Fundamentals of Digital Logic and Microcomputer Design. M. Rafiquzzaman

2

NUMBER SYSTEMS AND CODES

In this chapter we describe some of the fundamental concepts needed to implement and use a computer effectively. Thus the basics of number systems, codes, and error detection/correction are presented.

2.1 Number Systems

A computer, like all digital machines, utilizes two states to represent information. These two states are given the symbols 0 and 1. It is important to remember that these 0's and 1's are symbols for the two states and have no inherent numerical meanings of their own. These two digits are called binary digits (bits) and can be used to represent numbers of any magnitude. The microcomputer carries out all the arithmetic and logic operations internally using binary numbers. Because binary numbers are long, a more compact form using some other number system is preferable to represent them. The computer user finds it convenient to work with this compact form. Hence, it is important to understand the various number systems used with computers. These are described in the following sections.

2.1.1 General Number Representation

In general, a number N can be represented in the following form:

$$N = d_{p-1} \times b^{p-1} + d_{p-2} \times b^{p-2} + \ldots + d_0 \times b^0 + d_{-1} \times b^{-1} + \ldots + d_{-q} \times b^{-q} \qquad \textbf{2.1}$$

where b is the base or radix of the number system, the d's are the digits of the number system, p is the number of integer digits, and q is the number of fractional digits.

N can also be written as a string of digits whose integer and fractional portions are separated by the radix or decimal point (•). In this format, the number N is represented as

$$N = d_{p-1}\, d_{p-2} \ldots d_1\, d_0 \bullet d_{-1} \ldots d_{-q} \qquad \textbf{2.2}$$

If a number has no fractional portion, (e.g., $q = 0$ in the form of Equation 2.1), then the number is called an integer number or an integer. Conversely, if the number has no integer portion (e.g., $p = 0$ in the form of Equation 2.1), the number is called a fractional number or a fraction. If both p and q are not zero, then the number is called a mixed number.

Decimal Number System

In the decimal number system (base 10), which is most familiar to us, the integer number 125_{10} can be expressed as

$$125_{10} = 1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 \qquad \textbf{2.3}$$

In this equation, the left-hand side corresponds to the form given by Equation 2.2. The right-hand side of Equation 2.3 is represented by the form of equation 2.1, where $b = 10$, $d_2 = 1$, $d_1 = 2$, $d_0 = 5$, $d_{-1} = \ldots = d_{-q} = 0$, $p = 3$, and $q = 0$.

Now, consider the fractional decimal number 0.532_{10}. This number can be expressed as

$$0.532_{10} = 5 \times 10^{-1} + 3 \times 10^{-2} + 2 \times 10^{-3} \quad \textbf{2.4}$$

The left-hand side of Equation 2.4 corresponds to Equation 2.2. The right-hand side of Equation 2.4 is in the form of Equation 2.1, where $b = 10$, $d_{-1} = 5$, $d_{-2} = 3$, $d_{-3} = 2$, $q = 3$, $p = 0$, $d_{p-1} = \ldots = d_0 = 0$.

Finally, consider the mixed number 125.532_{10}. This number is in the form of Equation 2.2. Translating the number to the form of Equation 2.1 yields

$$125.532_{10} = 1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 + 5 \times 10^{-1} + 3 \times 10^{-2} + 2 \times 10^{-3} \quad \textbf{2.5}$$

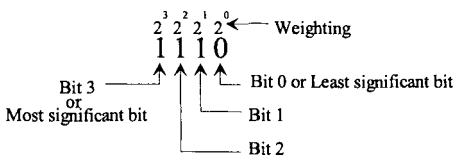
Comparing the right-hand side of Equation 2.5 with equation 2.1 yields $b = 10$, $p = 3$, $q = 3$, $d_2 = 1$, $d_1 = 2$, $d_0 = 5$, $d_{-1} = 5$, $d_{-2} = 3$, and $d_{-3} = 2$.

Binary Number System

In terms of Equation 2.1, the binary number system has a base or radix of 2 and has two allowable digits, 0 and 1. From Equation 2.1, a 4-bit binary number 1110_2 can be interpreted as

$$1110_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 14_{10}$$

This conversion from binary to decimal can be obtained by inspecting the binary number as follows:



Note that bits 0, 1, 2, and 3 have corresponding weighting values of 1, 2, 4, and 8. Because a binary number only contains 1's and 0's, adding the weighting values of only the bits of the binary number containing 1's will provide its decimal value. The decimal value of 1110_2 is 14_{10} (2 + 4 + 8), because bits 1, 2, and 3 have binary digit 1, whereas bit 0 contains 0.

Therefore, the decimal value of any binary number can be readily obtained by just adding the weighting values for the bit positions containing 1's. Furthermore, the value of the least significant bit (bit 0) determines whether the number is odd or even. For example, if the least significant bit is 1, the number is odd; otherwise, the number is even.

Next, consider a mixed number 101.01_2 as follows:

$$101.01_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} \quad \textbf{2.6}$$

The decimal or base 10 value of 101.01_2 is found from the right-hand side of Equation 2.6 as $4 + 0 + 1 + 0 + 1/4 = 5.25_{10}$.

Octal Number System

The radix or base of the octal number system is 8. There are eight digits, 0 through 7, allowed in this number system.

Consider the octal number 25.32_8, which can be interpreted as:

$$2 \times 8^1 + 5 \times 8^0 + 3 \times 8^{-1} + 2 \times 8^{-2}$$

The decimal value of this number is found by completing the summation of

$$16 + 5 + 3 \times 1/8 + 2 \times 1/64 = 16 + 5 + 0.375 + 0.03125 = 21.40625_{10}$$

One can convert a number from binary to octal representation easily by taking the binary digits in groups of 3 bits.

The octal digit is obtained by considering each group of 3 bits as a separate binary number capable of representing the octal digits 0 through 7. The radix point remains in its original position. The following example illustrates the procedure.

Suppose that it is desired to convert 1001.11_2 into octal form. First take the groups of 3 bits starting at the radix point. Where there are not enough leading or trailing bits to complete the triplet, 0's are appended. Now each group of 3 bits is converted to its corresponding octal digit.

$$\underbrace{001}_{1}\ \underbrace{001}_{1}\ .\ \underbrace{110_2}_{6} = 11.6_8$$

The conversion back to binary from octal is simply the reverse of the binary-to-octal process. For example, conversion from 11.6_8 to binary is accomplished by expanding each octal digit to its equivalent binary values as shown:

$$\overbrace{1}^{001}\ \overbrace{1}^{001}\ .\ \overbrace{6}^{110}$$

Hexadecimal Number System

The hexadecimal or base-16 number system has 16 individual digits. Each of these digits, as in all number systems, must be represented by a single unique symbol. The digits in the hexadecimal number system are 0 through 9 and the letters A through F. Letters were chosen to represent the hexadecimal digits greater than 9 because a single symbol is required for each digit. Table 2.1 lists the 16 digits of the hexadecimal number system and their corresponding binary and decimal values.

TABLE 2.1 Number Systems

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

2.1.2 Converting Numbers from One Base to Another

Binary-to-Decimal Conversion and Vice Versa

Consider converting 1100.01_2 to its decimal equivalent. As before,

$$1100.01_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$$
$$= 8 + 4 + 0 + 0 + 0 + .25$$
$$= 12.25_{10}$$

Continuous division by 2, keeping track of the remainders, provides a simple method of converting a decimal number to its binary equivalent. As an example, to convert decimal 12_{10} to its binary equivalent 1100_2, proceed as follows:

	quotient	+	remainder
$\frac{12}{2}$ =	6	+	0
$\frac{6}{2}$ =	3	+	0
$\frac{3}{2}$ =	1	+	1
$\frac{1}{2}$ =	0	+	1

$1\ 1\ 0\ 0_2$

Fractions

One can convert 0.0101_2 to its decimal equivalent as follows:

$$0.0101_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$$
$$= 0 + 0.25 + 0 + 0.0625$$
$$= 0.3125_{10}$$

A decimal fractional number can be converted to its binary equivalent as follows:

0.8125	0.6250	0.2500	0.5000
$\times 2$	$\times 2$	$\times 2$	$\times 2$
(1).6250	(1).2500	(0).5000	(1).0000
↓	↓	↓	↓
1	1	0	1

Therefore $0.8125_{10} = 0.1101_2$.

