1 Number Systems and Binary Codes

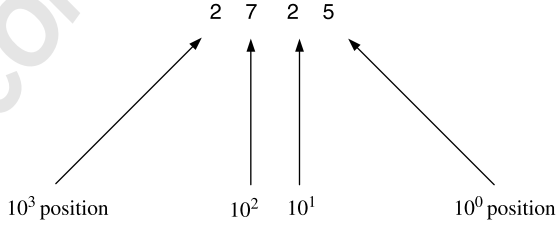
1.1 INTRODUCTION

In conventional arithmetic, a number system based on ten units (0 to 9) is used. However, arithmetic and logic circuits used in computers and other digital systems operate with only 0's and 1's because it is very difficult to design circuits that require ten distinct states. The number system with the basic symbols 0 and 1 is called *binary*. Although digital systems use binary numbers for their internal operations, communication with the external world has to be done in decimal systems. In order to simplify the communication, every decimal number may be represented by a unique sequence of binary digits; this is known as *binary encoding*. In this chapter we discuss number systems in general and the binary system in particular. In addition, we consider the octal and hexadecimal number systems and fixed- and floating-point representation of numbers. The chapter ends with a discussion on weighted and nonweighted binary encoding of decimal digits.

1.2 DECIMAL NUMBERS

The invention of decimal number systems has been the most important factor in the development of science and technology. The term decimal comes from the Latin word for "ten." The decimal number system uses positional number representation, which means that the value of each digit is determined by its position in a number.

The base (also called radix) of a number system is the number of symbols that the system contains. The decimal system has ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; in other words it has a base of 10. Each position in the decimal system is 10 times more significant than the previous position. For example, consider the four-digit number 2725:



Notice that the 2 in the 10^3 position has a different value than the 2 in the 10^1 position. The value of a decimal number is determined by multiplying each digit of the number by the

Principles of Modern Digital Design, by Parag K. Lala

value of the position in which the digit appears and then adding the products. Thus the number 2725 is interpreted as

$$2 \times 1000 + 7 \times 100 + 2 \times 10 + 5 \times 1 = 2000 + 700 + 20 + 5$$

that is, two thousand seven hundred twenty-five. In this case, 5 is the least significant digit (LSD) and the leftmost 2 is the most significant digit (MSD).

In general, in a number system with a base or radix r, the digits used are from 0 to $r - 1$. The number can be represented as

$$N = a_n r^n + a_{n-1} r^{n-1} + \cdots + a_1 r^1 + a_0 r^0 \tag{1.1}$$

where, for $n = 0, 1, 2, 3, \ldots,$

$r =$ base or radix of the number system
$a =$ number of digits having values between 0 and $r - 1$

Thus, for the number 2725, $a_3 = 2$, $a_2 = 7$, $a_1 = 2$, and $a_0 = 5$. Equation (1.1) is valid for all integers. For numbers between 0 and 1 (i.e., fractions), the following equation holds:

$$N = a_{-1} r^{-1} + a_{-2} r^{-2} + \cdots + a_{-n+1} r^{-n+1} + a_{-n} r^{-n} \tag{1.2}$$

Thus for the decimal fraction 0.8125,

$$\begin{aligned} N &= 0.8000 + 0.0100 + 0.0020 + 0.0005 \\ &= 8 \times 10^{-1} + 2 \times 10^{-2} + 1 \times 10^{-3} + 8 \times 10^{-4} \\ &= a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} + a_{-3} \times 10^{-3} + a_{-4} \times 10^{-4} \end{aligned}$$

where

$$\begin{aligned} a_{-1} &= 8 \\ a_{-2} &= 1 \\ a_{-3} &= 2 \\ a_{-1} &= 5 \end{aligned}$$

1.3 BINARY NUMBERS

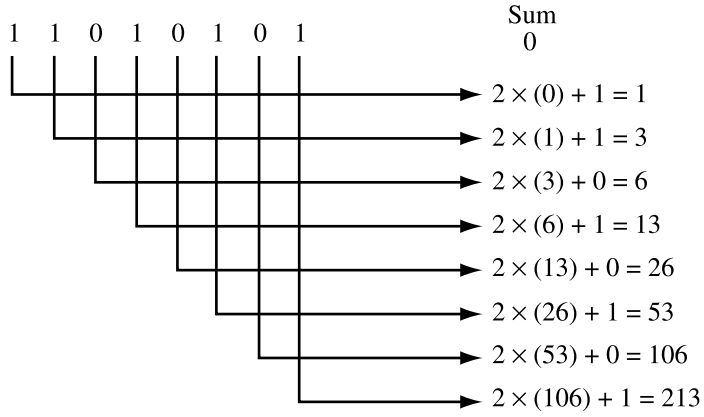
The binary numbers has a radix of 2. As $r = 2$, only two digits are needed, and these are 0 and 1. A binary digit, 0 or 1, is called a bit. Like the decimal system, binary is a positional system, except that each bit position corresponds to a power of 2 instead of a power of 10. In digital systems, the binary number system and other number systems closely related to it are used almost exclusively. However, people are accustomed to using the decimal number system; hence digital systems must often provide conversion between decimal and binary numbers. The decimal value of a binary number can be formed by multiplying each power of 2 by either 1 or 0, and adding the values together.

Example 1.1 Let us find the decimal equivalent of the binary number 101010.

$$
\begin{aligned}
N &= 101010 \\
&= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \quad \text{(using Eq. (1.1))} \\
&= 32 + 0 + 8 + 0 + 2 + 0 \\
&= 42
\end{aligned}
$$

An alternative method of converting from binary to decimal begins with the leftmost bit and works down to the rightmost bit. It starts with a sum of 0. At each step the current sum is multiplied by 2, and the next digit to the right is added to it.

Example 1.2 The conversion of 11010101 to decimal would use the following steps:



The reverse process, the conversion of decimal to binary, may be made by first decomposing the given decimal number into two numbers—one corresponding to the positional value just lower than the original decimal number and a remainder. Then the remainder is decomposed into two numbers: a positional value just equal to or lower than itself and a new remainder. The process is repeated until the remainder is 0. The binary number is derived by recording 1 in the positions corresponding to the numbers whose summation equals the decimal value.

