by 1). An example of binary multiplication is given below:

Example 2.16

Multiply 1001 by 1101.

Solution

1 0 0 1		Multiplicand
× 1 1 0 1		Multiplier
1 0 0 1	I	Partial Products
0 0 0 0	II	
1 0 0 1	III	
1 0 0 1	IV	
1 1 1 0 1 0 1		Final Product

In a digital circuit, the multiplication operation is performed by repeated additions of all partial products to obtain the final product.

2.5.4 Binary Division

Binary division is obtained using the same procedure as decimal division. An example of binary division is given below:

Example 2.17

Divide 1 1 1 0 1 0 1 by 1 0 0 1.

Solution

$$\begin{array}{r}
 \text{Divisor} \rightarrow 1001 \overline{) 1110101} \\
 \underline{1001} \\
 1011 \\
 \underline{1001} \\
 001001 \\
 \underline{1001} \\
 0000
 \end{array}
 \begin{array}{l}
 \leftarrow \text{Quotient} \\
 \leftarrow \text{Dividend}
 \end{array}$$

Ans: 1101

2.6 2'S COMPLEMENT ARITHMETIC

Digital circuits are used for performing binary arithmetic operations. It is possible to use the circuits designed for binary addition to perform the binary subtraction also if we can change the problem of subtraction to that of an addition. This concept eliminates the need of additional circuits for subtraction, rather the same adder circuits are used for both the operations. This makes design of arithmetic circuits very convenient and cheaper. For this purpose, 2's complement representation discussed in Section 2.4.3 is used.

2.6.1 Subtraction Using 2's Complement

Binary subtraction can be performed by adding the 2's complement of the subtrahend to the minuend. If a final carry is generated, discard the carry and the answer is given by the remaining bits which is positive (the minuend is greater than the subtrahend). If the final carry is 0, the answer is negative (the minuend is smaller than the subtrahend) and is in 2's complement form.

Example 2.18

Perform binary subtraction using 2's complement representation of negative numbers.

Solution

$$\begin{array}{rcl}
 \text{(i)} & \begin{array}{r} 7 \\ -5 \\ +2 \end{array} & \Rightarrow \begin{array}{r} 0111 \\ (+) 1011 \\ \hline 10010 \end{array} \\
 & & \begin{array}{l} \text{Minuend} \\ \text{2's complement of subtrahend} \\ \text{Discard final carry} \end{array}
 \end{array}$$

The answer is 0 0 1 0 equivalent to $(+2)_{10}$.

$$\begin{array}{rcl}
 \text{(ii)} & 5 & \longrightarrow & \begin{array}{r} 0\ 1\ 0\ 1 \\ (+) \ 1\ 0\ 0\ 1 \\ \hline 1\ 1\ 1\ 0 \end{array} & \begin{array}{l} \text{Minuend} \\ \text{2's complement of subtrahend} \end{array} \\
 & \begin{array}{r} -7 \\ \hline -2 \end{array} & & &
 \end{array}$$

The final carry = 0. Therefore, the answer is negative and is in 2's complement form. 2's complement of 1110 = 0010

Therefore, the answer is $(-2)_{10}$.

2.6.2 Addition/Subtraction in 2's Complement Representation

The addition/subtraction of signed binary numbers can most conveniently be performed using 2's complement representation of both the operands. This is the method most commonly used when these operations are performed using digital circuits and microprocessors.

Example 2.19

Perform the following operations using 2's complement method:

- (i) $48 - 23$ (ii) $23 - 48$ (iii) $48 - (-23)$ (iv) $-48 - 23$

Use 8-bit representation of numbers.

Solution

- (i) 2's complement representation of $+48 = 00110000$

2's complement representation of $-23 = 11101001$

$48 + (-23)$

$$\begin{array}{rcl}
 \longrightarrow & \begin{array}{r} 48 \\ + (-23) \\ \hline +25 \end{array} & \longrightarrow & \begin{array}{r} 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0 \\ (+) \ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1 \\ \hline 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1 \end{array} & = +25 \\
 & & & \uparrow & \\
 & & & \text{Discard Carry} &
 \end{array}$$

- (ii) 2's complement representation of $+23 = 00010111$

2's complement representation of $-48 = 11010000$

$23 - 48 = 23 + (-48)$

$$\begin{array}{rcl}
 \longrightarrow & \begin{array}{r} 23 \\ + (-48) \\ \hline -25 \end{array} & \longrightarrow & \begin{array}{r} 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1 \\ (+) \ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0 \\ \hline 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1 \end{array} & \longrightarrow & -25
 \end{array}$$

- (iii) $48 - (-23) = 48 + 23$

$$\begin{array}{rcl}
 \longrightarrow & \begin{array}{r} 48 \\ + (23) \\ \hline +71 \end{array} & \longrightarrow & \begin{array}{r} 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0 \\ (+) \ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1 \\ \hline 0\ 1\ 0\ 0\ 0\ 1\ 1\ 1 \end{array} & \longrightarrow & +71
 \end{array}$$

$$(iv) -48 - 23 = (-48) + (-23)$$

$$\begin{array}{r}
 \begin{array}{c} -48 \\ + (-23) \\ \hline -71 \end{array}
 \quad \Rightarrow \quad
 \begin{array}{r}
 \begin{array}{cccccccc} & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ (+) & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{array} \\
 \uparrow \\
 \text{Carry be ignored}
 \end{array}
 \quad \Rightarrow \quad -71
 \end{array}$$

From the above example, we observe the following:

- If the two operands are of the opposite sign, the result is to be obtained by the rule of subtraction using 2's complement (Sec. 2.6.1)
- If the two operands are of the same sign, the sign bit of the result (MSB) is to be compared with the sign bit of the operands. In case the sign bits are same, the result is correct and is in 2's complement form. If the sign bits are not same there is a problem of *overflow*, i.e. the result can not be accommodated using eight bits and the result is to be interpreted suitably. The result in this case will consist of nine bits, i.e. carry and eight bits, and the carry bit will give the sign of the number.

2.7 OCTAL NUMBER SYSTEM

The number system with base (or radix) eight is known as the *octal number system*. In this system, eight symbols, 0, 1, 2, 3, 4, 5, 6, and 7 are used to represent numbers. Similar to decimal and binary number systems, it is also a positional system and has, in general, two parts: integer and fractional, set apart by a radix (octal) point (.). Any number can be expressed in the form of Eq. (2.1) with $b = 8$ and $0 \leq (d_i \text{ or } d_{-j}) \leq 7$. The weights assigned to the various positions are given in Table 2.1. For example, $(6327.4051)_8$ is an octal number.

2.7.1 Octal-to-Decimal Conversion

Any octal number can be converted into its equivalent decimal number using the weights assigned to each octal digit position as given in Table 2.1.

Example 2.20

Convert $(6327.4051)_8$ into its equivalent decimal number.

Solution

Using the weights given in Table 2.1, we obtain

$$\begin{aligned}
 (6327.4051)_8 &= 6 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} \\
 &\quad + 0 \times 8^{-2} + 5 \times 8^{-3} + 1 \times 8^{-4} \\
 &= 3072 + 192 + 16 + 7 + \frac{4}{8} + 0 + \frac{5}{512} + \frac{1}{4096} \\
 &= (3287.5100098)_{10}
 \end{aligned}$$

$$\text{Thus, } (6327.4051)_8 = (3287.5100098)_{10}$$

2.7.2 Decimal-to-Octal Conversion

The conversion from decimal to octal (base-10 to base-8) is similar to the conversion procedure for base-10 to base-2 conversion. The only difference is that number 8 is used in place of 2 for division in the case of integers and for multiplication in the case of fractional numbers.