Unfortunately, binary-to-decimal fractional conversions are not always exact. Suppose that it is desired to convert 0.3615 into its binary equivalent:

0.3615	0.7230	0.4460	0.8920	0.7840
$\times 2$	$\times 2$	$\times 2$	$\times 2$	$\times 2$
(0).7230	(1).4460	(0).8920	(1).7840	(1).5680
↓	↓	↓	↓	↓
0	1	0	1	1

The answer is $0.01011\ldots_2$. As a check, let us convert back:

$$0.01011_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5}$$
$$= 0 + 0.25 + 0 + 0.0625 + 0.03125$$
$$= 0.34375$$

The difference is $0.3615 - 0.34375 = 0.01775$. This difference is caused by the neglected remainder 0.5680. The neglected remainder (0.5680) multiplied by the smallest computed term (0.03125) gives the total error:

$$0.5680 \times 0.03125 = 0.01775$$

Mixed Numbers

Finally, convert 13.25_{10} to its binary equivalent. It is convenient to carry out separate conversions for the integer and fractional parts. Consider first the integer number 13 as before:

		quotient	+	remainder
$\frac{13}{2}$	=	6	+	1
$\frac{6}{2}$	=	3	+	0
$\frac{3}{2}$	=	1	+	1
$\frac{1}{2}$	=	0	+	1
			13_{10} =	$1\ 1\ 0\ 1_2$

Now convert the fractionl part 0.25_{10} as follows:

0.25	0.50
$\times 2$	$\times 2$
0.50	1.00
0	1

Thus $0.25_{10} = 0.01_2$. Therefore $13.25_{10} = 1101.01_2$.

Note that the same procedure applies for converting a decimal integer number to other number systems such as octal or hexadecimal; Continuous division by the appropriate base (8 or 16) and keeping track of remainders converts a decimal number from decimal to the selected number system.

Binary-to-Hexadecimal Conversion and Vice Versa

The conversions between hexadecimal and binary numbers are done in exactly the same manner as the conversions between octal and binary, except that groups of 4 are used. The following examples illustrate this:

$$1011011_2 = \underbrace{0101}_{5}\ \underbrace{1011}_{B} = 5B_{16}$$

Note that the binary integer number is grouped in 4-bit units, starting from the least significant bit. Zeros are added with the most significant 4 bits if necessary. As with octal numbers, for fractional numbers this grouping into 4 bits is started from the radix point. Now consider converting $2AB_{16}$ into its binary equivalent as follows:

$$2AB_{16} = \begin{array}{ccc} 2 & A & B \\ \downarrow & \downarrow & \downarrow \\ 0010 & 1010 & 1011 \end{array}$$
$$= 001010101011_2$$

Hexadecimal-to-Decimal Conversion and Vice Versa

Consider converting the hexadecimal number $23A_{16}$ into its decimal equivalent and vice versa. This can be accomplished as follows:

$$23A_{16} = 2 \times 16^2 + 3 \times 16^1 + 10 \times 16^0$$
$$= 512 + 48 + 10 = 570_{10}$$

Note that in the equation, the value 10 is substituted for A.
Now to convert 570_{10} back to $23A_{16}$,

	quotient	+	remainder
$\frac{570}{16} =$	35	+	A
$\frac{35}{16} =$	2	+	3
$\frac{2}{16} =$	0	+	2

2 3 A

Thus, $570_{10} = 23A_{16}$.

Example 2.1

Determine by inspecting the binary equivalent of the following hexadecimal numbers whether they are odd or even. Then verify the result by their decimal equivalents.

(a) $2B_{16}$ (b) $A2_{16}$

Solution

(a)

	128	64	32	16	8	4	2	1	← Weighting
$2B_{16} =$	0	0	1	0	1	0	1	1_2	

The number is odd, since the least significant bit is 1.

Decimal value = $32 + 8 + 2 + 1 = 43_{10}$, which is odd.

(b)

	128	64	32	16	8	4	2	1	← Weighting
$A2_{16} =$	1	0	1	0	0	0	1	0_2	

The number is even, since the least significant bit is 0.

Decimal value = $128 + 32 + 2 = 162_{10}$, which is even.

2.2 Unsigned and Signed Binary Numbers

An unsigned binary number has no arithmetic sign. Unsigned binary numbers are therefore always positive. Typical examples are your age or a memory address which are always

positive numbers. An 8-bit unsigned binary integer represents all numbers from 00_{16} through FF_{16} (0_{10} through 255_{10}).

The techniques used to represent the signed integers are:

- Sign-magnitude approach
- Ones complement approach
- Twos complement approach

Because the sign of a number can be either positive or negative, only one bit, referred to as the sign bit, is needed to represent the sign. The widely used sign convention is that if the sign bit is zero, the number is positive; otherwise it is negative. (The rationale behind this convention is that the quantity $(-1)^s$ is positive when s = 0 and is negative when s = 1). Also, in all three approaches, the most significant bit of the number is considered to be the sign bit.

In sign-magnitude representation, the most significant bit of the given n-bit binary number holds the sign, and the remaining $n - 1$ bits directly give the magnitude of the negative number. For example, the sign-magnitude representation of +7 is 0111 and that of −4 is 1100. Table 2.2 represents all possible 4-bit patterns and their meanings in sign-magnitude form.

In Table 2.2, the sign-magnitude approach represents a signed number in a natural manner. With 4 bits we can only represent numbers in the range $-7 \leq x \leq +7$. In general, if there are n bits, then we can cover all numbers in the range $\pm(2^{n-1} - 1)$. Note that with $n - 1$ bits, any value from 0 to $2^{n-1} - 1$ can be represented. However, this approach leads to a confusion because there are two representations for the number zero (0000 means +0; 1000 means −0).

In the complement approach, positive numbers have the same representation as they do in the sign-magnitude representation. However, in this technique negative numbers are represented in a different manner. Before we proceed, let us define the term *complement* of a number. The complement of a number A, written as $\overline{A}$ (or A') is obtained by taking bit-by-bit complement of A. In other words, each 0 in A is replaced with 1 and vice versa. For example, the complement of the number 0100_2 is 1011_2 and that of 1111_2 is 0000_2. In the ones complement approach, a negative number, $-x$, is the complement of its positive

TABLE 2.2 All Possible 4-Bit Integers Represented in Sign-Magnitude Form

Bit Pattern	Interpretation as a Sign-Magnitude Integer
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	−0
1001	−1
1010	−2
1011	−3
1100	−4
1101	−5
1110	−6
1111	−7

TABLE 2.3 All Possible 4-Bit Integers Represented in Ones Complement Form

Bit Pattern	Interpretation as a Ones Complement Number
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	−7
1001	−6
1010	−5
1011	−4
1100	−3
1101	−2
1110	−1
1111	−0

representation. For example let us find the ones complement representation of 0100_2 ($+4_{10}$). The complement of 0100 is 1011, and this denotes the negative number -4_{10}. Table 2.3 summarizes all possible 4-bit binary patterns and their interpretations as ones complement numbers.

From Table 2.3, the ones complement approach does not handle negative numbers naturally. In other words, if the number is negative (when the sign bit is 1), its magnitude is not obvious from its ones complement. To determine its magnitude, one needs to take its ones complement. For example, consider the number 110110. The most significant bit indicates that this is a negative number. Because the number is negative, its magnitude cannot be obtained by directly looking at 110110. Instead, one needs to take the ones complement of 110110 to obtain 001001. The value of 001001 as a sign-magnitude number is +9. On the other hand, 110110 represents −9 in ones complement form. Like the sign-magnitude representation, the ones complement approach does not increase the range of numbers covered by a fixed number of bit patterns. For example, 4 bits cover the range −7 to +7. The same range is obtained with sign-magnitude representation. Note that the confusion of two distinct representations for zero exists in the ones complement approach.

