1

Number Systems, Number Representations, and Codes

Digital systems contain information that is represented as binary digits called *bits*. The alphabet of these bits is the set {0, 1}, which represents the logical value of the bits. The physical value is determined by the logic family being used. The transistor-transistor logic (TTL) family represents a logic 0 typically as +0.2 volts and a logic 1 typically as +3.4 volts using a +5 volt power supply; the emitter-coupled logic (ECL) 100K family represents a logic 0 typically as –1.7 volts and a logic 1 typically as –0.95 volts using a –4.5 volt power supply.

Thus, a signal can be asserted either positive (plus) or negative (minus), depending upon the active condition of the signal at that point. The word *positive*, as used here, does not necessarily mean a positive voltage level, but merely the more positive of two voltage levels, as is the case for ECL.

1.1 Number Systems

Numerical data are expressed in various positional number systems for each *radix* or *base*. A *positional number system* encodes a vector of n bits in which each bit is weighted according to its position in the vector. The encoded vector is also associated with a radix r, which is an integer greater than or equal to 2. A number system has exactly r digits in which each bit in the radix has a value in the range of 0 to $r - 1$, thus the highest digit value is one less than the radix. For example, the binary radix has two

digits which range from 0 to 1; the octal radix has eight digits which range from 0 to 7. An n-bit integer A is represented in a positional number system as follows:

$$A = (a_{n-1}a_{n-2}a_{n-3} \ \ldots \ a_1a_0) \tag{1.1}$$

where $0 \le a_i \le r - 1$. The high-order and low-order digits are a_{n-1} and a_0, respectively. The number in Equation 1.1 (also referred to as a vector or operand) can represent positive integer values in the range 0 to $r^n - 1$. Thus, a positive integer A is written as

$$A = a_{n-1}r^{n-1} + a_{n-2}r^{n-2} + a_{n-3}r^{n-3} + \ldots + a_1r^1 + a_0r^0 \tag{1.2}$$

The value for A can be represented more compactly as

$$A = \sum_{i=0}^{n-1} a_i r^i \tag{1.3}$$

The expression of Equation 1.2 can be extended to include fractions. For example,

$$A = a_{n-1}r^{n-1} + \ldots + a_1r^1 + a_0r^0 + a_{-1}r^{-1} + a_{-2}r^{-2} + \ldots + a_{-m}r^{-m} \tag{1.4}$$

Equation 1.4 can be represented as

$$A = \sum_{i=-m}^{n-1} a_i r^i \tag{1.5}$$

Adding 1 to the highest digit in a radix r number system produces a sum of 0 and a carry of 1 to the next higher-order column. Thus, counting in radix r produces the following sequence of numbers:

$$0, 1, 2, \ldots, (r-1), 10, 11, 12, \ldots, 1(r-1), \ldots.$$

Table 1.1 shows the counting sequence for different radices. The low-order digit will always be 0 in the set of r digits for the given radix. The set of r digits for various

radices is given in Table 1.2. In order to maintain one character per digit, the numbers 10, 11, 12, 13, 14, and 15 are represented by the letters A, B, C, D, E, and F, respectively.

Table 1.1 Counting Sequence for Different Radices

Decimal	$r = 2$	$r = 4$	$r = 8$
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	10	4
5	101	11	5
6	110	12	6
7	111	13	7
8	1000	20	10
9	1001	21	11
10	1010	22	12
11	1011	23	13
12	1100	30	14
13	1101	31	15
14	1110	32	16
15	1111	33	17
16	10000	100	20
17	10001	101	21

Table 1.2 Character Sets for Different Radices

Radix (base)	Character Sets for Different Radices
2	{0, 1}
3	{0, 1, 2}
4	{0, 1, 2, 3}
5	{0, 1, 2, 3, 4}
6	{0, 1, 2, 3, 4, 5}
7	{0, 1, 2, 3, 4, 5, 6}
8	{0, 1, 2, 3, 4, 5, 6, 7}
9	{0, 1, 2, 3, 4, 5, 6, 7, 8}
10	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

(Continued on next page)