Example 2.21

- (a) Convert $(247)_{10}$ into octal
 (b) Convert $(0.6875)_{10}$ into octal
 (c) Convert $(3287.510098)_{10}$ into octal

Solution

(a)

	Quotient	Remainder
$\begin{array}{r} 247 \\ 8 \end{array}$	30	7
$\begin{array}{r} 30 \\ 8 \end{array}$	3	6
$\begin{array}{r} 3 \\ 8 \end{array}$	0	3

3 6 7

Thus, $(247)_{10} = (367)_8$

(b)

$\begin{array}{r} 0.6875 \\ \times 8 \\ \hline 5.5000 \\ \downarrow \\ 5 \end{array}$	$\begin{array}{r} 0.5000 \\ \times 8 \\ \hline 4.0000 \\ \downarrow \\ 4 \end{array}$
---	---

Thus, $(0.6875)_{10} = (0.54)_8$

(c) Integer part:

	Quotient	Remainder
$3287/8$	410	7
$410/8$	51	2
$51/8$	6	3
$6/8$	0	6

6 3 2 7

Thus, $(3287)_{10} = (6327)_8$

Fractional part:

$\begin{array}{r} 0.510098 \\ \times 8 \\ \hline 4.0800784 \\ \downarrow \\ 4 \end{array}$	$\begin{array}{r} 0.0800784 \\ \times 8 \\ \hline 0.6406272 \\ \downarrow \\ 0 \end{array}$	$\begin{array}{r} 0.6406272 \\ \times 8 \\ \hline 5.1250176 \\ \downarrow \\ 5 \end{array}$	$\begin{array}{r} 0.1250176 \\ \times 8 \\ \hline 1.0001408 \\ \downarrow \\ 1 \end{array}$
--	---	---	---

Thus $(0.510098)_{10} \approx (0.4051)_8$

Therefore, $(3287.510098)_{10} = (6327.4051)_8$

From the above examples we observe that the conversion for fractional numbers may not be exact. In general, an approximate equivalent can be determined by terminating the process of multiplication by eight at the desired point.

2.7.3 Octal-to-Binary Conversion

Octal numbers can be converted into equivalent binary numbers by replacing each octal digit by its 3-bit equivalent binary. Table 2.6 gives octal numbers and their binary equivalents for decimal numbers 0 to 15.

Table 2.6 *Binary and Decimal Equivalents of Octal Numbers*

Octal	Decimal	Binary
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
10	8	001000
11	9	001001
12	10	001010
13	11	001011
14	12	001100
15	13	001101
16	14	001110
17	15	001111

Example 2.22

Convert $(736)_8$ into an equivalent binary number.

Solution

From Table 2.6, we observe the binary equivalents of 7, 3 and 6 as 111, 011, and 110, respectively. Therefore, $(736)_8 = (111\ 011\ 110)_2$.

2.7.4 Binary-to-Octal Conversion

Binary numbers can be converted into equivalent octal numbers by making groups of three bits starting from LSB and moving towards MSB for integer part of the number and then replacing each group of three bits

by its octal representation. For fractional part, the groupings of three bits are made starting from the binary point.

Example 2.23

Convert $(1001110)_2$ to its octal equivalent.

Solution

$$\begin{aligned}(1001110)_2 &= (\underline{001} \ 001 \ 110)_2 \\ &= (1 \ 1 \ 6)_8 \\ &= (116)_8\end{aligned}$$

Example 2.24

Convert $(0.10100110)_2$ to its equivalent octal number.

Solution

$$\begin{aligned}(0.10100110)_2 &= (0.101 \ 001 \ 100)_2 \\ &= (0.5 \ 1 \ 4)_8 \\ &= (0.514)_8\end{aligned}$$

Example 2.25

Convert the following binary numbers to octal numbers

- (a) 11001110001.000101111001
- (b) 1011011110.11001010011
- (c) 111110001.10011001101

Solution

- (a) $011 \ 001 \ 110 \ 001.000 \ 101 \ 111 \ 001 = (3161.0571)_8$
- (b) $001 \ 011 \ 011 \ 110.110 \ 010 \ 100 \ 110 = (1336.6246)_8$
- (c) $111 \ 110 \ 001.100 \ 110 \ 011 \ 010 = (761.4632)_8$

From the above examples we observe that in forming the 3-bit groupings, 0's may be required to complete the first (most significant digit) group in the integer part and the last (least significant digit) group in the fractional part.

2.7.5 Octal Arithmetic

Octal arithmetic rules are similar to the decimal or binary arithmetic. Normally, we are not interested in performing octal arithmetic operations using octal representation of numbers. This number system is normally

used to enter long strings of binary data into a digital system like a microcomputer. This makes the task of entering binary data in a microcomputer easier. Arithmetic operations can be performed by converting the octal numbers to binary numbers and then using the rules of binary arithmetic.

Example 2.26

Add $(23)_8$ and $(67)_8$.

Solution

$$\begin{array}{r} 23 = 010011 \\ (+)67 = 110111 \\ \hline (112)_8 = 1001\ 010 \end{array}$$

Example 2.27

Subtract (a) $(37)_8$ from $(53)_8$
(b) $(75)_8$ from $(26)_8$

Solution

Using 8-bit representation,

$$\begin{array}{rcl} \text{(a)} & (53)_8 = & 00101011 \\ & - (37)_8 = (+) & 11100001 \quad \text{Two's complement of } (37)_8 \\ \hline & (14)_8 = & 100001100 \\ \text{Discard carry } \xrightarrow{\quad} & & \\ \text{(b)} & (26)_8 = & 00010110 \\ & - (75)_8 = (+) & 11000011 \quad \text{Two's complement of } (75)_8 \\ \hline & - (47)_8 = & 11011001 \quad \text{Two's complement of result} \end{array}$$

$$\text{Two's complement of } 11011001 = 00\ 100\ 111 = (47)_8$$

Multiplication and division can also be performed using the binary representation of octal numbers and then making use of multiplication and division rules of binary numbers.

2.7.6 Applications of Octal Number System

In digital systems, binary numbers are required to be entered and certain results or status signals are required to be displayed. It is highly inconvenient to handle long strings of binary numbers. It may cause errors also. Therefore, octal numbers are used for entering the binary data and displaying certain informations. Therefore, the knowledge of octal number system is very important for the efficient use of microprocessors and other digital circuits. For example, the binary number 01111110 can easily be remembered as 376 and can be

entered as 376 using keys. Since digital circuits can process only zeros and ones, the octal numbers have to be converted into binary form using special circuits known as octal-to-binary converters before being processed by the digital circuits.

2.8 HEXADECIMAL NUMBER SYSTEM

Hexadecimal number system is very popular in computer uses. The base for hexadecimal number system is 16 which requires 16 distinct symbols to represent the numbers. These are numerals 0 through 9 and alphabets A through F. Since numeric digits and alphabets both are used to represent the digits in the hexadecimal number system, therefore, this is an alphanumeric number system. Table 2.7 gives hexadecimal numbers with their binary equivalents for decimal numbers 0 through 15. From the table, it is observed that there are 16 combinations of 4-bit binary numbers and sets of 4-bit binary numbers can be entered in the computer in the form of hexadecimal (hex.) digits. These numbers are required to be converted into binary representation, using hexadecimal-to-binary converter circuits before these can be processed by the digital circuits.

Table 2.7 *Binary and Decimal Equivalents of Hexadecimal Numbers*

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

2.8.1 Hexadecimal-to-Decimal Conversion

Using Eq. (2.1), hexadecimal numbers can be converted to their equivalent decimal numbers.

Example 2.28

Obtain decimal equivalent of hexadecimal number $(3A.2F)_{16}$

Solution

Using Eq. (2.1),

$$\begin{aligned}(3A.2F)_{16} &= 3 \times 16^1 + 10 \times 16^0 + 2 \times 16^{-1} + 15 \times 16^{-2} \\ &= 48 + 10 + \frac{2}{16} + \frac{15}{16^2} \\ &= (58.1836)_{10}\end{aligned}$$

The fractional part may not be an exact equivalent and therefore, may give a small error.

2.8.2 Decimal-to-Hexadecimal Conversion

For conversion from decimal to hexadecimal, the procedure used in binary as well as octal systems is applicable, using 16 as the dividing (for integer part) and multiplying (for fractional part) factor.