Now, let us discuss the two's complement approach. In this method, positive integers are represented in the same manner as they are in the sign-magnitude method. In other words, if the sign bit is zero, the number is positive and its magnitude can be directly obtained by looking at the remaining $n - 1$ bits. However, a negative number $-x$ can be represented in twos complement form as follows:

- Represent $+x$ in sign magnitude form and call this result y
- Take the ones complement of y to get $\bar{y}$ (or y')
- $\bar{y} + 1$ is the twos complement representation of $-x$.

The following example illustrates this:

Table 2.4 lists all possible 4-bit patterns along with their twos complement forms. From Table 2.4, it can be concluded that:

- The twos complement form does not provide two representations for zero.
- The twos complement form covers up to -8 in the negative side, and this is more than can be achieved with the other two methods. In general, with n bits, and using twos complement approach, one can cover all the numbers in the range $-(2^{n-1})$ to $+(2^{n-1} - 1)$.

It should be pointed out that 11111111_2 is $+255_{10}$ when interpreted as an unsigned number. On the other hand, 11111111_2 is -1_{10} when interpreted as a signed number. Note that typical 16-bit microprocessors have separate unsigned and signed multiplication and division instructions. Suppose that a microprocessor has the following multiplication and division instructions: MULU (Multiply two unsigned numbers), MULS (Multiply two signed numbers), DIVU (Divide two unsigned numbers), and DIVS (Divide two signed numbers). It is important for the programmer to clearly understand how to use these instructions.

For example, suppose that it is desired to compute $(X^2)/255$. Now, if X is a signed 8-bit number, the programmer should use the MULS instruction to compute X * X which is always unsigned (square of a number is always positive), and then use DIVU to compute $(X^2)/255$ (16-bit by 8-bit unsigned divide) since 255_{10} is positive. But, if the programmer uses DIVS, then both X * X and 255_{10} (FF_{16}) will be interpreted as signed numbers. FF_{16} will be interpreted as -1_{10}, using two's complement. and the result will be wrong. On the other hand, if X is an unsigned number, the programmer needs to use MULU and DIVU to compute $(X^2)/255$.

Example 2.2

Represent the following decimal numbers in twos complement form. Use 7 bits to represent the numbers:

(a) +39

(b) -43

Solution

(a) Because the number +39 is positive, its twos complement representation is the same as its sign-magnitude representation as shown here:

$$y = \underbrace{0}_{+}\ \underbrace{\overset{2^5}{1}\ \overset{2^4}{0}\ \overset{2^3}{0}\ \overset{2^2}{1}\ \overset{2^1}{1}\ \overset{2^0}{1}}_{39}$$

(b) In this case, the given number -43 is negative. The twos complement form of the number can be obtained as follows:

Step 1: Represent +43 in sign magnitude form

$$y = \underbrace{0}_{+}\ \underbrace{\overset{2^5}{1}\ \overset{2^4}{0}\ \overset{2^3}{1}\ \overset{2^2}{0}\ \overset{2^1}{1}\ \overset{2^0}{1}}_{43}$$

Step 2: Take the ones complement of y:

$$\bar{y} = 1\ 0\ 1\ 0\ 1\ 0\ 0$$

Step 3: Add one to $\bar{y}$ to get the final answer.

$$\begin{array}{r} 1010100 \\ +\quad\quad 1 \\ \hline 1010101 \end{array}$$

TABLE 2.4 All Possible 4-Bit Integers Represented in Twos Complement Form

Bit Pattern	Interpretation as a Twos Complement Number
0000	0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	−8
1001	−7
1010	−6
1011	−5
1100	−4
1101	−3
1110	−2
1111	−1

2.3 Codes

Codes are used extensively with computers to define alphanumeric characters and other information. Some of the codes used with computers are described in the following sections.

2.3.1 Binary-Coded-Decimal Code (8421 Code)

The 10 decimal digits 0 through 9 can be represented by their corresponding 4-bit binary numbers. The digits coded in this fashion are called binary-coded-decimal (BCD) digits in 8421 code, or BCD digits. Two unpacked BCD bytes are usually packed into a byte to form "packed BCD." For example, two unpacked BCD bytes 02_{16} and 05_{16} can be combined as a packed BCD byte 25_{16}. The concept of packed and unpacked BCD numbers are explained later in this section. Table 2.5 provides the bit encodings of the 10 decimal numbers.

The six possible remaining 4-bit codes as shown in Table 2.5 are not used and represent invalid BCD codes if they occur.

Consider obtaining the BCD bit encoding of the decimal number 356 as follows:

$$\underbrace{3}_{\downarrow 0011}\quad\underbrace{5}_{\downarrow 0101}\quad\underbrace{6}_{\downarrow 0110}$$

2.3.2 Alphanumeric Codes

A computer must be capable of handling nonnumeric information if it is to be very useful. In other words, a computer must be able to recognize codes that represent numbers, letters, and special characters. These codes are classified as alphanumeric or character codes. A complete and adequate set of necessary characters includes these:

1. 26 lowercase letters

TABLE 2.5 BCD Bit encodings of the 10 decimal numbers

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

2. 26 uppercase letters
3. 10 numeric digits (0−9)
4. About 25 special characters, which include + / # % , and so on.

This totals 87 characters. To represent 87 characters with some type of binary code would require at least 7 bits. With 7 bits there are $2^7 = 128$ possible binary numbers; 87 of these combinations of 0 and 1 bits serve as the code groups representing the 87 different characters.

The 8-bit byte has been universally accepted as the data unit for representing character codes. The two most common alphanumeric codes are known as the American Standard Code for Information Interchange (ASCII) and the Extended Binary-Coded Decimal Interchange Code (EBCDIC). ASCII is typically used with microprocessors. IBM uses EBCDIC code. Eight bits are used to represent characters, although 7 bits suffice, because the eighth bit is frequently used to test for errors and is referred to as a parity bit. It can be set to 1 or 0, so that the number of 1 bits in the byte is always odd or even.

Table 2.6 shows a list of ASCII and EBCDIC codes. Some EBCDIC codes do not have corresponding ASCII codes. Note that decimal digits 0 through 9 are represented by 30_{16} through 39_{16} in ASCII. On the other hand, these decimal digits are represented by $F0_{16}$ though $F9_{16}$ in EBCDIC.

A computer program is usually written for code conversion when input/output devices of different codes are connected to the computer. For example, suppose it is desired to enter a number 5 into a computer via an ASCII keyboard and print this data on an EBCDIC printer. The ASCII keyboard will generate 35_{16} when the number 5 is pushed. The ASCII code 35_{16} for the decimal digit 5 enters into the computer and resides

TABLE 2.6 ASCII and EBCDIC Codes in Hex.