Example 1.3 Let us consider the conversion of decimal number 426 to binary:

$$
\begin{aligned}
426 &= 256 + 170 \\
&= 256 + 128 + 42 \\
&= 256 + 128 + 32 + 10 \\
&= 256 + 128 + 32 + \ 8 + 2 \\
&\qquad\ \uparrow \quad\ \uparrow \quad\ \uparrow \quad\ \uparrow \quad\ \uparrow \\
&\qquad 2^8 \quad 2^7 \quad 2^5 \quad 2^3 \quad 2^1
\end{aligned}
$$

Thus $426_{10} = 110101010_2$ (the subscript indicates the value of the radix).

An alternative method for converting a decimal number to binary is based on successive division of the decimal number by the radix number 2. The remainders of the divisions, when written in reverse order (with the first remainder on the right), yield the binary equivalent to the decimal number. The process is illustrated below by converting 353_{10} to binary,

$$\frac{353}{2} = 176, \text{ remainder } 1$$
$$\frac{176}{2} = 88, \text{ remainder } 0$$
$$\frac{88}{2} = 44, \text{ remainder } 0$$
$$\frac{44}{2} = 22, \text{ remainder } 0$$
$$\frac{22}{2} = 11, \text{ remainder } 0$$
$$\frac{11}{2} = 5, \text{ remainder } 1$$
$$\frac{5}{2} = 2, \text{ remainder } 1$$
$$\frac{2}{2} = 1, \text{ remainder } 0$$
$$\frac{1}{2} = 0, \text{ remainder } 1$$

Thus $353_{10} = 101100001_2$.

So far we have only considered whole numbers. Fractional numbers may be converted in a similar manner.

Example 1.4 Let us convert the fractional binary number 0.101011 to decimal. Using Eq. (1.2), we find

$$\begin{aligned} N &= 0.101011 \\ &= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6} \end{aligned}$$

where $a_{-1} = 1$, $a_{-2} = 0$, $a_{-3} = 1$, $a_{-4} = 0$, $a_{-5} = 1$, $a_{-6} = 1$.

Thus

$$\begin{aligned} N &= 0.101011 \\ &= \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{64} = 0.671875 \end{aligned}$$

A decimal fraction can be converted to binary by successively multiplying it by 2; the integral (whole number) part of each product, 0 or 1, is retained as the binary fraction.

Example 1.5 Derive the binary equivalent of the decimal fraction 0.203125. Successive multiplication of the fraction by 2 results in

$$\begin{array}{rr} & 0.203125 \\ & \underline{\qquad 2} \\ a_{-1} = 0 & 0.406250 \\ & \underline{\qquad 2} \\ a_{-2} = 0 & 0.812500 \\ & \underline{\qquad 2} \\ a_{-3} = 1 & 0.625000 \\ & \underline{\qquad 2} \\ a_{-4} = 1 & 0.250000 \\ & \underline{\qquad 2} \\ a_{-5} = 0 & 0.500000 \\ & \underline{\qquad 2} \\ a_{-6} = 1 & 0.000000 \end{array}$$

Thus the binary equivalent of 0.203125_{10} is 0.001101_2. The multiplication by 2 is continued until the decimal number is exhausted (as in the example) or the desired accuracy is achieved. Accuracy suffers considerably if the conversion process is stopped too soon. For example, if we stop after the fourth step, then we are assuming 0.0011 is approximately equal to 0.20315, whereas it is actually equal to 0.1875, an error of about 7.7%.

1.3.1 Basic Binary Arithmetic

Arithmetic operations using binary numbers are far simpler than the corresponding operations using decimal numbers due to the very elementary rules of addition and multiplication. The rules of binary addition are

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0 \ \text{(carry 1)} \end{aligned}$$

As in decimal addition, the least significant bits of the addend and the augend are added first. The result is the sum, possibly including a carry. The carry bit is added to the sum of the digits of the next column. The process continues until the bits of the most significant column are summed.

Example 1.6 Let us consider the addition of the decimal numbers 27 and 28 in binary.

Decimal	Binary	
27	11011	Addend
+28	+11100	Augend
55	110111	Sum
	11000	Carry

To verify that the sum is correct, we convert 110111 to decimal:

$$\begin{aligned} &1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 32 + 16 + 0 + 4 + 2 + 1 \\ &= 55 \end{aligned}$$

Example 1.7 Let us add -11 to -19 in binary. Since the addend and the augend are negative, the sum will be negative.

Decimal	Binary	
19	10011	
11	01011	
30	11110	Sum
	00011	Carry

In all digital systems, the circuitry used for performing binary addition handles two numbers at a time. When more than two numbers have to be added, the first two are added, then the resulting sum is added to the third number, and so on.

Binary subtraction is carried out by following the same method as in the decimal system. Each digit in the *subtrahend* is deducted from the corresponding digit in the *minuend* to obtain the difference. When the minuend digit is less than the subtrahend digit, then the radix number (i.e., 2) is added to the minuend, and a *borrow* 1 is added to the next subtrahend digit. The rules applied to the binary subtraction are

$$\begin{aligned} &0 - 0 = 0 \\ &0 - 1 = 1 \quad \text{(borrow 1)} \\ &1 - 0 = 1 \\ &1 - 1 = 0 \end{aligned}$$

Example 1.8 Let us consider the subtraction of 21_{10} from 27_{10} in binary:

Decimal	Binary	
27	11011	Minuend
-21	-10101	Subtrahend
6	00110	Difference
	00100	Borrow

It can easily be verified that the difference 00110_2 corresponds to decimal 6.

Example 1.9 Let us subtract 22_{10} from 17_{10}. In this case, the subtrahend is greater than the minuend. Therefore the result will be negative.