Table 1.2 Character Sets for Different Radices

Radix (base)	Character Sets for Different Radices
11	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A}
12	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B}
13	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C}
14	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D}
15	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E}
16	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Example 1.1 Count from decimal 0 to 25 in radix 5. Table 1.2 indicates that radix 5 contains the following set of four digits: {0, 1, 2, 3, 4}. The counting sequence in radix 5 is:

$$\begin{aligned}
000, 001, 002, 003, 004 &= (0 \times 5^2) + (0 \times 5^1) + (4 \times 5^0) = 4_{10} \\
010, 011, 012, 013, 014 &= (0 \times 5^2) + (1 \times 5^1) + (4 \times 5^0) = 9_{10} \\
020, 021, 022, 023, 024 &= (0 \times 5^2) + (2 \times 5^1) + (4 \times 5^0) = 14_{10} \\
030, 031, 032, 033, 034 &= (0 \times 5^2) + (3 \times 5^1) + (4 \times 5^0) = 19_{10} \\
040, 041, 042, 043, 044 &= (0 \times 5^2) + (4 \times 5^1) + (4 \times 5^0) = 24_{10} \\
100 &= (1 \times 5^2) + (0 \times 5^1) + (0 \times 5^0) = 25_{10}
\end{aligned}$$

Example 1.2 Count from decimal 0 to 25 in radix 12. Table 1.2 indicates that radix 12 contains the following set of twelve digits: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B}. The counting sequence in radix 12 is:

$$\begin{aligned}
00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B &= (0 \times 12^1) + (11 \times 12^0) = 11_{10} \\
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B &= (1 \times 12^1) + (11 \times 12^0) = 23_{10} \\
20, 21 &= (2 \times 12^1) + (1 \times 12^0) = 25_{10}
\end{aligned}$$

1.1.1 Binary Number System

The radix is 2 in the *binary number system*; therefore, only two digits are used: 0 and 1. The low-value digit is 0 and the high-value digit is $(r - 1) = 1$. The binary number system is the most conventional and easily implemented system for internal use in a digital computer; therefore, most digital computers use the binary number system. There is a disadvantage when converting to and from the externally used decimal system; however, this is compensated for by the ease of implementation and the speed of execution in binary of the four basic operations: addition, subtraction, multiplication, and division. The radix point is implied within the internal structure of the computer; that is, there is no specific storage element assigned to contain the radix point.

The weight assigned to each position of a binary number is as follows:

$$2^{n-1}2^{n-2} \ldots 2^3\, 2^2\, 2^1\, 2^0 . 2^{-1}2^{-2}2^{-3} \ldots 2^{-m}$$

where the integer and fraction are separated by the radix point (binary point). The decimal value of the binary number 1011.101_2 is obtained by using Equation 1.4, where $r = 2$ and $a_i \in \{0,1\}$ for $-m \le i \le n-1$. Therefore,

$$\begin{array}{cccccccc} 2^3 & 2^2 & 2^1 & 2^0 & . & 2^{-1} & 2^{-2} & 2^{-3} \\ 1 & 0 & 1 & 1 & . & 1 & 0 & 1_2 \end{array} = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3})$$
$$= 11.625_{10}$$

Digital systems are designed using bistable storage devices that are either reset (logic 0) or set (logic 1). Therefore, the binary number system is ideally suited to represent numbers or states in a digital system, since radix 2 consists of the alphabet 0 and 1. These bistable devices can be concatenated to any length n to store binary data. For example, to store 1 byte (8 bits) of data, eight bistable storage devices are required as shown in Figure 1.1 for the value 0110 1011 (107_{10}). Counting in binary is shown in Table 1.3, which shows the weight associated with each of the four binary positions. Notice the alternating groups of 1s in Table 1.3. A binary number is a group of n bits that can assume 2^n different combinations of the n bits. The range for n bits is 0 to $2^n - 1$.

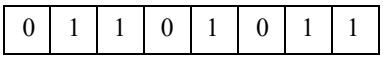


Figure 1.1 Concatenated 8-bit storage elements.

Table 1.3 Counting in Binary

Decimal	Binary			
	8	4	2	1
	2^3	2^2	2^1	2^0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0

(Continued on next page)

Table 1.3 Counting in Binary

Decimal	Binary			
	8	4	2	1
	2^3	2^2	2^1	2^0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

The binary weights for the bit positions of an 8-bit integer are shown in Table 1.4; the binary weights for an 8-bit fraction are shown in Table 1.5.