Character	ASCII	EBCDIC	Character	ASCII	EBCDIC	Character	ASCII	EBCDIC	Character	ASCII	EBCDIC
@	40			60		blank	20	40	NUL	00	
A	41	C1	a	61	81	!	21	5A	SOH	01	
B	42	C2	b	62	82	"	22	7F	STX	02	
C	43	C3	c	63	83	#	23	7B	ETX	03	
D	44	C4	d	64	84	$	24	5B	EOT	04	37
E	45	C5	e	65	85	%	25	6C	ENQ	05	
F	46	C6	f	66	86	&	26	50	ACK	06	
G	47	C7	g	67	87	'	27	7D	BEL	07	
H	48	C8	h	68	88	(	28	4D	BS	08	16
I	49	C9	i	69	89	)	29	5D	HT	09	05
J	4A	D1	j	6A	91	*	2A	5C	LF	0A	25
K	4B	D2	k	6B	92	+	2B	4E	VT	0B	
L	4C	D3	l	6C	93	,	2C	6B	FF	0C	
M	4D	D4	m	6D	94	-	2D	60	CR	0D	15
N	4E	D5	n	6E	95	.	2E	4B	SO	0E	
O	4F	D6	o	6F	96	/	2F	61	SI	0F	
P	50	D7	p	70	97	0	30	F0	DLE	10	
Q	51	D8	q	71	98	1	31	F1	DC1	11	
R	52	D9	r	72	99	2	32	F2	DC2	12	
S	53	E2	s	73	A2	3	33	F3	DC3	13	
T	54	E3	t	74	A3	4	34	F4	DC4	14	
U	55	E4	u	75	A4	5	35	F5	NAK	15	
V	56	E5	v	76	A5	6	36	F6	SYN	16	
W	57	E6	w	77	A6	7	37	F7	ETB	17	
X	58	E7	x	78	A7	8	38	F8	CAN	18	
Y	59	E8	y	79	A8	9	39	F9	EM	19	
Z	5A	E9	z	7A	A9	:	3A		SUB	1A	
[	5B		{	7B		;	3B	5E	ESC	1B	
\	5C		|	7C	4F	<	3C	4C	FS	1C	
]	5D		}	7D		=	3D	7E	GS	1D	
^	5E		~	7E		>	3E	6E	RS	1E	
_	5F	6D	DEL	7F	07	?	3F	6F	US	1F	

in the computer's memory. To print the digit 5 on the EBCDIC printer, a program must be written that will convert the ASCII code 35_{16} for 5 to its EBCDIC code $F5_{16}$. The output of this program is $F5_{16}$. This will be input to the EBCDIC printer. Because the printer only understands EBCDIC codes, it inputs the EBCDIC code $F5_{16}$ and prints the digit 5.

Let us now discuss packed and unpacked BCD codes in more detail. For example, in order to enter 24 in decimal into a computer, the two keys (2 and 4) will be pushed on the ASCII keyboard. This will generate 32 and 34 (32 and 34 are ASCII codes in hexadecimal for 2 and 4 respectively) inside the computer. A program can be written to convert these ASCII codes into unpacked BCD 02 and 04, and then convert to packed BCD 24 or to binary inside the computer to perform the desired operation.

2.3.3 Excess-3 Code

The excess-3 representation of a decimal digit *d* can be obtained by adding 3 to its value. All decimal digits and their excess-3 representations are listed in Table 2.7.

The excess-3 code is an unweighted code because its value is obtained by adding three to the corresponding binary value. The excess-3 code is self-complementing. For example, decimal digit 0 in excess-3 (0011) is ones complement of 9 in excess three (1100). Similarly, decimal digit 1 is ones complement of 8, and so on. This is why some older computers used

TABLE 2.7 Excess-3 Representation of Decimal Digits

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

excess three code. Conversion between excess-3 and decimal numbers is illustrated below:

Decimal number	1	9	8	3
Excess-3 Representation	0100	1100	1011	0110

2.3.4 Gray Code

Sometimes codes can also be constructed using a property called reflected symmetry. One such code is known as the Gray code. The Gray code is used in Karnaugh maps for simplifying combinational logic design. This topic is covered in Chapter 4. Before we proceed, we briefly explain the concept of reflected symmetry. Consider the two bits 0 and 1, and stack these two bits. Assume that there is a plane mirror in front of this stack and produce the reflected image of the stack as shown in the following:

```
      0
      1
mirror← ——
      1
      0
```

Appending a zero to all elements of the stack above the plane mirror and appending a one to all elements of the stack that lies below the mirror will provide the following result:

```
Appended zeros { 0 0
               { 0 1
               ———
Appended ones  { 1 1
               { 1 0

0 0 0                  0 0 0
0 0 1                  0 0 1
0 1 1      Mirror      0 1 1
0 1 0 ↙                0 1 0
      ——————————→
1 1 0 ↘ Result         1 1 0
1 1 1 ↗ after moving   1 1 1
1 0 1   the mirror     1 0 1
1 0 0                  1 0 0
```

FIGURE 2.1 The process of obtaining 3-bit reflected binary code

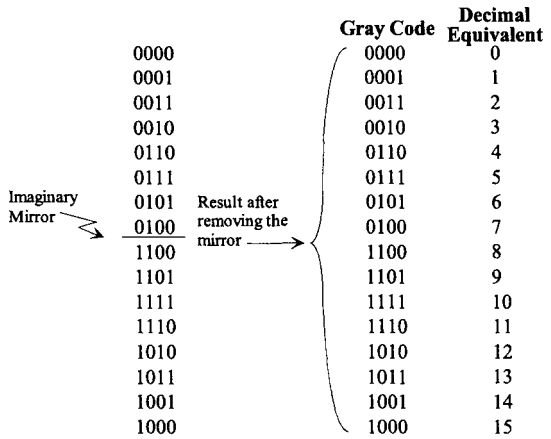


FIGURE 2.2 The process of obtaining a 4-bit Gray code from a 3-bit Gray code.

Now, removal of the plane mirror will result in a stack of 2-bit Gray Code as follows:

0 0
0 1
1 1
1 0

Here, any two adjacent bit patterns differ only in one bit. For example, the patterns 11 and 10 differ only in the least significant bit.

Repeating the reflection operation on the stack of 2-bit binary patterns, a 3-bit Gray code can be obtained. Two adjacent binary numbers differ in only one bit. The result is shown in Figure 2.1.

Applying the reflection process to the 3-bit Gray code, 4-bit Gray Code can be obtained. This is shown in Figure 2.2.

The Gray code is useful in instrumentation systems to digitally represent the position of a mechanical shaft. In these applications, one bit change between characters is required. For example, suppose that a shaft is divided into eight segments and each shaft is assigned a number. If binary numbers are used, an error may occur while changing segment 7 (0111_2) to segment 8 (1000_2). In this case, all 4 bits need to be changed. If the sensor representing the most significant bit takes longer to change, the result will be 0000_2, representing segment 0. This can be avoided by using Gray code, in which only one bit changes when going from one number to the next.

2.3.5 Unicode

Basically, computers work with numbers. Note that letters and other characters are stored in computers as numbers; a number is assigned to each one of them.

Before the invention of unicode, there were numerous encoding systems for assigning these numbers. It is not possible for a single encoding system to cover all the languages in the world. For example, a single encoding system was not able to assign all the letters, punctuation, and common technical symbols. Typical encoding systems can

conflict with each other. For example, two different characters can be assigned with the same number in two different encoding systems. Also, different numbers can be assigned the same character in two different encodings. These types of assignments of numbers can create problems for certain computers such as servers which need to support several different encodings. Hence, when data is transferred between different encodings or platforms, the data may be corrupted.

Unicode avoids this by assigning a unique number to each character regardless of the platform, the program, or the language. More information on Unicode can be obtained at the Web site at *www.unicode.org*.

2.4 Fixed-Point and Floating-Point Representations

A number representation assuming a fixed location of the radix point is called *fixed-point representation*. The range of numbers that can be represented in fixed-point notation is severely limited. The following numbers are examples of fixed-point numbers:

$$0110.1100_2,\ 51.12_{10},\ DE.2A_{16}$$

In typical scientific computations, the range of numbers is very large. Floating-point representation is used to handle such ranges. A floating-point number is represented as $N \times r^p$, where N is the mantissa or significand, r is the base or radix of the number system, and p is the exponent or power to which r is raised.