Decimal	Binary	
17	10001	
-22	-10110	
-5	-00101	Difference
	00001	Borrow

Binary multiplication is performed in the same way as decimal multiplication, by multiplying, then shifting one place to the left, and finally adding the partial products. Since the multiplier can only be 0 or 1, the partial product is either zero or equal to the multiplicand. The rules of multiplication are

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$1 \cdot 0 = 0$$
$$1 \cdot 1 = 1$$

Example 1.10 Let us consider the multiplication of the decimal numbers 67 by 13 in binary:

Decimal	Binary	
67	1000011	Multiplicand
$\times$ 13	1101	Multiplier
871	1000011	First partial product
	0000000	Second partial product
	1000011	Third partial product
	1000011	Fourth partial product
	1101100111	Final product

Example 1.11 Let us multiply 13.5 by 3.25.

Decimal	Binary	
13.5	1101.10	Multiplicand
$\times$ 3.25	11.01	Multiplier
43.875	110110	First partial product
	000000	Second partial product
	110110	Third partial product
	110110	Fourth partial product
	101011.1110	Final product

The decimal equivalent of the final product is $43 + 0.50 + 0.25 + 0.125 = 43.875$.

The process of binary division is very similar to standard decimal division. However, division is simpler in binary because when one checks to see how many times the divisor fits into the dividend, there are only two possibilities, 0 or 1.

Example 1.12 Let us consider the division of 101110 (46_{10}) by 111 (7_{10})

```
                0001      Quotient
Divisor 111 ) 101110      Dividend
              0111
              ----
               100
```

Since the divisor, 111, is greater than the first three bits of the dividend, the first three quotient bits are 0. The divisor is less than the first four bits of the dividend; therefore

the division is possible, and the fourth quotient bit is 1. The difference is less than the divisor, so we bring down the net bit of the dividend:

$$\begin{array}{rr} & 00011 \\ 111 & \overline{)\,101110} \\ & \underline{0111} \\ & 1001 \\ & \underline{111} \\ & 10 \end{array}$$

The difference is less than the divisor, so the next bit of the dividend is brought down:

$$\begin{array}{rr} & 000110 \\ 111 & \overline{)\,101110} \\ & \underline{0111} \\ & 1001 \\ & \underline{111} \\ & 100 \quad \text{Remainder} \end{array}$$

In this case the dividend is less than the divisor; hence the next quotient bit is 0 and the division is complete. The decimal conversion yields $46/7 = 6$ with remainder 4, which is correct.

The methods we discussed to perform addition, subtraction, multiplication, and division are equivalents of the same operations in decimal. In digital systems, all arithmetic operations are carried out in modified forms; in fact, they use only addition as their basic operation.

1.4 OCTAL NUMBERS

Digital systems operate only on binary numbers. Since binary numbers are often very long, two shorthand notations, *octal* and *hexadecimal*, are used for representing large binary numbers. The octal number system uses a base or radix of 8; thus it has digits from 0 to $r - 1$, or $8 - 1$, or 7. As in the decimal and binary systems, the positional value of each digit in a sequence of numbers is definitely fixed. Each position in an octal number is a power of 8, and each position is 8 times more significant than the previous position. The number 375_8 in the octal system therefore means

$$\begin{aligned} 3 \times 8^2 + 7 \times 8^1 + 5 \times 8^0 &= 192 + 56 + 5 \\ &= 253_{10} \end{aligned}$$

Example 1.13 Let us determine the decimal equivalent of the octal number 14.3.

$$\begin{aligned} 14.3_{\text{g}} &= 1 \times 8^1 + 4 \times 8^0 + 3 \times 8^{-1} \\ &= 8 + 4 + 0.375 \\ &= 12.375 \end{aligned}$$

The method for converting a decimal number to an octal number is similar to that used for converting a decimal number to binary (Section 1.2), except that the decimal number is successively divided by 8 rather than 2.

Example 1.14 Let us determine the octal equivalent of the decimal number 278.

$$\frac{278}{8} = 34, \text{ remainder } 6$$
$$\frac{34}{8} = 4, \text{ remainder } 2$$
$$\frac{4}{8} = 0, \text{ remainder } 4$$

Thus $278_{10} = 426_8$.

Decimal fractions can be converted to octal by progressively multiplying by 8; the integral part of each product is retained as the octal fraction. For example, 0.651_{10} is converted to octal as follows:

$$\begin{array}{rl} & 0.651 \\ & \underline{\quad\quad 8} \\ 5 & 0.208 \\ & \underline{\quad\quad 8} \\ 1 & 0.664 \\ & \underline{\quad\quad 8} \\ 5 & 0.312 \\ & \underline{\quad\quad 8} \\ 2 & 0.496 \\ & \underline{\quad\quad 8} \\ 3 & 0.968 \\ & \text{etc.} \end{array}$$

According to Eq. (1.2), $a_{-1} = 5$, $a_{-2} = 1$, $a_{-3} = 5$, $a_{-4} = 2$, and $a_{-5} = 3$; hence $0.651_{10} = 0.51523_8$. More octal digits will result in more accuracy.

A useful relationship exists between binary and octal numbers. The number of bits required to represent an octal digit is three. For example, octal 7 can be represented by binary 111. Thus, if each octal digit is written as a group of three bits, the octal number is converted into a binary number.

Example 1.15 The octal number 324_8 can be converted to a binary number as follows:

$$\begin{array}{ccc} 3 & 2 & 4 \\ \downarrow & \downarrow & \downarrow \\ 011 & 010 & 100 \end{array}$$

Hence $324_8 = 11010100_2$; the most significant 0 is dropped because it is meaningless, just as 0123_{10} is the same as 123_{10}.

The conversion from binary to octal is also straightforward. The binary number is partitioned into groups of three starting with the least significant digit. Each group of three binary digits is then replaced by an appropriate decimal digit between 0 and 7 (Table 1.1).