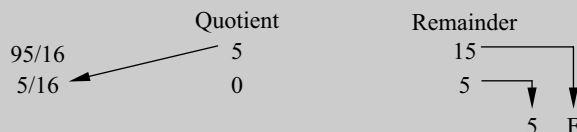
Example 2.29

Convert the following decimal numbers into hexadecimal numbers.

- (a) 95.5 (b) 675.625

Solution

(a) *Integer part*



Thus, $(95)_{10} = (5F)_{16}$

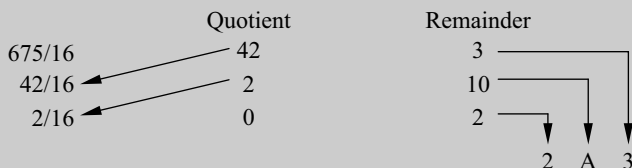
Fractional part

$$\begin{array}{r} 0.5 \\ \times 16 \\ \hline 8.0 \\ \downarrow \\ 8 \end{array}$$

Thus, $(0.5)_{10} = (0.8)_{16}$

Therefore, $(95.5)_{10} = (5F.8)_{16}$

(b) *Integer part*



Thus, $(675)_{10} = (2A3)_{16}$

Fractional part

$$\begin{array}{r}
 0.625 \\
 \times 16 \\
 \hline
 10.000 \\
 \downarrow \\
 A
 \end{array}$$

Thus, $(0.625)_{10} = (0.A)_{16}$

Therefore, $(675.625)_{10} = (2A3.A)_{16}$

2.8.3 Hexadecimal-to-Binary Conversion

Hexadecimal numbers can be converted into equivalent binary numbers by replacing each hex digit by its equivalent 4-bit binary number.

Example 2.30

Convert $(2F9A)_{16}$ to equivalent binary number.

Solution

Using Table 2.7, find the binary equivalent of each hex digit.

$$\begin{aligned}
 (2F9A)_{16} &= (0010 \ 1111 \ 1001 \ 1010)_2 \\
 &= (0010111110011010)_2
 \end{aligned}$$

2.8.4 Binary-to-Hexadecimal Conversion

Binary numbers can be converted into the equivalent hexadecimal numbers by making groups of four bits starting from LSB and moving towards MSB for integer part and then replacing each group of four bits by its hexadecimal representation.

For the fractional part, the above procedure is repeated starting from the bit next to the binary point and moving towards the right.

Example 2.31

Convert the following binary numbers to their equivalent hex numbers.

- (a) 10100110101111
(b) 0.00011110101101

Solution

$$\begin{aligned} \text{(a)} \quad (10100110101111)_2 &= \underbrace{0010}_2 \underbrace{1001}_9 \underbrace{1010}_A \underbrace{1111}_F \\ \therefore (10100110101111)_2 &= (29AF)_{16} \\ \text{(b)} \quad (0.00011110101101)_2 &= \underbrace{0.0001}_1 \underbrace{1110}_E \underbrace{1011}_B \underbrace{0100}_4 \\ \therefore (0.00011110101101)_2 &= (0.1EB4)_{16} \end{aligned}$$

Example 2.32

Convert the binary numbers of Example 2.25 to hexadecimal numbers.

Solution

$$\begin{aligned} \text{(a)} \quad 110\ 0111\ 0001.0001\ 0111\ 1001 &= (671.179)_{16} \\ \text{(b)} \quad 10\ 1101\ 1110.1100\ 1010\ 011 &= (2DE.CA6)_{16} \\ \text{(c)} \quad 1\ 1111\ 0001.1001\ 1001\ 101 &= (1F1.99A)_{16} \end{aligned}$$

From the above examples, we observe that in forming the 4-bit groupings 0's may be required to complete the first (most significant digit) group in the integer part and the last (least significant digit) group in the fractional part.

2.8.5 Conversion from Hex-to-Octal and Vice-Versa

Hexadecimal numbers can be converted to equivalent octal numbers and octal numbers can be converted to equivalent hex numbers by converting the hex/octal number to equivalent binary and then to octal/hex, respectively.

Example 2.33

Convert the following hex numbers to octal numbers.

(a) $A72E$ (b) $0.BF85$

Solution

$$\begin{aligned}
 \text{(a)} \quad (A72E)_{16} &= (1010 \ 0111 \ 0010 \ 1110)_2 \\
 &= \underbrace{001}_{12} \ \underbrace{010}_{3} \ \underbrace{011}_{4} \ \underbrace{100}_{5} \ \underbrace{101}_{6} \ \underbrace{110}_{7} \\
 &= (123456)_8 \\
 \text{(b)} \quad (0.BF85)_{16} &= (0.1011 \ 1111 \ 1000 \ 0101)_2 \\
 &= 0.\underbrace{101}_{5} \ \underbrace{111}_{7} \ \underbrace{111}_{4} \ \underbrace{000}_{0} \ \underbrace{010}_{2} \ \underbrace{100}_{6} \\
 &= (0.577024)_8
 \end{aligned}$$

Example 2.34

Convert $(247.36)_8$ to equivalent hex number.

Solution

$$\begin{aligned}
 (247.36)_8 &= (010 \ 100 \ 111.011 \ 110)_2 \\
 &= (0 \ \underbrace{1010}_{A} \ \underbrace{0111}_{7} \cdot \underbrace{0111}_{7} \ \underbrace{1000}_{8})_2 \\
 &= (A7.78)_{16}
 \end{aligned}$$

2.8.6 Hexadecimal Arithmetic

The rules for arithmetic operations with hexadecimal numbers are similar to the rules for decimal, octal, and binary systems. The information can be handled only in binary form in a digital circuit and it is easier to enter the information using hexadecimal number system. Since arithmetic operations are performed by the digital circuits on binary numbers, therefore hexadecimal numbers are to be first converted into binary numbers. Arithmetic operations will become clear from the following examples.

Example 2.35

Add $(7F)_{16}$ and $(BA)_{16}$

Solution


$$\begin{array}{r}
 7F = \quad 01111111 \\
 (+) BA = \quad 10111010 \\
 \hline
 (139)_{16} = 100111001
 \end{array}$$

Example 2.36

Subtract (a) $(5C)_{16}$ from $(3F)_{16}$
 (b) $(7A)_{16}$ from $(C0)_{16}$

Solution

(a) $3F = 00111111$
 $-5C = (+)10100100$ Two's complement of $(5C)_{16}$
 $-1D = 11100011$ Two's complement of result
 Two's complement of $11100011 = 0001\ 1101 = (1D)_{16}$

(b) $C0 = 11000000$
 $-7A = (+)10000110$ Two's complement of $(7A)_{16}$
 $46 = 101000110$
 Discard carry 

Multiplication and division can also be performed using the binary representation of hexadecimal numbers and then making use of multiplication and division rules of binary numbers.

2.9 CODES

Computers and other digital circuits process data in the binary format. Various binary codes are used to represent data which may be numeric, alphabets or special characters. Although, in every code used the information is represented in binary form, the interpretation of this binary information is possible only if the code in which this information is available is known. For example, the binary number 1000001 represents 65 (decimal) in straight binary, 41 (decimal) in BCD and alphabet *A* in ASCII code. A user must be very careful about the code being used while interpreting information available in the binary format. Codes are also used for error detection and error correction in digital systems.

Some of the commonly used codes are given below.

2.9.1 Straight Binary Code

This is used to represent numbers using natural (or straight) binary form as discussed in Section 2.3. Various arithmetic operations can be performed in this form. Binary codes for decimal numbers 0 to 15 are given in Table 2.8. It is a weighted code since a weight is assigned to every position.

2.9.2 Natural BCD Code

In this code, decimal digits 0 through 9 are represented (coded) by their natural binary equivalents using four bits and each decimal digit of a decimal number is represented by this four bit code individually. For example, $(23)_{10}$ is represented by 0010 0011 using BCD code, rather than $(10111)_2$. From this it is observed that it requires more number of bits to code a decimal number using BCD code than using the straight binary code. However, inspite of this disadvantage it is very convenient and useful code for input and output operations in digital systems.