Some examples of numbers in floating-point notation and their fixed-point decimal equivalents are:

fixed-point numbers	floating-point representation
0.0167_{10}	0.167×10^{-1}
1101.10_{12}	0.1101101×2^4
$BE.2A9_{16}$	$0.BE2A9 \times 16^2$

In converting from fixed-point to floating-point number representation, we normalize the resulting mantissas; that is, the digits of the fixed-point numbers are shifted so that the highest-order nonzero digit appears to the right of the decimal point, and consequently a 0 always appears to the left of the decimal point. This convention is normally adopted in floating-point number representation. Because all numbers will be assumed to be in normalized form, the binary point is not required to be represented in computers.

Typical 32-bit microprocessors such as the Intel 80486/Pentium and the Motorola 68040 and PowerPC contain on-chip floating-point hardware. This means that these microprocessors can be programmed using instructions to perform operations such as addition, subtraction, multiplication, and division using floating-point numbers.

2.5 Arithmetic Operations

As mentioned before, computers can only add. Therefore, all other arithmetic operations are typically accomplished via addition. All numbers inside the computer are in binary form. These numbers are usually treated internally as integers, and any fractional arithmetic must be implemented by the programmer in the program. The arithmetic and logic unit (ALU) in the computer's CPU performs typical arithmetic and logic operations. The ALUs perform function such as addition, subtraction, magnitude comparison, ANDing, and ORing of two binary or packed BCD numbers. The procedures involved in executing these functions are

discussed next to provide an understanding of the basic arithmetic operations performed in a typical microprocessor. The logic operations are covered in Chapter 3

2.5.1 Binary Arithmetic

Addition

The addition of two binary numbers is carried out in the same way as the addition of decimal numbers. However, only four possible combinations can occur when adding two binary digits (bits):

augend	+	addend	=	carry	sum	decimal value
0	+	0	=	0	0	0
1	+	0	=	0	1	1
0	+	1	=	0	1	1
1	+	1	=	1	0	2

The following are some examples of binary addition. The corresponding decimal additions are also included.

```
  010  (2)          111 ← carry
+ 011  (3)          101.11   (5.75)
  101  (5)        + 011.10   (3.50)
                   1 001.01   (9.25)
                  ↗
              final carry
```

Addition is the most important arithmetic operation in microprocessors because the operations of subtraction, multiplication, and division as they are performed in most modern digital computers use only addition as their basic operation.

The addition of two unsigned numbers is performed in the same way as illustrated above. Also, the addition of two numbers in the sign-magnitude form is performed in the same manner as ordinary arithmetic. For example, if both numbers have the same signs, the two numbers are added and the common sign is assigned to the result. On the other hand, if the numbers have opposite signs, the number with smaller magnitude is subtracted from the number with larger magnitude and the result is assigned with the sign of the number with larger magnitude. For example, (-14) + (+18) = + (18 - 14) = +4. This is performed by subtracting the smaller magnitude 14 from the higher magnitude 18 and the sign of the larger magnitude 18 (+ in this case) is assigned to the result. The same rules apply to binary numbers in sign-magnitude form.

Subtraction

As mentioned before, computers can usually only add binary digits; they cannot directly subtract. Therefore, the operation of subtraction in microprocessors is performed using the operation of addition using complement arithmetic. In general, the b's complement of an m-digit number, M is defined as $b^m - M$ for $M \neq 0$ and 0 for $M = 0$. Note that for base 10, $b = 10$ and 10^m is a decimal number with a 1 followed by m 0's. For example, 10^4 is 10000; 1 followed by four 0's. On the other hand, $b = 2$ for binary and 2^m indicates 1 followed by m 0's. For example, 2^3 means 1000 in binary.

The $(b-1)$'s complement of an m-digit number, M is defined as $(b^m - 1) - M$.

Therefore, the b's complement of an m-digit number, M can be obtained by adding 1 to its $(b-1)$'s complement. Next, let us illustrate the concept of complement arithmetic by means of some examples. Consider a 4-digit decimal number, 5786. In this case, $b=10$ for base 10 and $m=4$ since there are four digits.

10's complement of $5786 = 10^4 - 5786 = 10000 - 5786 = 4214$

Now, let us obtain 10's complement of 5786 using $(10-1)$'s or 9's complement arithmetic as follows: 9's complement of $5786 = (10^4 - 1) - 5786 = 9999 - 5786 = 4213$

Hence, 10's complement of 5786 = 9's complement of 5786 + 1 = 4213 + 1 = 4214.

Next, let us determine the 2's complement of a 3-bit binary number, 010. In this case, $b = 2$ for binary and $m = 3$ since there are three bits in the number.

2's complement of $010 = 2^3 - 010 = 1000 - 010$.

Using paper and pencil method, the result of subtraction can be obtained as follows:

$$\begin{array}{r} 1000_2 \\ -010_2 \\ \hline 110_2 \end{array}$$

Note that in the above, 110_2 is -2 in decimal when interpreted as a signed number. Therefore, 2's complement of a number negates the number being complemented. This will be explained later in this section.

The 2's complement of 010 can be obtained using its 1's complement arithmetic as follows:

1's complement of $010 = (2^3 - 1) - 010 = 111 - 010 = 101$

2's complement of $101 = 101 + 1 = 110$

From the above procedure for obtaining the 1's complement of 010, it can be concluded that the 1's complement of a binary number can be achieved by subtracting each bit of the binary number from 1. This means that when subtracting a bit (0 or 1) from 1, one can have either $1 - 0 = 1$ or $1 - 1 = 0$; that is, the 1's complement of 0 is 1 and the 1's complement of 1 is 0. In general, the 1's complement of a binary number can be obtained by changing 0's to 1's and 1's to 0's.

The procedure for performing X-Y (both X and Y are in base 2) using 1's complement can be performed as follows:

Step 1. Add the minuend X to the 1's complement of the subtrahend Y.

Step 2. Check the result in step 1 for a carry. If there is a carry, add 1 to the least significant bit to obtain the result. If there is no carry, take the 1's complement of the number obtained in step 1 and place a negative sign in front of the result.

For example, consider two 6-bit numbers (arbitrarily chosen), $X = 010011_2 = 19_{10}$ and $Y = 110001_2 = 49_{10}$. X-Y= 19 - 49 = -30 in decimal. The operation X-Y using 1's complement can be performed as follows:

$$\begin{array}{rr} & X = 010011 \\ \text{Add 1's complement of} & Y = 001110 \\ \hline & 100001 \end{array}$$

Since there is no carry, Result = - (1's Complement of 100001) = $-011110_2 = -30_{10}$. Next consider, $X = 101100_2 = 44_{10}$ and $Y = 011000_2 = 24_{10}$. In decimal, X-Y =

44-24 = 20.

Using 1's complement, X-Y can be obtained as follows:

X = 101100

Add 1's Complement of Y = 100111

Carry →1 010011

Since there is a carry, Result = 010011 + 1 = $+010100_2 = +20_{10}$.

Next, let us describe the procedure of subtracting decimal numbers using addition. This process requires the use of the 10's complement form. The 10's complement of a number can be obtained by subtracting the number from 10.

Consider the decimal subtraction 7 − 4 = 3. The 10's complement of 4 is 10 − 4 = 6. The decimal subtraction can be performed using the 10's complement addition as follows:

minuend 7

10's complement of subtrahend + 6

→ 13

ignore final carry of 1 to obtain the subtraction result of 3.

When a larger number is subtracted from a smaller number, there is no carry to be discarded. Consider the decimal subtraction 4 −7 =−3. The 10's complement of 7 is 10 −7 = 3.

Therefore,

minuend 4

10's complement of subtrahend + 3

→ 7

no final carry

When there is no final carry, the final answer is the negative of the 10's complement of 7. Therefore, the correct result of subtraction is −(10−7) = −3.

The same procedures can be applied for performing binary subtraction. In the case of binary subtraction, the twos complement of the subtrahend is used.

As mentioned before, the twos complement of a binary number is obtained by replacing each 0 with a 1 and each 1 with a 0 and adding 1 to the resulting number. The first step generates a ones complement or simply the complement of a binary number. For example, the ones complement of 10010101 is 01101010. Note that the ones complement of a binary number can be obtained by using inverters; eight inverters are required for generating ones complement of an 8-bit number.