TABLE 1.1 Binary to Octal Conversion

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Example 1.16 Let us convert 110011101001_2 to octal:

$$\underbrace{110}_{6}\ \underbrace{011}_{3}\ \underbrace{101}_{5}\ \underbrace{001}_{1}$$

The octal representation of the binary number is 6351_8. If the leftmost group of a partitioned binary number does not have three digits, it is padded on the left with 0's. For example, 1101010 would be divided as

$$\underbrace{001}_{1}\ \underbrace{101}_{5}\ \underbrace{010}_{2}$$

The octal equivalent of the binary number is 152_8. In case of a binary fraction, if the bits cannot be grouped into 3-bit segments, the 0's are added on the right to complete groups of three. Thus 110111.1011 can be written

$$\underbrace{110}_{6}\ \underbrace{111}_{7}\ .\ \underbrace{101}_{5}\ \underbrace{100}_{4}$$

As shown in the previous section, the binary equivalent of a decimal number can be obtained by successively dividing the number by 2 and using the remainders as the answer, the first remainder being the lowest significant bit, and so on. A large number of divisions by 2 are required to convert from decimal to binary if the decimal number is large. It is often more convenient to convert from decimal to octal and then replace each digit in octal in terms of three digits in binary. For example, let us convert 523_{10} to binary by going through octal.

$$\frac{523}{8} = 65, \text{ remainder } 3$$

$$\frac{65}{8} = 8, \text{ remainder } 1$$

$$\frac{8}{8} = 1, \text{ remainder } 0$$

$$\frac{1}{8} = 0, \text{ remainder } 1$$

Thus

$$\begin{aligned}(523)_{10} &= (1 \quad 0 \quad 1 \quad 3)_8 \\ &= (001 \quad 000 \quad 001 \quad 011)_2\end{aligned}$$

It can be verified that the decimal equivalent of 001000001011_2 is 523_{10}:

$$\begin{aligned}1 \times 2^9 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 &= 512 + 8 + 2 + 1 \\ &= 523_{10}\end{aligned}$$

Addition and subtraction operations using octal numbers are very much similar to that use in decimal systems. In octal addition, a carry is generated when the sum exceeds 7_{10}. For example,

$$\begin{array}{r} 153_8 \\ +\,327_8 \\ \hline 502_8 \end{array}$$

$3 + 7 = 10_{10} = 2 + 1$ carry ⟵ first column
$5 + 2 + 1$ carry $= 0 + 1$ carry ⟵ second column
$1 + 3 + 1$ carry $= 5$ ⟵ third column

In octal subtraction, a borrow requires that 8_{10} be added to the minuend digit and a 1_{10} be added to the left adjacent subtrahend digit.

$$\begin{array}{r} 670_8 \\ -\,125_8 \\ \hline 543_8 \end{array}$$

$0 - 5 = (8 - 5 + 1\text{ borrow})_{10} = 3 + 1$ borrow ⟵ first column
$7 - (2 + 1\text{ borrow}) = 7 - 3 = 4$ ⟵ second column
$6 - 1 = 5$ ⟵ third column

1.5 HEXADECIMAL NUMBERS

The hexadecimal numbering system has a base 16; that is, there are 16 symbols. The decimal digits 0 to 9 are used as the first ten digits as in the decimal system, followed by the letters A, B, C, D, E, and F, which represent the values 10, 11, 12, 13, 14, and 15, respectively. Table 1.2 shows the relationship between decimal, binary, octal, and hexadecimal number systems. The conversion of a binary number to a hexadecimal number consists of partitioning the binary numbers into groups of 4 bits, and representing each group with its hexadecimal equivalent.

TABLE 1.2 Number Equivalents

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Example 1.17 The binary number 1010011011110001 is grouped as

$$1010 \quad 0110 \quad 1111 \quad 0001$$

which is shown here in hexadecimal:

$$A6F1_H$$

The conversion from hexadecimal to binary is straightforward. Each hexadecimal digit is replaced by the corresponding 4-bit equivalent from Table 1.2. For example, the binary equivalent of $4AC2_H$ is

$$\begin{array}{cccc} 4 & A & C & 2 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 0100 & 1010 & 1110 & 0010 \end{array}$$

Thus $4AC2_H = 0100101011100010_2$.

Sometimes it is necessary to convert a hexadecimal number to decimal. Each position in a hexadecimal number is 16 times more significant than the previous position. Thus the decimal equivalent for $1A2D_H$ is

$$\begin{aligned} &1 \times 16^3 + A \times 16^2 + 2 \times 16^1 + D \times 16^0 \\ &= 1 \times 16^3 + 10 \times 16^2 + 2 \times 16^1 + 13 \times 16^0 \\ &= 6701 \end{aligned}$$

Hexadecimal numbers are often used in describing the data in a computer memory. A computer memory stores a large number of words, each of which is a standard size collection

of bits. An 8-bit word is known as a *byte*. A hexadecimal digit may be considered as half of a byte. Two hexadecimal digits constitute one byte, the rightmost 4 bits corresponding to half a byte, and the leftmost 4 bits corresponding to the other half of the byte. Often a half-byte is called a *nibble*.

Hexadecimal addition and subtraction are performed as for any other positional number system.

Example 1.18 Let us find the sum of 688_H and 679_H.

$$\begin{array}{r} 688_H \\ 679_H \\ \hline D01_H \end{array}$$

$$\begin{aligned} 8 + 9 &= 17_{10} = 1 + 1 \text{ carry} \longleftarrow \text{first column} \\ 8 + 7 + 1 \text{ carry} &= 16_{10} = 0 + 1 \text{ carry} \longleftarrow \text{second column} \\ 6 + 6 + 1 \text{ carry} &= 13_{10} = D \longleftarrow \text{third column} \end{aligned}$$

Hexadecimal subtraction requires the same need to carry digits from left to right as in octal and decimal.