This code is also known as 8-4-2-1 code or simply BCD code. 8, 4, 2, and 1 are the weights of the four bits of the binary code of each decimal digit similar to straight binary number system. Therefore, this is

a *weighted* code and arithmetic operations can be performed using this code, which will be discussed in Chapter 6. BCD codes for decimal digits 0 through 9 are given in Table 2.8.

Table 2.8 *Various Binary Codes*

Decimal Number	Binary				BCD				Excess-3				Gray			
	B_3	B_2	B_1	B_0	D	C	B	A	E_3	E_2	E_1	E_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
1	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1
2	0	0	1	0	0	0	1	0	0	1	0	1	0	0	1	1
3	0	0	1	1	0	0	1	1	0	1	1	0	0	0	1	0
4	0	1	0	0	0	1	0	0	0	1	1	1	0	1	1	0
5	0	1	0	1	0	1	0	1	1	0	0	0	0	1	1	1
6	0	1	1	0	0	1	1	0	1	0	0	1	0	1	0	1
7	0	1	1	1	0	1	1	1	1	0	1	0	0	1	0	0
8	1	0	0	0	1	0	0	0	1	0	1	1	1	1	0	0
9	1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	1
10	1	0	1	0									1	1	1	1
11	1	0	1	1									1	1	1	0
12	1	1	0	0									1	0	1	0
13	1	1	0	1									1	0	1	1
14	1	1	1	0									1	0	0	1
15	1	1	1	1									1	0	0	0

2.9.3 Excess-3 Code

This is another form of BCD code, in which each decimal digit is coded into a 4-bit binary code. The code for each decimal digit is obtained by adding decimal 3 to the natural BCD code of the digit. For example, decimal 2 is coded as $0010 + 0011 = 0101$ in Excess-3 code. It is not a weighted code. This code is a self-complementing code, which means 1's complement of the coded number yields 9's complement of the number itself. For example, Excess-3 code of decimal 2 is 0101, its 1's complement is 1010 which is Excess-3 code for decimal 7, which is 9's complement of 2. The self complementing property of this code helps considerably in performing subtraction operation in digital systems. Excess-3 codes for decimal digits 0 through 9 are given in Table 2.8.

2.9.4 Gray code

It is a very useful code in which a decimal number is represented in binary form in such a way so that each Gray-code number differs from the preceding and the succeeding number by a single bit. For example, the Gray code for decimal number 5 is 0111 and for 6 is 0101. These two codes differ by only one bit position (third from the left). This code is used extensively for shaft encoders because of this property. It is not a weighted code. The Gray code is a reflected code and can be constructed using this property as given below.

- (i) A 1-bit Gray code has two code words 0 and 1 representing decimal numbers 0 and 1 respectively
- (ii) An n -bit ($n \geq 2$) Gray code will have first 2^{n-1} Gray codes of $(n-1)$ -bits written in order with a leading 0 appended.
- (iii) The last 2^{n-1} Gray codes will be equal to the Gray code words of an $(n-1)$ -bit Gray code, written in reverse order (assuming a mirror placed between first 2^{n-1} and last 2^{n-1} Gray codes) with a leading 1 appended.

Example 2.37

Determine (a) 1-bit (b) 2-bit (c) 3-bit Gray codes and tabulate along with their equivalent decimal numbers.

Solution

(a) 1-bit Gray code is constructed using (i) above.

<i>Decimal number</i>	<i>Gray code</i>
0	0
1	1

(b) 2-bit Gray code is constructed using (ii) and (iii) above and Gray code of 1-bit

<i>Decimal number</i>	<i>Gray code</i>
0	0 0
1	<u>0</u> 1
2	1 1
3	1 0

(c) 3-bit Gray code is constructed using 2-bit Gray code.

<i>Decimal number</i>	<i>Gray code</i>
0	0 0 0
1	0 0 1
2	0 1 1
3	<u>0</u> 1 0
4	1 1 0
5	1 1 1
6	1 0 1
7	1 0 0

4-bit Gray code is given in Table 2.8.

2.9.5 Octal Code

It is a 3-bit binary code, in which each of the octal digits 0 through 7 is coded into 3-bit straight binary number. For example, code for octal digit 4 is 100. Using this code, octal numbers can be coded into straight binary form or the binary numbers can be represented by octal numbers as discussed in Section 2.7.

This code is used for binary inputs in digital computers, microprocessors, etc.

2.9.6 Hexadecimal Code

It is a 4-bit binary code (Section 2.8) used for input/output in digital computers, microprocessors, etc.

Example 2.38

Represent the decimal number 27 in binary form using

- | | |
|---------------------|-----------------------|
| (i) Binary Code | (iv) Gray code |
| (ii) BCD code | (v) Octal code |
| (iii) Excess-3 code | (vi) Hexadecimal code |

Solution

- (i) The decimal number is converted into straight binary form. Its value is 11011
 (ii) Each digit of the decimal number is coded using 4-bit BCD code as given below

0010 0111

- (iii) Each digit of the decimal number is coded using 4-bit Excess-3 code as given below

0101 1010

- (iv) 5 bits are required to represent 27, therefore, 5-bit Gray code is constructed and 27 is represented as 10110.

(v) $(27)_{10} = (33)_8 = 011\ 011$

(vi) $(27)_{10} = (1B)_{16} = 0001\ 1011$

Example 2.39

Represent the decimal numbers (a) 396 and (b) 4096 in binary form in

- | | |
|-----------------------------------|-----------------|
| (i) Binary code (straight binary) | (iv) Octal code |
| (ii) BCD code | (v) Hex code |
| (iii) Excess-3 code | |

Solution

- (a) (i) $396 = 110001100$
 (ii) $396 = 001110010110$
 (iii) $396 = 011011001001$
 (iv) $396 = (614)_8 = 110001100$
 (v) $396 = (18C)_{16} = 000110001100$

- (b) (i) $4096 = 1000000000000$
 (ii) $4096 = 0100000010010110$
 (iii) $4096 = 01110011\ 11001001$
 (iv) $4096 = (10000)_8 = 001\ 000\ 000\ 000\ 000$
 (v) $4096 = (1000)_{16} = 0001\ 0000\ 0000\ 0000$

2.9.7 Alphanumeric Codes

In many situations, digital systems are required to handle data that may consist of numerals, letters, and special symbols. For example, an university with thousands of students may use a digital computer to process the examination results. In this the names of the students, subjects, etc. are to be represented in the binary form. Therefore, it is necessary to have a binary code for alphabets also. If we use an n -bit binary code, we can represent 2^n elements using this code. Therefore, to represent 10 digits 0 through 9 and 26 alphabets A through Z , we need a minimum of 6 bits ($2^6 = 64$, but $2^5 = 32$ is not sufficient). One possible 6-bit alphanumeric code is given in Table 2.9. It is used in many computers to represent alphanumeric characters and symbols internally and therefore can be called *internal code*. Frequently, there is a need to represent more than 64 characters including the lower case letters and special control characters for the transmission of digital information. For this reason the following two codes are normally used:

1. Extended BCD Interchange Code (EBCDIC)
2. American Standard Code for Information Interchange (ASCII)

These are given in Tables 2.9 and 2.10 respectively.