The twos complement of a binary number is formed by adding 1 to the ones complement of the number. For example, the twos complement of 10010101 is found as follows:

binary number	10010101
1's complement	01101010
add 1	+ 1
2's complement	01101011

Now, using the twos complement, binary subtraction can be carried out. Consider the

following subtraction using the normal (pencil and paper) procedure:

minuend	0101	(5)
subtrahend	−0011	(−3)
result	$0010_2 = 2_{10}$	

Using the twos complement subtraction,

minuend	0101
2's complement of subtrahend	1101
	→ 1 0010

discard final carry

The final answer is 0010 (decimal 2).

Consider another example. Using pencil and paper method:

minuend	0101	(5)
subtrahend	− 1001	(−9)
result	− 0100	(−4)

Using the twos complement,

minuend	0101
2's complement of subtrahend	0111
result	→ 1100

no final carry

Therefore, the final answer is −(twos complement of 1100) = −0100, which is −4 in decimal.

Computers typically handle signed numbers by using the most significant bit of a number as the sign bit. If this bit is zero, the number is positive; if this bit is one, the number is negative. Computers use twos complement of the number to represent negative binary numbers and obtain the sign of the result from the most significant bit. However, computers perform ones complement operation on the final carry in order to reflect the true borrow. This is useful for multiprecision subtraction. Also, in the paper and pencil method, the sign of the result of binary subtraction using twos complement can be obtained by utilizing either the most significant bit of the result or the ones complement of the final carry.

For example, the number $+22_{10}$ can be represented using 8 bits as:

$$\overbrace{\underbrace{0}_{\text{sign bit (positive)}}\quad 0010110_2}^{+22_{10}}$$

Hence,

$$-22_{10} = \overbrace{\underbrace{1}_{\text{sign bit (negative)}}\quad 1101010}^{\text{twos complement of } +22_{10}}$$

We now show the procedures for carrying out the addition and subtraction in computers using twos complement arithmetic.

Examples of arithmetic operations of the signed binary numbers are give below. Assume 5 bits to represent each number.

1. Both augend and addend are positive:

0	0101	+5	augend
0	0011	+3	addend
0	1000	+8	

sign bits are all positive

2. Augend is positive, addend is negative:

	0	0101	+5	augend
	1	1101	−3	addend
1	0	0010	+2	

ignore final carry; sign bits

Note that the twos complement of 3 is 11101.

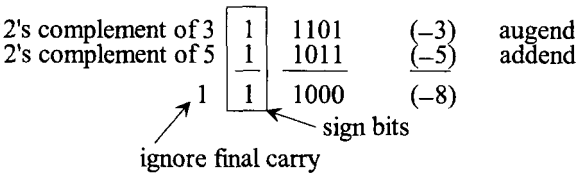
Consider another example:

	0	0011	+3	augend
	1	1011	−5	addend
1	1	1110	−2	

no final carry; sign bits

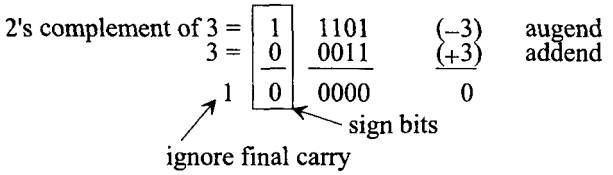
The result is the twos complement of 11110, which is 00010, and therefore, the final answer is -2_{10}.

3. Both augend and addend are negative:



Therefore, the result in binary is 11000. Since the most significant bit is 1, the result is negative. Hence, the result in decimal will be −(twos complement of 11000), which is -8_{10}.

4. The augend and addend are equal with opposite signs:



The final answer is zero.

In all these cases, the sign bit of each of the numbers is conceptually isolated from the number itself. The subtraction operation performed here is similar to twos complement subtraction. For example, when subtracting the subtrahend from the minuend using twos complement, the subtrahend is converted into its twos complement along with the sign bit. If the sign bit of the subtrahend is 1 (for negative subtrahend), its twos complement converts the sign bit from 1 to 0. To perform the subtraction, the twos complement of the subtrahend is added to the minuend. The sign bit of the result indicates whether the answer is positive or negative.

However, an error (indicated by overflow in a microprocessor) may occur while performing twos complement arithmetic. The overflow arises from the representation of the sign flag by the most significant bit of a binary number in signed binary operation. The computer automatically sets an overflow bit to 1 if the result of an arithmetic operation is too big for the computer's maximum word size; otherwise it is reset to 0. To clearly understand the concept of overflow, consider the following examples for 8-bit numbers. Let C_7 be the carry out of the most significant bit (sign bit) and C_6 be the carry out of the previous (bit 6) data bit (seventh bit). We will show by means of numerical examples that as long as C_7 and C_6 are the same, the result is always correct. If, however, C_7 and C_6 are different, the result is incorrect and sets the overflow bit to 1. Now consider the following cases.

Case 1. C_7 and C_6 are the same.

```
  00000110      06₁₆
  00010100    + 14₁₆
0 00011010      1A₁₆
C7 = 0
    ↑
  C6 = 0
```

```
  01101000      68₁₆
  11111010    - 06₁₆
1 01100010      62₁₆
C7 = 1
    ↑
  C6 = 1
```

Therefore when C_7 and C_6 are either both 0 or both 1, a correct answer is obtained.

Case 2. C_7 and C_6 are different.

```
  01011001      59₁₆
  01000101    + 45₁₆
0 10011110    - 62₁₆ ?
C7 = 0
    ↑
  C6 = 1
```

$C_6 = 1$ and $C_7 = 0$ give an incorrect answer because the result shows that the addition of two positive numbers is negative.

```
  10110110      -4A₁₆
  10000001    - 7F₁₆
1 00110111    + 37₁₆ ?
C7 = 1
    ↑
  C6 = 0
```

$C_6 = 0$ and $C_7 = 1$ provide an incorrect answer because the result indicates that the addition of two negative numbers is positive. Hence, the overflow bit will be set to zero if the carries C_7 and C_6 are the same, that is, if both C_7 and C_6 are either 0 or 1. On the other hand, the overflow flag will be set to 1 if the carries C_7 and C_6 are different. The answer is incorrect when the overflow bit is set to 1. Thus,

$$\text{Overflow} = C_7 \oplus C_6.$$

Note that the symbol $\oplus$ represents exclusive-OR logic operation. Exclusive-OR means that when two inputs are the same (both one or both zero), the output is zero. On the other hand, if two inputs are different, the output is one. The overflow can be considered as the output while C_6 and C_7 are the two inputs. The exclusive-OR operation is covered in Chapter 3.

When performing signed arithmetic using pencil and paper, one must consider the overflow bit to ensure that the result is correct. An overflow of one after a signed operation

indicates that the result is too large to be accommodated in the number of bits assigned. One must increase the number of bits for the correct result.

Example 2.3

Perform the following signed operations and comment on the results. Assume twos complement numbers.

(a) $A = 1010_2$, $B = 0100_2$. Find A − B.

(b) Perform $(-3_{10}) - (-2_{10})$ using twos complement and 4 bits.