Example 1.19 Let us compute $2A5_H - 11B_H$ as shown:

$$\begin{array}{r} 2A5_H \\ 11B_H \\ \hline 18A_H \end{array}$$

$$\begin{aligned} 5 - B = (21 - 11 + 1 \text{ borrow})_{10} &= 10 + 1 \text{ borrow} \\ &= A + 1 \text{ borrow} \longleftarrow \text{first column} \\ A - (1 + 1 \text{ borrow}) = (10 - 2)_{10} &= 8 \longleftarrow \text{second column} \\ 2 - 1 &= 1 \longleftarrow \text{third column} \end{aligned}$$

1.6 SIGNED NUMBERS

So far, the number representations we considered have not carried sign information. Such unsigned numbers have a magnitude significance only. Normally a prefix + or − may be placed to the left of the magnitude to indicate whether a number is positive or negative. This type of representation, known as *sign-magnitude representation*, is used in the decimal system. In binary systems, an additional bit known as sign bit, is added to the left of the most significant bit to define the sign of a number. A 1 is used to represent − and a 0 to represent +. Table 1.3 shown 3-bit numbers in terms of signed and unsigned equivalents. Notice that there are two representations of the number 0, namely, +0 and −0. The range of integers that can be expressed in a group of three bits is from $-(2^2 - 1) = -3$ to $+(2^2 - 1) = +3$, with one bit being reserved to denote the sign.

TABLE 1.3 Signed and Unsigned Binary Numbers

	Decimal Equivalent	
Binary	Signed	Unsigned
000	+0	0
001	+1	1
010	+2	2
011	+3	3
100	−0	4
101	−1	5
110	−2	6
111	−3	7

Although the sign-magnitude representation is convenient for computing the negative of a number, a problem occurs when two numbers of opposite signs have to be added. To illustrate, let us find the sum of $+2_{10}$ and -6_{10}.

$$+2_{10} = 0010$$
$$-6_{10} = \underline{1110}$$
$$10000$$

The addition produced a sum that has 5 bits, exceeding the capability of the number system (sign +3 bits); this results in an *overflow*. Also, the sum is wrong; it should be −4 (i.e., 1100) instead of 0.

An alternative representation of negative numbers, known as the *complement* form, simplifies the computations involving signed numbers. The complement representation enjoys the advantage that no sign computation is necessary. There are two types of complement representations: diminished radix complement and radix complement.

1.6.1 Diminished Radix Complement

In the decimal system ($r = 10$) the complement of a number is determined by subtracting the number from $(r - 1)$, that is, 9. Hence the process is called finding the 9's complement. For example,

9's complement of 5 $(9 - 5) = 4$
9's complement of 63 $(99 - 63) = 36$
9's complement of 110 $(999 - 110) = 889$

In binary notation ($r = 2$), the diminished radix complement is known as the *1's complement*. A positive number in 1's complement is represented in the same way as in sign-magnitude representation. The 1's complement representation of a negative number x is derived by subtracting the binary value of x from the binary representation of $(2^n - 1)$, where n is the number of bits in the binary value of x.

Example 1.20 Let us compute the 1's complement form of -43. The binary value of $43 = 00101011$. Since $n = 8$, the binary representation of $2^8 - 1$ is

$$\begin{aligned} 2^8 &= 100000000 \\ &\quad -1 \\ \hline 2^8 - 1 &= \ \ 11111111 \end{aligned}$$

Hence the 1's complement form of -43 is

$$\begin{aligned} 2^8 - 1 &= \ \ 11111111 \\ -43 &= -00101011 \\ \hline &\quad\ \ 11010100 \end{aligned}$$

Notice that the 1's complement form of -43 is a number that has a 0 in every position that $+43$ has a 1, and vice versa. Thus the 1's complement of any binary number can be obtained by complementing each bit in the number.

A 0 in the most significant bit indicates a positive number. The sign bit is not complemented when negative numbers are represented in 1's complement form. For example, the 1's complement form of -25 will be represented as follows:

$$\begin{aligned} -25 &= 111001 \quad \text{(sign-magnitude form)} \\ &= 100110 \quad \text{(1's complement form)} \end{aligned}$$

Table 1.4 shows the comparison of 3-bit unsigned, signed, and 1's complement values. The advantage of using 1's complement numbers is that they permit us to perform subtraction by actually using the addition operation. It means that, in digital systems, addition and subtraction can be carried out by using the same circuitry.

The addition operation for two 1's complement numbers consists of the following steps:

(i) Add the two numbers including the sign bits.

(ii) If a carry bit is produced by the leftmost bits (i.e., the sign bits), add it to the result. This is called *end-around carry*.

TABLE 1.4 Comparison of 3-Bit Signed, Unsigned, and 1's Complement Values

	Decimal Equivalent		
Binary	Unsigned	Sign-Magnitude	1's Complement
000	0	+0	+0
001	1	+1	+1
010	2	+2	+2
011	3	+3	+3
100	4	−0	−3
101	5	−1	−2
110	6	−2	−1
111	7	−3	−0

Example 1.21 Let us add -7 to -5:

$$\begin{array}{rr} \text{1's complement form of } -7 = & 11000 \\ \text{1's complement form of } -5 = & 11010 \\ \text{Sum} & 110010 \\ \text{Carry} & \rightarrow 1 \\ \text{1's complement sum} & 10011 \end{array}$$

The result is a negative number. By complementing the magnitude bits we get

$$11100, \text{ that is, } -12$$

Thus the sum of -7 and -5 is -12, which is correct.

The subtraction operation for two 1's complement numbers can be carried out as follows:

(i) Derive the 1's complement of the subtrahend and add it to the minuend.
(ii) If a carry bit is produced by the leftmost bits (i.e., the sign bits), add it to the result.

Example 1.22 Let us subtract $+21$ from $+35$:

$$\begin{array}{rrl} \text{Minuend} = +35 = & 0100011 & \\ \text{Subtrahend} = +21 = & 1101010 & \text{(in 1's complement form)} \\ \text{Sum} = & 10001101 & \\ \text{Carry} & \rightarrow 1 & \\ \text{1's complement sum} = & 0001110 & \end{array}$$

The sign is correct, and the decimal equivalent $+14$ is also correct.

The 1's complement number system has the advantage that addition and subtraction are actually one operation. Unfortunately, there is still a dual representation for 0. With 3-bit numbers, for example, 000 is *positive zero* and 111 is *negative zero*.