Table 2.9 Some Alphanumeric Codes

Character	6-bit Internal code	8-bit EBCDIC code
<i>A</i>	010001	11000001
<i>B</i>	010010	11000010
<i>C</i>	010011	11000011
<i>D</i>	010100	11000100
<i>E</i>	010101	11000101
<i>F</i>	010110	11000110
<i>G</i>	010111	11000111
<i>H</i>	011000	11001000
<i>I</i>	011001	11001001
<i>J</i>	100001	11010001
<i>K</i>	100010	11010010
<i>L</i>	100011	11010011
<i>M</i>	100100	11010100
<i>N</i>	100101	11010101
<i>O</i>	100110	11010110
<i>P</i>	100111	11010111
<i>Q</i>	101000	11011000
<i>R</i>	101001	11011001

(Continued)

Table 2.9 (Continued)

Character	6-bit Internal code	8-bit EBCDIC code
<i>S</i>	110010	11100010
<i>T</i>	110011	11100011
<i>U</i>	110100	11100100
<i>V</i>	110101	11100101
<i>W</i>	110110	11100110
<i>X</i>	110111	11100111
<i>Y</i>	111000	11101000
<i>Z</i>	111001	11101001
0	000000	11110000
1	000001	11110001
2	000010	11110010
3	000011	11110011
4	000100	11110100
5	000101	11110101
6	000110	11110110
7	000111	11110111
8	001000	11111000
9	001001	11111001
blank	110000	01000000
.	011011	01001011
(111100	01001101
+	010000	01001110
\$	101011	01011011
*	101100	01011100
)	011100	01011101
—	100000	01100000
/	110001	01100001
,	111011	01101011
=	001011	01111110

Table 2.10 The ASCII Code

		b_6			0	0	0	0	1	1	1	1
			b_5		0	0	1	1	0	0	1	1
				b_4	0	1	0	1	0	1	0	1
b_3	b_2	b_1	b_0		0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	'	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	,	7	G	W	g	w
1	0	0	0	8	BS	CAN	(8	H	X	h	x
1	0	0	1	9	HT	EM)	9	I	Y	i	y
1	0	1	0	A	LF	SUB	*	:	J	Z	j	z
1	0	1	1	B	VT	ESC	+	;	K	[k	{
1	1	0	0	C	FF	FS	,	<	L	\	l	
1	1	0	1	D	CR	GS	-	=	M]	m	}
1	1	1	0	E	SO	RS	.	>	N	^	n	~
1	1	1	1	F	SI	US	/	?	O	-	o	DEL

The code is read from this table as:

$$\begin{array}{rccccccc}
 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
 H = 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 = (48)_{16} \\
 n = 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 = (6E)_{16}
 \end{array}$$

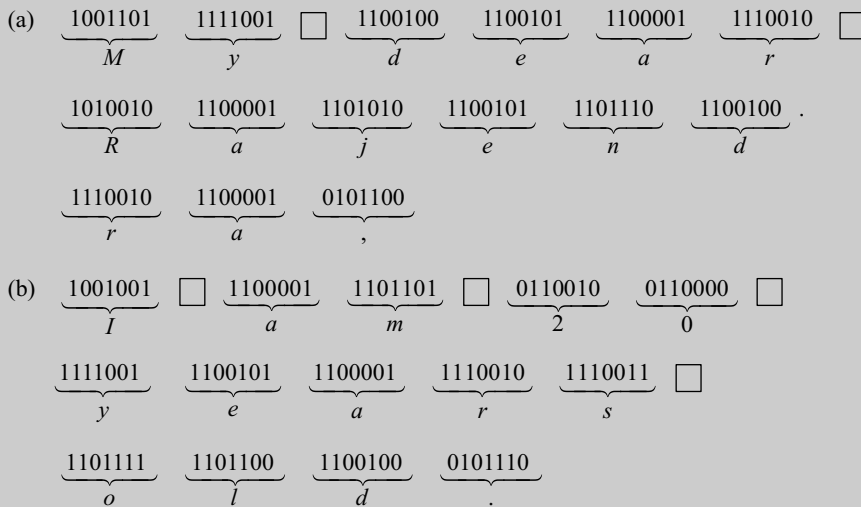
Example 2.40

Encode the following in ASCII Code:

- (a) My dear Rajendra,
 (b) I am 20 years old.

Solution

Each of the alphabet, numeral, and the special character is represented by 7-bit ASCII code given in Table 2.10.



The code for SPACE has not been added above, which is 0100000

2.10 ERROR DETECTING AND CORRECTING CODES

Digital signals are processed for performing various operations and are transmitted from one circuit or system to another circuit or system. When these binary signals are transmitted from one location (transmitter) to another location (receiver), transmission errors may occur because of electrical noise in the transmission channel. Due to transmission error a signal transmitted as a 0 may be received as a 1 or vice-versa. In complex digital systems, millions of bits per second are manipulated and it is desired to have high data integrity, or at least a violation of data integrity must be detectable.

A digital data in the form of a code word such as BCD, ASCII etc. is transmitted and there is always a finite probability of occurrence of an error in a single bit position. The probability of occurrence of error in two or more bit positions simultaneously is substantially smaller. It is desired to *detect* the error in the received data word, *locate* its bit position and *correct* it. Various codes are used for the detection and correction of error. Since the probability of simultaneous occurrence of error in two or more bit positions is negligibly small, therefore, we restrict our discussion to the detection and correction of single error, i.e. error in one bit position.

2.10.1 Error-detecting Codes

When a digital information is transmitted, it may not be received correctly by the receiver. At the receiving end it may or may not be possible to detect whether the information has been received correctly or not. Let us consider BCD code given in Table 2.8 is transmitted and the code corresponding to decimal 9, i.e. 1001 is transmitted and is received as 1011. Since 1011 is an invalid BCD code, therefore, it may be detected by the receiver. On the other hand if it is received as 0001 which is a valid BCD code for decimal 1, the receiver will interpret it as decimal 1 and the error is not detected. In general, the erroneous word received may or may not be a valid code.

To ensure that the occurrence of a single error always results in an invalid code, so as to avoid its incorrect interpretation by the receiver, the code must possess the property that the occurrence of any single

error transforms a valid code word into an invalid code word. Since for an n -bit code there are 2^n possible combinations, therefore, if it is desired to make this code an error-detecting code, only half of the possible 2^n combinations should be included to form the code. This means, if an extra bit is attached to the n -bit code word to make the number of bits $n + 1$ in such a way so as to make the number of ones in the resulting $(n + 1)$ -bit code even or odd, it will certainly be an error-detecting code.

The criterion of minimum distance of a code can also be used as a useful property of an error-detecting code. The minimum distance of a code is the smallest number of bits in which any two code words differ. A code is an error-detecting code if and only if its minimum distance is two or more.

For detection of error an extra bit known as *parity bit* is attached to each code word to make the number of ones in the code even (*even parity*) or odd (*odd parity*).

Table 2.11 shows BCD code with parity bit attached making it even parity code or odd parity code.

Table 2.11 **BCD Code with Parity Bit**

BCD code				BCD code with even parity					BCD code with odd parity				
<i>D</i>	<i>C</i>	<i>B</i>	<i>A</i>	<i>P</i>	<i>D</i>	<i>C</i>	<i>B</i>	<i>A</i>	<i>P</i>	<i>D</i>	<i>C</i>	<i>B</i>	<i>A</i>
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	1	0	0	0	1	0	0	0	0	1
0	0	1	0	1	0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	1	1	1	0	0	1	1
0	1	0	0	1	0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1	1	0	1	0	1
0	1	1	0	0	0	1	1	0	1	0	1	1	0
0	1	1	1	1	0	1	1	1	0	0	1	1	1
1	0	0	0	1	1	0	0	0	0	1	0	0	0
1	0	0	1	0	1	0	0	1	1	1	0	0	1

From Table 2.11 we observe that a parity bit (P) 0 or 1 is attached to every code word so as to make the number of ones even or odd for even and odd parity respectively. It can be easily verified that the minimum distance between any two code words with parity bit attached is two. The parity bit 1 or 0 is attached to the code to be transmitted at the transmitter end and the parity of the received $(n + 1)$ -bit word is checked at the receiving end. If there is only one error, the erroneous code is detected at the receiving end by parity check. If odd number of bits are transmitted erroneously, then also the parity check will detect the incorrect code but if there are even number of bits received incorrectly, this method will not detect error. The parity check method can only detect error in the transmitted word at the receiving end. It can not locate the bit which has changed and, therefore, the question of correction does not arise.