Solution

(a) The most significant bit of A is 1, so A is a negative number whereas B is a positive number.

$$\begin{array}{rr} A = & 1\ 0\ 1\ 0 \qquad (-6_{10}) \\ \text{Add 2's complement of } B = & +\ 1\ 1\ 0\ 0 \qquad -(+4_{10}) \\ & \overline{0\ 1\ 1\ 0 = 6} \qquad -10_{10} \\ C_3 = 1 \leftarrow & \\ C_2 = 0 & \end{array}$$

Because C_3 and C_2 are different, there is an overflow and the result is incorrect. Four bits are too small to hold the correct answer. If we increase the number of bits for A and B to 5, the correct result can be obtained as follows:

$$A = -6_{10} = 11010_2$$
$$B = +4_{10} = 00100_2$$

$$\begin{array}{rr} A = & 1\ 1\ 0\ 1\ 0_2 \\ \text{Add 2's complement of } B = & +\ 1\ 1\ 1\ 0\ 0_2 \\ & \overline{1\ 0\ 1\ 1\ 0_2} \\ C_4 = 1 \leftarrow & \\ C_3 = 1 & \end{array}$$

The result is correct because C_4 and C_3 are the same. The most significant bit of the result is 1. This means that the result is negative. Therefore, to express the result in base-10, one must take the twos complement and convert the binary number to decimal and place a negative sign in front of it. Thus, twos complement of $10110_2 = -01010_2 = -10_{10}$.

(b)

-3_{10} = 2's complement of $+3_{10}$

$= 1101_2$

-2_{10} = 2's complement of $+2_{10}$

$= 1110_2$

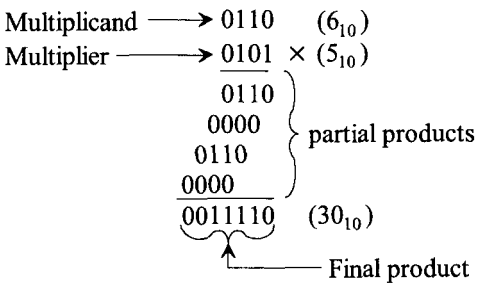
$$\begin{array}{rr} -3_{10} = & 1\ 1\ 0\ 1_2 \qquad (-3_{10}) \\ \text{Add 2's complement of} -2_{10} = & +0\ 0\ 1\ 0_2 \qquad -(-2_{10}) \\ & \overline{1\ 1\ 1\ 1} \qquad -1_{10} \\ C_3 = 0 \leftarrow & \\ C_2 = 0 & \end{array}$$

C_2 and C_3 are the same, so the result is correct. The most significant bit of the

result is 1. This means that the result is negative. To find the result in decimal, one must take the twos complement of the result and place a negative sign in front of it. Twos complement of $1111_2 = -1_{10}$

Multiplication of Unsigned Binary Numbers

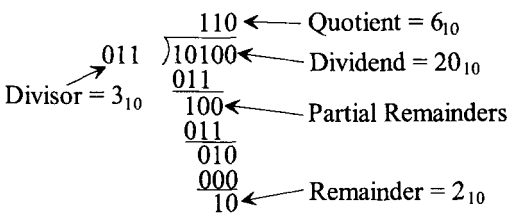
Multiplication of two binary numbers can be carried out in the same way as is done with the decimal numbers using pencil and paper. Consider the following example:

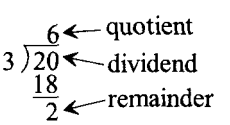


Several multiplication algorithms are available. Multiplication of two unsigned numbers can be accomplished via repeated addition. For example, to multiply 4_{10} by 3_{10}, the number 4_{10} can be added twice to itself to obtain the result, 12_{10}.

Division of Unsigned Binary Numbers

Binary division is carried out in the same way as the division of decimal numbers. As an example, consider the following division:





Division between unsigned numbers can be accomplished via repeated subtraction. For example, consider dividing 7_{10} by 3_{10} as follows:

Dividend	Divisor	Subtraction Result	Counter
7_{10}	3_{10}	7 – 3 = 4	1
		4 – 3 = 1	1 + 1 = 2

Quotient = Counter value = 2
Remainder = subtraction result = 1

Here, one is added to a counter whenever the subtraction result is greater than the

divisor. The result is obtained as soon as the subtraction result is smaller than the divisor.

2.5.2 BCD Arithmetic

Many computers have instructions to perform arithmetic operations using packed BCD numbers. Next, we consider some examples of packed BCD addition and subtraction.

BCD Addition

The two cases that may occur while adding two packed BCD numbers are considered next. Consider adding packed BCD numbers 25 and 33:

25	0010	0101
+33	0011	0011
58	0101	1000

In this example, none of the sums of the pairs of decimal digits exceeded 9; therefore, no decimal carries were produced. For these reasons, the BCD addition process is straightforward and is actually the same as binary addition.

Now consider the addition of 8 and 4 in BCD:

8	0000	1000	
+4	0000	0100	
12	0000	1100	← invalid code group for BCD

The sum 1100 does not exist in BCD code. It is one of the six forbidden or invalid 4-bit code groups. This has occurred because the sum of two digits exceeds 9. Whenever this occurs, the sum has to be corrected by the addition of 6 (0110) to skip over the six invalid code groups. For example,

8	0000	1000	
+4	0000	0100	
12	0000	1100	invalid sum
	+0000	0110	add 6 for correction
	0001	0010	BCD for 12
	1	2	

As another example, add packed BCD numbers 56 and 81:

56	0101	0110	BCD for 56
+81	1000	0001	BCD for 81
137	1101	0111	invalid sum in 2nd digit
	+0110		add 6 for correction
0001	0011	0111	
1	3	7	← correct answer 137

Therefore, it can be concluded that addition of two BCD digits is correct if the binary sum is less than or equal to 1001 (9 in decimal). A binary sum greater than 1001_2 results into an invalid BCD sum; adding 0110_2 to an invalid BCD sum provides the correct sum with an output carry of 1. Furthermore, addition of two BCD digits (each digit having a maximum value of 9) along with carry will require correction if the sum is in the range 16 decimal through 19 decimal. It can be concluded that a correction is necessary for the following:

i) If the binary sum is greater than or equal to decimal 16 (This will generate a carry of one)

ii) If the binary sum is 1010_2 through 1111_2.

For example, consider adding packed BCD numbers 97 and 39:

	111←Intermediate Carries		
97	1001	0111	BCD for 97
+39	0011	1001	BCD for 39
136	1101	0000	invalid sum
	+0110	+0110	add 6 for correction
	0001	0011	0110	
	1	3	6	← correct answer 136

BCD Subtraction

Subtraction of packed BCD numbers can be accomplished in a number of different ways. One method is to add the 10's complement of the subtrahend to the minuend using packed BCD addition rules, as described earlier.

One means of finding the 10's complement of a d-digit packed BCD number N is to take the twos complement of each digit individually, producing a number N_1. Then, ignoring any carries, add the d-digit factor M to N_1, where the least significant digit of M is 1010 and all remaining digits of M are 1001.

As an example, consider subtracting 26_{10} from 84_{10} using BCD subtraction. This can be accomplished as follows:

26_{10}	0010	0110
	2	6

Now, the 10's complement of 26_{10} can be found according to the rules by individually determining the twos complement of 2 and 6, adding the 10's complement factor, and discarding any carries. The twos complement of 2 is 1110, and the twos complement of 6 is 1010. Therefore,

2's complement of each digit of 26_{10}		1110		1010
addition factor to find 10's complement		+1001		1010
10's complement of 26_{10}	(1)	0111	(1)	0100
		7		4

ignore these carries

10's complement of 26_{10}		0111	0100
84_{10}		+1000	0100
		1111	1000
BCD correction factor		+0110	
	(1)	0101	1000
		5	8

ignore carry

Therefore, the final answer is 58_{10}.

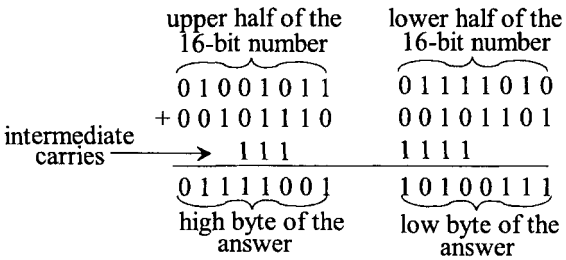
2.5.3 Multiword Binary Addition and Subtraction

In many cases, the word length of a particular microprocessor may not be large enough to represent the desired magnitude of a number. Suppose, for example, that numbers in the range from 0 to 65,535 are to be used in an 8-bit microprocessor in binary addition

and subtraction operations using the twos complement number representation. This can be accomplished by storing the 16-bit numbers each in two 8-bit memory locations. Addition or subtraction of the two 16-bit numbers is implemented by adding or subtracting the lower 8 bits of each number, storing the result in 8-bit memory location or register, and then adding the two high-order parts of the number with any carry or borrow generated from the first addition or subtraction. The latter partial sum or difference will be the high-order portion of the result. Therefore, the two 8-bit operations together comprise the 16-bit result.