1.6.2 Radix Complement

In the decimal system, the radix complement is the 10's complement. The 10's complement of a number is the difference between 10 and the number. For example,

$$\begin{aligned} &\text{10's complement of 5,} && 10 - 5 = 5 \\ &\text{10's complement of 27,} && 100 - 27 = 73 \\ &\text{10's complement of 48,} && 100 - 48 = 52 \end{aligned}$$

In binary number system, the radix complement is called the 2's complement. The *2's*

complement representation of a positive number is the same as in sign-magnitude form. The 2's complement representation of a negative number is obtained by complementing the sign-magnitude representation of the corresponding positive number and adding a 1 to the least significant position. In other words, the 2's complement form of a negative number can be obtained by adding 1 to the 1's complement representation of the number.

Example 1.23 Let us compute the 2's complement representation of −43:

$$\begin{aligned} +43 &= 0101011 && \text{(sign-magnitude form)} \\ &= 1010100 && \text{(1's complement form)} \\ &\quad +1 && \text{(add 1)} \\ &\overline{1010101} && \text{2's complement form} \end{aligned}$$

Table 1.5 shows the comparisons of four representations of 3-bit binary numbers. The bit positions in a 2's complement number have the same weight as in a conventional binary number except that the weight of the sign bit is negative. For example, the 2's complement number 1000011 can be converted to decimal in the same manner as a binary number:

$$\begin{aligned} -2^6 + 2^1 + 2^0 &= -64 + 2 + 1 \\ &= -61 \end{aligned}$$

A distinct advantage of 2's complement form is that, unlike 1's complement form, there is a unique representation of 0 as can be seen in Table 1.5. Moreover, addition and subtraction can be treated as the same operation as in 1's complement; however, the carry bit can be ignored and the result is always in correct 2's complement notation. Thus addition and subtraction are easier in 2's complement than in 1's complement.

An *overflow* occurs when two 2's complement numbers are added, if the carry-in bit into the sign bit is different from the carry-out bit from the sign bit. For example, the

TABLE 1.5 Various Representations of 3-Bit Binary Numbers

	Decimal Equivalent			
Binary	Unsigned	Signed	1's Complement	2's Complement
000	0	+0	+0	+0
001	1	+1	+1	+1
010	2	+2	+2	+2
011	3	+3	+3	+3
100	4	−0	−3	−4
101	5	−1	−2	−3
110	6	−2	−1	−2
111	7	−3	−0	−1

following addition will result in an overflow:

```
                0 ◄── Carry-in
              101010        (−22)
              101001        (−23)
             1010011
Carry-out ──┘
```

Hence the result is invalid.

Example 1.24 Let us derive the following using 2's complement representation:

(i) $+13$ -7 (ii) -15 -6 (iii) -9 $+6$ (iv) -5 -1

```
                1 ◄── Carry-in
(i)   +13      01101
      −7       11001
      +6      100110
Carry-out ──┘
```

Since the carry-in is equal to the carry-out, there is no overflow; the carry-out bit can be ignored. The sign bit is positive. Thus the result is $+6$.

```
                      0 ◄── Carry-in
(ii)    −15          10001
        −6           11010
                    101011
Carry-out ──┘
```

The carry-out bit is not equal to the carry-in bit. Thus there is an overflow and the result is invalid.

```
                      0 ◄── Carry-in
(iii)   −9           10111
        +6           00110
        −3          011101
Carry-out ──┘
```

There is no overflow, and the sign bit is negative. The decimal equivalent of the result is $-2^4 + 2^3 + 2^2 + 2^0 = -3$.

```
                      1 ◄── Carry-in
(iv)    −5           1011
        −1           1111
        −6          11010
Carry-out ──┘
```

There is no overflow, and the sign bit is negative. The result, as expected, is $-2^3 + 2^1 = -6$.

An important advantage of 2's complement numbers is that they can be *sign-extended* without changing their values. For example, if the 2's complement number 101101 is

shifted right (i.e., the number becomes 1101101), the decimal value of the original and the shifted number remains the same (−19 in this case).

1.7 FLOATING-POINT NUMBERS

Thus far, we have been dealing mainly with fixed-point numbers in out discussion. The word *fixed* refers to the fact that the radix point is placed at a fixed place in each number, usually either to the left of the most significant digit or to the right of the least significant digit. With such a representation, the number is always either a fraction or an integer. The main difficulty of fixed-point arithmetic is that the range of numbers that could be represented is limited. Figure 1.1 illustrates the fixed-point representation of a signed four-digit decimal number; the range of numbers that can be represented using this configuration is 9999. In order to satisfy this limited range, scaling has to be used.

For example, to add +50.73 to +40.24 we have to multiply both numbers by 100 before addition and then adjust the sum, keeping in mind that there should be a decimal point two places from the right. The scale factor in this example is 100.

An alternative representation of numbers, known as the *floating-point* format, may be employed to eliminate the scaling factor problem. In a floating-point number, the radix point is not placed at a fixed place; instead, it "floats" to various places in a number so that more digits can be assigned to the left or to the right of the point. More digits on the left of the radix point permit the representation of larger numbers, whereas more digits on the right of the radix point result in more digits for the fraction. Numbers in floating point format consist of two parts—a fraction and an exponent; they can be expressed in the form

$$\text{fraction} \times \text{radix}^{\text{exponent}}$$

The fraction is often referred to as the *mantissa* and can be represented in sign-magnitude, diminished radix complement, or radix complement form. For example, the decimal number 236,000 can be written 0.236×10^6. In a similar manner, very small numbers may be represented using negative exponents. For example, 0.00000012 may be written 0.12×10^{-6}. By adjusting the magnitude of the exponent, the range of numbers covered can be considerably enlarged. Leading 0's in a floating-point number may be removed by shifting the mantissa to the left and decreasing the exponent accordingly; this process is known as *normalization* and floating-point numbers without leading 0's are called normalized. For example, the normalized of floating-point number 0.00312×10^5 is 0.312×10^3. Similarly, a binary fraction such as 0.001×2^4 would be normalized to 0.1×2^2.

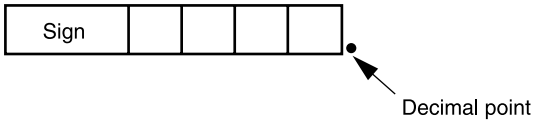


FIGURE 1.1 Fixed-point number representation.