The parity generator and parity checker circuits have been discussed in Chapter 6.

Example 2.41

Formulate 8-bit ASCII code for Example 2.40 and represent it in hexadecimal code with

- (i) even parity
- (ii) odd parity

Solution

(i) For Example 2.40(a):

The 7-bit ASCII code for M is 1001101, It contains four ones. To make an 8-bit even parity code corresponding to this, one 0 is attached to it as the most-significant bit. Therefore, the 8-bit code will be 01001101. Its hexadecimal representation is 4D. Similarly ASCII code for y is 1111001, which contains five 1s and therefore, a 1 as MSB is attached to it for obtaining its 8-bit even parity code, i.e. 11111001. Its hex representation is F9.

In the same way each character is formulated. The complete sentence (including the spaces) in hexadecimal code is

4DF9A0E465E172A0D2E16A65EEE472E1AC

For Example 2.40 (b):

Using the above procedure, we get,

C9A0E1EDA0B230A0F965E172F3A06F6CE42E

(ii) For Example 2.40(a):

The 7-bit ASCII code for M is 1001101, when a 1 is attached to this as MSB, it becomes the 8-bit code for M. In hexadecimal code it is CD. Similarly, a 0 is attached as MSB to the 7-bit ASCII code of y to make it 8-bit with odd parity. In the same way each character is formulated. The complete sentence (including the spaces) in hexadecimal code is

CD792064E561F2205261EAE56E64F2612C

For Example 2.40 (b):

Using the above procedure, we get,

4920616D2032B02079E561F27320EFEC64AE

Example 2.42

Find out the minimum distance of

- (a) ASCII code.
- (b) 8-bit ASCII code with even parity.
- (c) 8-bit ASCII code with odd parity.

Solution

From the Table 2.10, examine the ASCII codes of various alphabets, numerals, special characters, etc. and find out the distance between any two code words. For example, ASCII code for alphabet *a* is 1100001 and for alphabet *c* is 1100011. These two code words differ in only one bit position (b_1), which shows that the minimum distance of ASCII code is 1.

- (b) ASCII code with even parity for
 a is 11100001 and for
 c is 01100011

These two code words differ in two bit positions (b_1 and b_7). Similarly, it can be verified for any two code words that the distance is 2 or more. Therefore, the minimum distance of ASCII code with even parity is two.

- (c) Similar to part—(b) it can be verified that the minimum distance of 8-bit ASCII code with odd parity is two.

2.10.2 Error-correcting Codes

As discussed above, by adding a single parity bit along with the information, or message being transmitted, an error in single bit position can be detected. The parity check gives only information that the received message is incorrect. It can not locate the bit position in which error has occurred and, therefore, can not correct the error.

Let us consider a 4-bit binary word 0101 is transmitted along with an even parity bit. Due to transmission error in one bit position, the erroneous word received may be 00100, 00111, 00001, or 01101 depending on the error in bit position b_0 , b_1 , b_2 , and b_3 respectively.

Let us examine whether these erroneous words are possible with any other message being transmitted. If the word 01100 is transmitted, it results in 00100 at the receiving end due to an error in bit position b_3 . Similarly, the other erroneous words are received due to the transmission error in some other words being transmitted. This indicates that a minimum distance of two can not locate the bit position in the incorrectly received word. Therefore, for a code to be error correcting, its minimum distance must be more than two.

If the minimum distance is three, every error in single bit results in an invalid code word which is at a distance of one from the original code word and at a distance of two from any other valid code word. Therefore, a single bit error can be detected and located using this code. Once the error bit is located it can be inverted to correct the erroneously received message.

In general, a code is said to be error-correcting code if the correct code word can be deduced from the erroneous word. The capability of a code to be error detecting and/or error-correcting can be determined from its minimum distance. If a code's minimum distance is $2c + d + 1$, it can correct errors in upto c bits and detect errors in upto d additional bits. Table 2.12 gives possible values of c and d for various values of minimum distance of a code.

Table 2.12

Minimum distance of a code	c	d
1	0	0
2	0	1
3	0	2
	1	0
4	0	3
	1	1

From Table 2.12, we observe that if the minimum distance of a code is 4, it can correct upto one bit ($c = 1$) and detect errors in upto two ($c + d = 1 + 1$) bits. The same code can also be used for detecting errors in upto 3 bits but correct no errors ($c = 0$).

Example 2.43

Four messages are encoded in the following code words:

Message	Code
M_1	01101
M_2	10011
M_3	00110
M_4	11000

Determine the minimum distance of this code.

Solution

To determine the minimum distance, we find out the number of locations in which any code word differs from any other code word. These are given below:

Distance between the codes

M_1 and M_2	4
M_1 and M_3	3
M_1 and M_4	3
M_2 and M_3	3
M_2 and M_4	3
M_3 and M_4	4

Therefore, the minimum distance of this code is three.

Example 2.44

Consider the following four codes:

<i>Code A</i>	<i>Code B</i>	<i>Code C</i>	<i>Code D</i>
0001	000	01011	000000
0010	001	01100	001111
0100	011	10010	110011
1000	010	10101	
	110		
	111		
	101		
	100		

Which of the following properties is satisfied by each of the above codes?

- (a) Detects single error
- (b) Detects double errors
- (c) Detects triple errors
- (d) Corrects single error
- (e) Corrects double errors
- (f) Corrects single error and detects double errors

Solution

- (a) Code *A* has a minimum distance of 2
- (b) The minimum distance of code *C* is 3, therefore, it can detect double errors.
- (c) Code *D* has a minimum distance of 4, therefore, it can detect triple errors.
- (d) Code *C* and code *D*
- (e) None
- (f) Code *D*

Hamming Code Hamming code is an error-correcting code. It is constructed by adding a number of parity bits to each group of n -bit information, or message in such a way so as to be able to locate the bit position in which error occurs. Let us assume that k parity bits p_1, p_2, \dots, p_k are added to the n -bit message to form an $(n + k)$ -bit code. The value of k must be chosen in such a way so as to be able to describe the location of any of the $n + k$ possible error bit locations and also 'no error' condition. Consequently, k must satisfy the inequality

$$2^k \geq n + k + 1. \quad (2.2)$$

The location of each of the $n + k$ bits within a code word is assigned a decimal number, starting from 1 to the *MSB* and $n + k$ to the *LSB*. k parity checks are performed on selected bits of each code word. Each parity check includes one of the parity bits. The result of each parity check is recorded as 1 if error has been detected and as 0 if no error has been detected. Let the results of the parity checks involving the parity bits p_k, p_{k-1}, \dots are c_1, c_2, \dots respectively. Bit c_1 is 1 if an error is detected and 0 if there is no error. Similarly c_2, c_3, \dots , etc. The decimal value of the binary word formed c_1, c_2, \dots, c_k gives the decimal value assigned to the location of the erroneous bit. If there is no error, then the decimal value will be 0. This decimal number is the position or location number. The parity bits p_1, p_2, \dots are placed in locations 1, 2, 4, $\dots, 2^{k-1}$.

Example 2.45

Find out the value of k for converting BCD code into Hamming code and the bit positions of the resulting Hamming code.

Solution

The value of k must be chosen to satisfy the eqn. (2.2) since $n = 4$, therefore,

$$2^k \geq k + 5$$

The minimum value of k for which it is satisfied is 3. Therefore, three parity bits are attached to each of the BCD code for constructing the Hamming code. It will be a 7-bit code with bit positions

p_1	p_2	n_1	p_3	n_2	n_3	n_4
1	2	3	4	5	6	7

Values (0 or 1) are assigned to the parity bits so as to make the Hamming code have either even parity or odd parity and when an error occurs, the position number will take on the value assigned to the location of the erroneous bit.

In the case of BCD code with three parity bits there are seven error positions. Table 2.13 gives these error positions and the corresponding values of the position number.