Here are some examples of 16-bit addition and subtraction.

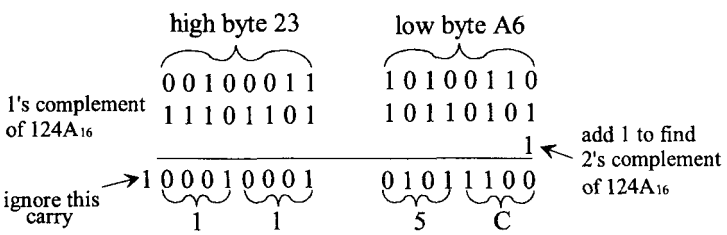
16-Bit Addition



The low-order 8-bit addition can be computed by using the microprocessor's ADD instruction and the high-order 8-bit sum can be obtained by using the ADC (ADD with carry) instruction in the program.

16-Bit Subtraction

Consider $23A6_{16} - 124A_{16} = 115C_{16}$.



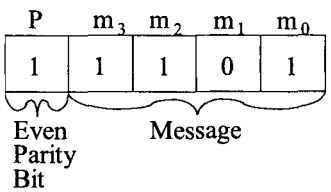
The low-order 8-bit subtraction can be obtained by using SUB instruction of the microprocessor, and the high-order 8-bit subtraction can be obtained by using SBB (SUBTRACT with borrow) instruction in the program.

2.6 Error Correction and Detection

In digital systems, it is possible that the transmitted information is not received correctly. Note that a computer is a digital system in which information transfer can take place in many ways. For example, data may be moved from a CPU register to another device or vice versa. When the transmitted data is not received correctly at the receiving end, an error occurs. One possible cause for such errors is noise problems during transmission. To avoid these problems, error detection and correction may be necessary. In a digital system, an error occurs when a 0 is changed to a 1 and vice versa. Correction of this error means

replacement of a 1 with 0 and vice versa. The reliability of digital data depends on the methods employed for error detection and correction.

The simplest way to detect the presence of an error is by adding a single bit, called the "parity" bit, to the message bits and then transmitting the message along with the parity bit. The parity bit is usually computed in two ways: even parity and odd parity. In the even parity method, the parity bit is added in such a way that after its inclusion, the number of 1's in the message together with the parity bit is an even number. On the other hand, in an odd parity scheme, the parity bit is added in such a way that the number of 1's in the message and the parity bit is an odd number. For example, suppose that the message to be transmitted is 0110. If even parity is used by the transmitting computer, the transmitted data along with the parity bit will be 00110. On the other hand, if odd parity is used, the data to be transmitted will be 10110. The parity computation can be implemented in hardware by using exclusive-OR gates (to be discussed in Chapter 3). Usually for a given message, the parity bit is generated using either an even or odd parity scheme by the transmitting computer. The message is then transmitted along with the parity bit. At the receiving end, the parity is checked by the receiving computer. If there is a discrepancy, the data received will obviously be incorrect. For example, suppose that the message bits are 1101. The even parity bit for this message is 1. The transmitted data will be



Suppose that an error occurs in the least significant bit; that is m0 is changed from 1 to 0 during transmission. The received data will be:

1	1	1	0	0

The receiving computer performs a parity check on this data by counting the number of ones and finds it to be an odd number, three. Therefore, an error is detected.

With a single parity bit, an error due to a single bit change can be detected. Errors due to 2-bit changes during transmission will go undetected. In such situations, multiple parity bits are used. One such technique is the "Hamming code," which uses 3 parity bits for a 4-bit message.

QUESTIONS AND PROBLEMS

2.1 Convert the following unsigned binary numbers into their decimal equivalents:
(a) 01110101_2 (b) 1101.101_2 (c) 1000.111_2

2.2 Convert the following numbers into binary:
(a) 152_{10} (b) 343_{10}

2.3 Convert the following numbers into octal:
(a) 1843_{10} (b) 1766_{10}

2.4 Convert the following numbers into hexadecimal
(a) 1987_{10} (b) 3072_{10}

2.5 Convert the following binary numbers into octal and hexadecimal numbers:
(a) 1101011100101 (b) 11000011100110000011

2.6 Using 8 bits, represent the integers −48 and 52 in
(a) sign magnitude form
(b) ones complement form
(c) twos complement form

2.7 Identify the following unsigned binary numbers as odd or even without converting them to decimal: 11001100_2; 00100100_2; 01111001_2.

2.8 Convert 532.372_{10} into its binary equivalent.

2.9 Convert the following hex numbers to binary: $15FD_{16}$; $26EA_{16}$.

2.10 Provide the BCD bit encodings for the following decimal numbers:
(a) 11264 (b) 8192

2.11 Represent the following numbers in excess-3:
(a) 678 (b) 32874 (c) 61440

2.12 What is the excess-3 equivalent of octal 1543?

2.13 Represent the following binary numbers in BCD:
(a) 0001 1001 0101 0001
(b) 0110 0001 0100 0100 0000

2.14 Express the following binary numbers into excess-3:
(a) 0101 1001 0111
(b) 0110 1001 0000

2.15 Perform the following unsigned binary addition. Include the answer in decimal.

$$\begin{array}{r} 1\,0\,1\,1.0\,1 \\ +\,0\,1\,1\,0.0\,1\,1 \\ \hline \end{array}$$

2.16 Perform the indicated arithmetic operations in binary. Assume that the numbers are in decimal and represented using 8 bits. Express the results in decimal. Use the twos complement approach for carrying out all subtractions.

(a) $\begin{array}{r} 14 \\ +17 \\ \hline \end{array}$ (c) $\begin{array}{r} 32 \\ -14 \\ \hline \end{array}$

(b) $\begin{array}{r} 34 \\ +28 \\ \hline \end{array}$ (d) $\begin{array}{r} 34 \\ -42 \\ \hline \end{array}$

2.17 Using twos complement, perform the following subtraction: $3AFA_{16} - 2F1E_{16}$. Include the answer in hex.

2.18 Using 9's and 10's complement arithmetic, perform the following arithmetic operations:

(a) $254_{10} - 132_{10}$ (b) $783_{10} - 807_{10}$

2.19 Perform the following arithmetic operations in binary using 6 bits. Assume that all numbers are signed decimal. Use twos complement arithmetic. Indicate if there is any overflow.

(a)	14	(b)	7	(c)	27
	+8		+(−7)		+(−19)
(d)	(−24)	(e)	19	(f)	(−17)
	+(−19)		−(−12)		−(−16)

2.20 Perform the following unsigned multiplication in binary using a minimum number of bits required for each decimal number using the pencil and paper method:

$$12 \times 52$$

2.21 Perform the following unsigned division in binary using a minimum number of bits required for each decimal number:

$$3\overline{)14}$$

2.22 Obtain the bit encodings of the following numbers and then perform the indicated arithmetic operations using BCD:

(a)	54	(b)	782	(c)	82
	+48		+219		−58

2.23 Find the odd parity bit for the following binary message to be transmitted: 10110000.

2.24 Repeat Problem 2.20 using repeated addition.

2.25 Repeat Problem 2.21 using repeated subtraction.

2.26 If a transmitting computer sends the 8-bit binary message 11000111 using an even parity bit. Write the 9-bit data with the parity bit in the most significant bit. If the receiving computer receives the 9-bit data as 110000111, is the 8-bit message received correctly? Comment.