1.8 BINARY ENCODING

In Section 1.1 it was shown how decimal numbers can be represented by equivalent binary numbers. Since there are ten decimal numbers (0, 1, . . . , 9), the minimum number of bits required to represent a decimal number is four, giving 16 ($=2^4$) possible combinations, of which only ten are used. Binary codes are divided into two groups—*weighted* and *nonweighted*.

1.8.1 Weighted Codes

In a weighted code each binary digit is assigned a weight w; the sum of the weights of the I bits is equal to the decimal number represented by the four-bit combination. In other words, if d_j ($i = 0 , \ldots , 3$) are the digit values and w_j ($i = 0 , \ldots , 3$) are the corresponding weights, then the decimal equivalent of a 4-bit binary number is given by

$$d_3w_3 + d_2w_2 + d_1w_1 + d_0w_0$$

If the weight assigned to each binary digit is exactly the same as that associated with each digit of a binary number (i.e., $w_0 = 2^\circ = 1$, $w_1 = 2^1 = 2$, $w_2 = 2^2 = 4$, and $w_3 = 2^3 = 8$), then the code is called the *BCD (binary-coded decimal)* code. The BCD code differs from the conventional binary number representation in that, in the BCD code, each decimal digit is binary coded. For example, the decimal number 15 in conventional binary number representation is

1111

whereas in the BCD code, 15 is represented by

```
0001 0101
'-'-'  '-'-'
  1     5
```

the decimal digits 1 and 5 each being binary coded.

Several forms of weighted codes are possible, since the codes depend on the weight assigned to the binary digits. Table 1.6 shows the decimal digits and their weighted

TABLE 1.6 Weighted Binary Codes

Decimal Number	8421	7421	4221	8421
0	0000	0000	0000	0000
1	0001	0001	0001	0111
2	0010	0010	0010	0110
3	0011	0011	0011	0101
4	0100	0100	1000	0100
5	0101	0101	0111	1011
6	0110	0110	1100	1010
7	0111	1000	1101	1001
8	1000	1001	1110	1000
9	1001	1010	1111	1111

code equivalents. The 7421 code has fourteen 1's in its representation, which is the minimum number of 1's possible. However, if we represent decimal 7 by 0111 instead of 1000, the 7421 code will have sixteen 1' instead of fourteen. In the 4221 code, the sum of the weights is exactly $9(=4+2+2+1)$. Codes whose weights add up to 9 have the property that the 9's complement of a number (i.e., $9-N$, where N is the number) represented in the code can be obtained simply by taking the 1's complement of its coded representation. For example, in the 4221 code shown in Table 1.6, the decimal number 7 is equivalent to the code word 1101; the 9's complement of 7 is 2 $(=9-7)$, and the corresponding code word is 0010, which is the 1' complement of 1101. Codes having this property are known as *self-complementing* codes. Similarly, the 8421 is also a self-complementing code.

Among the weighted codes, the BCD code is by far the most widely used. It is useful in applications where output information has to be displayed in decimal. The addition process in BCD is the same as in simple binary as long as the sum is decimal 9 or less. For example,

Decimal	BCD
6	0110
+3	+0011
9	1001

However, if the sum exceeds decimal 9, the result has to be adjusted by adding decimal 6 (0110) to it. For example, let us add 5 to 7:

Decimal	BCD	
7	0111	
+5	+0101	
12	1100	12 (not a legal BCD number)
	+0110	Add 6
	0001 0010	
	1 2	

As another example, let us add 9 to 7:

Decimal		BCD
9		1001
+7		+0111
16	0001	0000
	1	0

Although the result consists of two valid BCD numbers, the sum is incorrect. It has to be corrected by adding 6 (0110). This is required when there is a carry from the most significant bit of a BCD number to the next higher BCD number. Thus the correct result is

	0001	0000
+		0110
	0001	0110
	1	6

Other arithmetic operations in BCD can also be performed.

TABLE 1.7 Excess-3 Code

Decimal	Excess-3
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

1.8.2 Nonweighted Codes

In the nonweighted codes there are no specific weights associated with the digits, as was the case with weighted codes. A typical nonweighted code is the *excess-3* code. It is generated by adding 3 to a decimal number and then converting the result to a 4-bit binary number. For example, to encode the decimal number 6 to excess-3 code, we first add 3 to 6. The resulting sum, 9, is then represented in binary (i.e., 1001). Table 1.7 lists the excess-3 code representations of the decimal digits.

Excess-3 code is a self-complementing code and is useful in arithmetic operations. For example, consider the addition of two decimal digits whose sum will result in a number greater than 9. If we use BCD code, no carry bit will be generated. On the other hand, if excess-3 code is used, there will be a natural carry to the next higher digit; however, the sum has to be adjusted by adding 3 to it. For example, let us add 6 to 7:

Decimal	Excess-3	
7	1010	
+6	1001	
13	10011	
Carry ↗	00110011	Add 3 to both sum
	0100 0110	and carry bit
	1 3	

In excess-3 code, if we add two decimal digits whose sum is 9 or less, then the sum should be adjusted by subtracting 3 from it. For example,

Decimal	Excess-3	
6	1001	
+2	+0101	
8	1110	
	0011	Subtract 3
	1011	
	8	

For subtraction in excess-3 code, the difference should be adjusted by adding 3 to it. For example,

Decimal	Excess-3		
17	0100	1010	
−11	0100	0100	
6	0000	0110	
		0011	Add 3
		1001	
		6	

Another code that uses four unweighted binary digits to represent decimal numbers is the *cyclic code*. Cyclic codes have the unique feature that the successive codewords differ only in one bit position. Table 1.8 shows an example of such a code.

One type of cyclic code is the *reflected code*, also known as the *Gray code*. A 4-bit Gray code is shown in Table 1.9. Notice that in Table 1.9, except for the most significant bit

TABLE 1.8 Cyclic Code

Decimal	Cyclic
0	0000
1	0001
2	0011
3	0010
4	0110
5	0100
6	1100
7	1110
8	1010
9	1000

TABLE 1.9 Gray Code

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

position, all columns are "reflected" about the midpoint; in the most significant bit position, the top half is all 0's and the bottom half all 1's.