Table 2.13

Error position	Position number		
	c_1	c_2	c_3
0 (no error)	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

From Table 2.13 we observe, that if an error occurs in positions

1, 3, 5, 7	then $c_3 = 1$
2, 3, 6, 7	then $c_2 = 1$
4, 5, 6, 7	then $c_1 = 1$

Therefore, p_1 is selected so as to establish even (or odd) parity in positions 1, 3, 5, 7

p_2 is selected so as to establish even (or odd) parity in positions 2, 3, 6, 7

p_3 is selected so as to establish even (or odd) parity in positions 4, 5, 6, 7

Example 2.46

Construct Hamming code for BCD 0110. Use even parity.

Solution

For 4-bit code three parity bits p_1, p_2 , and p_3 are appended in locations 1, 2, and 4 respectively as shown below:

Position	1	2	3	4	5	6	7
	p_1	p_2	n_1	p_3	n_2	n_3	n_4
Original BCD			0		1	1	0
Even parity in positions 1, 3, 5, 7 requires $p_1 = 1$	1		0		1	1	0
Even parity in positions 2, 3, 6, 7 requires $p_2 = 1$	1	1	0		1	1	0
Even parity in positions 4, 5, 6, 7 requires $p_3 = 0$	1	1	0	0	1	1	0

Therefore, Hamming code for BCD digit 0110 with even parity is 1100110.

Example 2.47

If the Hamming code sequence 1100110 is transmitted and due to error in one position, is received as 1110110, locate the position of the error bit using parity checks and give the method for obtaining the correct sequence.

Solution

Parity check on 4, 5, 6, 7 (0110) positions gives $c_1 = 0$ (even parity)

Parity check on 2, 3, 6, 7 (1110) positions gives $c_2 = 1$ (odd parity)

Parity check on 1, 3, 5, 7 (1110) positions gives $c_3 = 1$ (odd parity)

Therefore, the position number formed is $c_1 c_2 c_3 = 011$, which means that the location of the error is in position 3.

To correct the error the bit received in location 3 is complemented and the correct message 1100110 is received.

Example 2.48

- Find the distance between the BCD digits 0110 and 0111.
- Determine Hamming codes for 0110 and 0111 and find the distance between them. Use even parity.

Solution

- The distance between the BCD numbers 0110 and 0111 is 1 since only the LSB is different.
- The Hamming codes for these are given below

BCD code				Hamming code						
D	C	B	A	p_1	p_2	n_1	p_3	n_2	n_3	n_4
				1	2	3	4	5	6	7
0	1	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	1	1	1	1

These Hamming codes differ in positions 1, 2, 4, and 7, thus the distance between them is four.

Example 2.49

Some 8–4–2–1 code words are transmitted in Hamming code with even parity checking. The following words are received.

- | | |
|-------------|-------------|
| (a) 0101000 | (e) 1110011 |
| (b) 0011101 | (f) 1111001 |
| (c) 1100100 | (g) 1101001 |
| (d) 1100110 | (h) 1000010 |

- Find out the correctly received words, if any.
- Determine the words that have single error and specify the correct decimal digit.
- Find out the words received with double error, if any.

Solution

(a) 0101000

Parity check in positions 4, 5, 6, 7 1000 odd $c_1 = 1$ Parity check in positions 2, 3, 6, 7 1000 odd $c_2 = 1$ Parity check in positions 1, 3, 5, 7 0000 even $c_3 = 0$ The error position is $c_1 c_2 c_3 = 110 = (6)_{10}$.

Therefore, the correct message is 0101010 which is decimal 2.

(b) By performing the parity checks, we obtain $c_1 c_2 c_3 = 101 = 5$

Therefore, the correct message is 0011001 which corresponds to decimal 9.

(c) Here, $c_1 c_2 c_3 = 110 = 6$.

Therefore, the correct message is 1100110 which corresponds to decimal 6.

(d) Here, $c_1 c_2 c_3 = 000$ which means there is no error. This code corresponds to decimal 6.(e) The parity checks give $c_1 c_2 c_3 = 001 = (1)_{10}$, which means the correct message is 0110011. This corresponds to 4-bit message 1011 which is not a valid BCD digit. This shows that there is one more error in this, which can not be corrected i.e. its location can not be determined.(f) Here, $c_1 c_2 c_3 = 011 = (3)_{10}$.

The correct message is 1101001 which corresponds to BCD 1.

(g) Here, $c_1 c_2 c_3 = 000$, which means there is no error. The BCD word is 1.(h) The parity checks give $c_1 c_2 c_3 = 111 = (7)_{10}$. The correct message is 1000011 corresponding to BCD 3.**Example 2.50**

For ASCII code

- determine the number of parity bits which must be appended to the code to make it an error-correcting code i.e. Hamming code.
- determine the locations of the parity bits.

Solution(a) Since $n = 7$, therefore using Eq. (2.2), we obtain

$$2^k \geq 7 + k + 1$$

where, k is the number of parity bits to be attached. Thus gives $k = 4$.

- To determine the locations in which the parity bits are to be attached, we construct Table 2.14 which gives the error positions and the corresponding values of the position numbers.

Table 2.14

Error position	Position number			
	c_1	c_2	c_3	c_4
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

From Table 2.14 we observe, that if an error occurs in positions

1, 3, 5, 7, 9, 11 then $c_4 = 1$

2, 3, 6, 7, 10, 11 then $c_3 = 1$

4, 5, 6, 7 then $c_2 = 1$

8, 9, 10, 11 then $c_1 = 1$

Therefore,

p_1 is selected so as to establish even parity in positions 1, 3, 5, 7, 9, 11
 p_2 is selected so as to establish even parity in positions 2, 3, 6, 7, 10, 11
 p_3 is selected so as to establish even parity in positions 4, 5, 6, 7
 p_4 is selected so as to establish even parity in positions 8, 9, 10, 11

Since one parity bit must be involved in each value of c , therefore, we observe that the parity bits must be located in positions 1, 2, 4 and 8.

Thus the Hamming code will be

p_1	p_2	n_1	p_3	n_2	n_3	n_4	p_4	n_5	n_6	n_7
1	2	3	4	5	6	7	8	9	10	11

SUMMARY

Various number systems that are widely used in digital circuits, microprocessors, computers, etc. have been presented. The rules of arithmetic operations like addition, subtraction, multiplication, division, etc. are given.

Different codes are in use in digital systems for representing numerals, alphabets and special symbols and some of the more commonly used codes have been introduced. The ASCII is the most commonly used code in computers. Any programme and data are entered into the memory of the computer using key-board. When any key is pressed, its ASCII code is generated which gets stored in the memory.

The knowledge of these number systems and codes is very essential for the effective understanding of various digital systems including microprocessors.

When digital data is transmitted from one location to another location error may occur in transmission due to the presence of electrical noise. To detect and correct the error, error-detection and error-correction codes are used. These codes are based on the principle of parity checking. The concepts of error-detection, error-correction together with the codes used have been discussed.

GLOSSARY

Alphanumeric codes Codes that represent numerals, alphabets (letters) and other usual symbols, for example, ASCII code.

ASCII Code (American Standard Code for Information Interchange) A 7-bit code widely used in computers and related areas for representation of alphanumeric characters and special symbols.

Base (or Radix) of a number system The number of distinct symbols (digits) used in a number system.

BCD (Binary-coded decimal) A code for representing decimal numbers in which each decimal digit is represented by its 4-bit binary code. See also Natural BCD.

Byte A group of eight bits.

Code A system of representation of numeric, alphabets or special characters in a binary form for processing and transmission using digital techniques.

Complement Inversion of the value of a binary number, variable, or expression.

Complementation The process of determining complement of a binary number, variable, or expression.

Decimal number system A number system with base (or radix) 10. The ten digits used to represent any number are: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Encoding The process of coding alphabets, numerals and symbols in binary format.

Even parity A digital word (string of 0s and 1s) having an even number of ones.

Error correcting code A code used for correction of error in the transmission of digital signals.

Error detecting code A code used for detection of error in the transmission of digital signals.