A decimal number can be converted to Gray code by first converting it to binary. The binary number is converted to the Gray code by performing a modulo-2 sum of each digit (starting with the least significant digit) with its adjacent digit. For example, if the binary representation of a decimal number is

$$b_3 \quad b_2 \quad b_1 \quad b_0$$

then the corresponding Gray code word, $G_3G_2G_1G_0$, is

$$\begin{aligned} G_3 &= b_3 \\ G_2 &= b_3 \oplus b_2 \\ G_1 &= b_2 \oplus b_1 \\ G_0 &= b_1 \oplus b_0 \end{aligned}$$

where $\oplus$ indicates exclusive-OR operation (i.e., modulo-2 addition), according to the following rules:

$$\begin{aligned} 0 \oplus 0 &= 0 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 0 &= 1 \\ 1 \oplus 1 &= 0 \end{aligned}$$

As an example let us convert decimal 14 to Gray code.

Decimal	Binary			
	b_3	b_2	b_l	b_0
14	1	1	1	0

Therefore

$$\begin{aligned} G_3 &= b_3 &&= 1 \\ G_2 &= b_3 \oplus b_2 &&= 1 \oplus 1 = 0 \\ G_1 &= b_2 \oplus b_1 &&= 1 \oplus 1 = 0 \\ G_0 &= b_1 \oplus b_0 &&= 1 \oplus 0 = 1 \end{aligned}$$

Thus the Gray code word for decimal 14 is

G_3	G_2	G_1	G_0
1	0	0	1

The conversion of a Gray code word to its decimal equivalent is done by following this sequence in reverse. In other words, the Gray code word is converted to binary and

then the resulting binary number is converted to decimal. To illustrate, let us convert 1110 from Gray to decimal:

$$
\begin{array}{rl}
 & G_3 \quad G_2 \quad G_1 \quad G_0 \\
 & 1 \quad\ 1 \quad\ 1 \quad 0 \\
b_3 = G_3 = 1 & \\
G_2 = b_3 \oplus b_2 & \\
\therefore\ b_2 = G_2 \oplus b_3 & \text{(since } G_2 \oplus b_3 = b_3 \oplus b_2 \oplus b_3\text{)} \\
= 1 \oplus 1 & \text{(since } G_2 = 1 \text{ and } b_3 = 1\text{)} \\
= 0 & \\
G_1 = b_2 \oplus b_1 & \\
\therefore\ b_1 = G_1 \oplus b_2 & \\
= 1 \oplus 0 & \\
= 1 & \\
G_0 = b_1 \oplus b_0 & \\
\therefore\ b_0 = 1 &
\end{array}
$$

Thus the binary equivalent of the Gray code word 1110 is 1011, which is equal to decimal 11. The first ten codewords of the Gray code shown in Table 1.9 can be utilized as reflected BCD code if desired. The middle ten codewords can be used as reflected excess-3 code.

EXERCISES

1. Convert the following decimal numbers to binary:
   a. 623
   b. 73.17
   c. 53.45
   d. 2.575

2. Convert the following binary numbers to decimal:
   a. 10110110
   b. 110000101
   c. 100.1101
   d. 1.001101

3. Convert the following binary numbers to hexadecimal and octal:
   a. 100101010011
   b. 001011101111
   c. 1011.111010101101
   d. 1111.100000011110

4. Convert the following octal numbers to binary and hexadecimal.
   a. 1026

b. 7456
c. 5566
d. 236.2345

5. Convert the following hexadecimal numbers to binary and octal:
a. EF69
b. 98AB5
c. DAC.IBA
d. FF.EE

6. Perform the addition of the following binary numbers:

a.	**b**.	**c**.
100011	10110110	10110011
1101	11100011	1101010

7. Perform the following subtractions, where each of the numbers is in binary form:

a.	**b**.	**c**.
101101	1010001	10000110
111110	1001111	1110001

8. Add the following pairs of numbers, where each number is in hexadecimal form:

a.	**b**.	**c**.
ABCD	129A	EF23
75EF	AB22	C89

9. Repeat Exercise 8 using subtraction instead of addition.

10. Add the following pairs of numbers, where each number is in octal form:

a.	**b**.	**c**.
7521	62354	3567
4370	3256	2750

11. Repeat Exercise 10 using subtraction instead of addition.

12. Derive the 6-bit *sign-magnitude*, *1's complement*, *and 2's complement* of the following decimal numbers:
a. $+22$
b. -31
c. $+17$
d. -1

13. Find the sum of the following pairs of decimal numbers assuming 8-bit 1's complement representation of the numbers:
a. $+61+(-23)$
b. $-56+(-55)$
c. $+28+(+27)$
d. $-48+(+35)$

14. Repeat Exercise 13 assuming 2's complement representation of the decimal numbers.

15. Assume that X is the *2's complement* of an n-bit binary number Y. Prove that the 2's complement of X is Y.

16. Find the floating-point representation of the following numbers:

a. $(326.245)_{10}$

b. $(101100.100110)_2$

c. $(-64.462)_8$

17. Normalize the following floating-point numbers:

a. 0.000612×10^6

b. 0.0000101×2^4

18. Encode each of the ten decimal digits using the weighted binary codes.

a. 4, 4, 1, −2

b. 7, 4, 2, −1

19. Given the following *weighted* codes determine whether any of these is *self-complementing*.

a. (8, 4, −3, −2)

b. (7, 5, 3, −6)

c. (6, 2, 2, 1)

20. Represent the decimal numbers 535 and 637 in

a. BCD code

b. Excess-3 code

Add the resulting numbers so that the sum in each case is appropriately encoded.

21. Subtract 423 from 721, assuming the numbers are represented in *excess-3* code.

22. Determine two forms for a cyclic code other than the one shown in Table 1.8.

23. Assign a binary code, using the minimum number of bits, to encode the 26 letters in the alphabet.