Excess-3 code A BCD code formed by adding 3(0011) to the binary equivalent of the decimal number.

Gray code A code in which only one bit changes between successive numbers.

Hamming code An error correcting code.

HEX Abbreviation for hexadecimal.

Hexadecimal code A method of representing binary numbers in which each group of 4 bits (starting from the right most bit) is represented by its hex digit.

Hexadecimal number system A number system that uses digits 0 through 9 and the alphabets A through F. Its base (or radix) is 16.

Least significant bit (LSB) The right-most bit of a binary number. It has the least weight.

Least significant digit (LSD) The right most digit of a number.

Most significant bit (MSB) The left-most bit of a binary number.

Most significant digit (MSD) The left-most digit of a number.

Natural BCD BCD representation that uses natural binary numbers.

Nibble A group of four bits.

Nine's complement Nine's complement of an N -digit-decimal number is the number obtained by subtracting it from an N -digit number consisting of all 9's.

Octal code A code in which each group of three bits starting from LSB is represented by its equivalent octal digit.

Octal number system A number system with base (or radix) 8 that uses digits 0, 1, 2, 3, 4, 5, 6, and 7.

Odd parity A digital word having an odd number of ones.

One's complement The number obtained by complementing each bit of a binary number.

One's complement representation A binary representation used for representing positive as well as negative numbers (signed numbers).

Parity A term used to specify the number of ones in a digital word as odd or even.

Parity bit An extra bit attached to a binary word to make the parity of the resultant word even or odd.

Parity checker A logic circuit that checks the parity of a binary word.

Parity generator A logic circuit that generates an additional bit which when appended to a digital word makes its parity as desired (odd or even).

Positional number system A number system in which value of a digit depends upon its position in the number.

Radix Same as the base.

Sign bit The MSB of a signed binary number. If 0, the number is positive, when it is 1, the number is negative.

Signed binary number A binary number that is either positive or negative.

Sign-magnitude representation A representation system for signed binary numbers in which the MSB represents the sign and the remaining bits represent the magnitude of the number.

Two's complement Binary number obtained by adding one to the one's complement of a binary number.

Two's complement representation A method of representation of binary number in which negative numbers are represented by two's complement of their positive equivalents.

Weighted code A binary code in which weight is assigned to each position in the number.

Word A group of bits.

REVIEW QUESTIONS

- 2.1 The radix of a binary number system is _____ and the digits used are _____.
- 2.2 In _____ number system, 16 distinct symbols are used to specify any number.
- 2.3 A byte contains _____ bits.

- 2.4 The weights assigned in an 8-bit binary number to LSB and MSB are _____ and _____ respectively.
- 2.5 The MSB of a signed-binary number indicates its _____.
- 2.6 2's complement of a 2's complement is _____.
- 2.7 The number of bits required to represent 25 in BCD is _____.
- 2.8 The Excess-3 code for decimal number 8 is _____.
- 2.9 The number of bits in ASCII code is _____.
- 2.10 The number of characters represented by ASCII code is _____.
- 2.11 The parity of 01110010 is _____.
- 2.12 The minimum distance required for a code to be error detecting code is _____.
- 2.13 A minimum distance of _____ is required for a code to be error correcting code.
- 2.14 The process of subtraction gets converted into that of addition by using _____.
- 2.15 Gray code is a _____. (weighted/non-weighted)
- 2.16 The distance between the code words 10010 and 10101 is _____.
- 2.17 A single parity bit attached to 8421 code makes its minimum distance _____.
- 2.18 A minimum of _____ parity bits are required for generating Hamming code for 8421 code.
- 2.19 The number of parity bits required for generating Hamming code for ASCII code is _____.
- 2.20 In 7-bit Hamming code for BCD, the parity bits are at _____ locations.
- 2.21 The minimum distance of ASCII code changes from _____ to _____ in its Hamming code.

PROBLEMS

2.1 Determine the decimal numbers represented by the following binary numbers:

- | | | | |
|------------|--------------|---------------|-------------|
| (a) 111001 | (c) 11111110 | (e) 1101.0011 | (g) 0.11100 |
| (b) 101001 | (d) 1100100 | (f) 1010.1010 | |

2.2 Determine the binary numbers represented by the following decimal numbers:

- | | | |
|---------|-----------|-----------|
| (a) 37 | (c) 15 | (e) 11.75 |
| (b) 255 | (d) 26.25 | (f) 0.1 |

2.3 Add the following groups of binary numbers:

- | | |
|---|--|
| (a) $\begin{array}{r} 1011 \\ + 1101 \\ \hline \end{array}$ | (b) $\begin{array}{r} 1010.1101 \\ + 101.01 \\ \hline \end{array}$ |
|---|--|

2.4 Perform the following subtractions using 2's complement method:

- | | | |
|---------------------|---------------------|-----------------------------|
| (a) $01000 - 01001$ | (b) $01100 - 00011$ | (c) $0011.1001 - 0001.1110$ |
|---------------------|---------------------|-----------------------------|

2.5 Convert the following numbers from decimal to octal and then to binary. Compare the binary numbers obtained with the binary numbers obtained directly from the decimal numbers.

- | | | |
|---------|---------|------------|
| (a) 375 | (b) 249 | (c) 27.125 |
|---------|---------|------------|

2.6 Convert the following binary numbers to octal and then to decimal. Compare the decimal numbers obtained with the decimal numbers obtained directly from the binary numbers.

- | | | |
|---------------------|---------------------|--------------|
| (a) 11011100.101010 | (b) 01010011.010101 | (c) 10110011 |
|---------------------|---------------------|--------------|

- 2.7** Convert the decimal numbers in Problem 2.5 to hexadecimal and then to binary. Compare the binary numbers obtained with the binary numbers obtained directly from the decimal numbers.
- 2.8** Convert the binary numbers in Problem 2.6 to hexadecimal and then to decimal. Compare the decimal numbers obtained with the decimal numbers obtained directly from the binary numbers.
- 2.9** Encode the following decimal numbers in BCD code:
- (a) 46 (b) 327.89 (c) 20.305
- 2.10** Encode the decimal numbers in Problem 2.9 to Excess-3 code.
- 2.11** Encode the decimal number 46 to Gray code.
- 2.12** Use the 6-bit internal code to represent the statement.

$$P = 3 * Q$$

- 2.13** Write your full name in
- (a) ASCII code (b) EBCDIC code (c) 6-bit internal code
- Include blanks wherever necessary.
- 2.14** Attach an even parity bit as MSB for
- (a) ASCII code (b) EBCDIC code
- 2.15** Repeat Problem 2.14 for odd parity.
- 2.16** Find the number of bits required to encode:
- (a) 56 elements of information (b) 130 elements of information
- 2.17** Write 8-bit ASCII code (parity and 7-bit code) obtained in Problems 2.14 and 2.15 in hexadecimal format.
- 2.18** Develop the binary subtraction rules using one's complement representation for negative numbers.
- 2.19** How many bits of memory are required for storing 100 names of a group of people, assuming that no name occupies more than 20 characters (including spaces)? Assume 7-bit ASCII code with parity bit.
- 2.20** A line printer is capable of printing 132 characters in a single line and each character is represented by ASCII code. How many bits are required to print each line?
- 2.21** How many words can be added to code *A*, in Example 2.44, without changing its error-detection and correction capabilities? Give a possible set of such words. Is this set unique?
- 2.22** Find the number and positions of parity bits to be added to construct Hamming code for an 8-bit data word.
- 2.23** Determine Hamming code sequence with odd parity for natural BCD for making it an error correcting code.
- 2.24** For ASCII code words 1010010 and 1010000
- (a) determine the distance between them.
- (b) Determine Hamming code words and distance between them.
- 2.25** Construct Hamming codes for the following 8-bits words
- (a) 10101010 (b) 00000000 (c) 11111111
- 2.26** For some 8-bit data words, the following Hamming code words are received. Determine the correct data words. Assume even parity check.
- (a) 000011101010 (b) 101110000110 (c) 101111110100