Table 1.4 Binary Weights for an 8-Bit Integer

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

Table 1.5 Binary Weights for an 8-Bit Fraction

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

Each 4-bit binary segment has a weight associated with the segment and is assigned the value represented by the low-order bit of the corresponding segment, as shown in the first row of Table 1.6. The 4-bit binary number in each segment is then multiplied by the value of the segment. Thus, the binary number 0010 1010 0111 1100 0111 is equal to the decimal number $59{,}335_{10}$ as shown below.

$$(2 \times 8192) + (10 \times 4096) + (7 \times 256) + (12 \times 16) + (7 \times 1) = 59{,}335_{10}$$

Table 1.6 Weight Associated with 4-Bit Binary Segments

8192	4096	256	16	1
0001	0001	0001	0001	0001
0010	1010	0111	1100	0111

1.1.2 Octal Number System

The radix is 8 in the *octal number system*; therefore, eight digits are used, 0 through 7. The low-value digit is 0 and the high-value digit is $(r - 1) = 7$. The weight assigned to each position of an octal number is as follows:

$$8^{n-1}\,8^{n-2}\,\ldots\,8^3\,8^2\,8^1\,8^0\,.\,8^{-1}\,8^{-2}\,8^{-3}\,\ldots\,8^{-m}$$

where the integer and fraction are separated by the radix point (octal point). The decimal value of the octal number 217.6_8 is obtained by using Equation 1.4, where $r = 8$ and $a_i \in \{0,1,2,3,4,5,6,7\}$ for $-m \le i \le n - 1$. Therefore,

$$\begin{array}{cccccl} 8^2 & 8^1 & 8^0 & . & 8^{-1} & \\ 2 & 1 & 7 & . & 6_8 & = (2 \times 8^2) + (1 \times 8^1) + (7 \times 8^0) + (6 \times 8^{-1}) \\ & & & & & = 143.75_{10} \end{array}$$

When a count of 1 is added to 7_8, the sum is zero and a carry of 1 is added to the next higher-order column on the left. Counting in octal is shown in Table 1.7, which shows the weight associated with each of the three octal positions.

Table 1.7 Counting in Octal

Decimal	Octal		
	64	8	1
	8^2	8^1	8^0
0	0	0	0
1	0	0	1
2	0	0	2
3	0	0	3
4	0	0	4
5	0	0	5
6	0	0	6
7	0	0	7
8	0	1	0
9	0	1	1
…		…	
14	0	1	6
15	0	1	7
16	0	2	0
17	0	2	1
…		…	

Continued on next page

Table 1.7 Counting in Octal

Decimal	Octal		
	64	8	1
	8^2	8^1	8^0
22	0	2	6
23	0	2	7
24	0	3	0
25	0	3	1
…		…	
30	0	3	6
31	0	3	7
…		…	
84	1	2	4
…		…	
242	3	6	2
…		…	
377	5	7	1

Binary-coded octal Each octal digit can be encoded into a corresponding binary number. The highest-valued octal digit is 7; therefore, three binary digits are required to represent each octal digit. This is shown in Table 1.8, which lists the eight decimal digits (0 through 7) and indicates the corresponding octal and binary-coded octal (BCO) digits. Table 1.8 also shows octal numbers of more than one digit.

Table 1.8 Binary-Coded Octal Numbers

Decimal	Octal	Binary-Coded Octal
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
8	10	001 000
9	11	001 001
10	12	001 010

(Continued on next page)

Table 1.8 Binary-Coded Octal Numbers

Decimal	Octal	Binary-Coded Octal		
11	13		001	011
…	…		…	
20	24		010	100
21	25		010	101
…	…		…	
100	144	001	100	100
101	145	001	100	101
…	…		…	
267	413	100	001	011
…	…		…	
385	601	110	000	001

1.1.3 Decimal Number System

The radix is 10 in the *decimal number system*; therefore, ten digits are used, 0 through 9. The low-value digit is 0 and the high-value digit is $(r - 1) = 9$. The weight assigned to each position of a decimal number is as follows:

$$10^{n-1}10^{n-2} \ldots \; 10^3 \; 10^2 \; 10^1 \; 10^0 . \; 10^{-1}10^{-2}10^{-3} \; \ldots \; 10^{-m}$$

where the integer and fraction are separated by the radix point (decimal point). The value of 6357_{10} is immediately apparent; however, the value is also obtained by using Equation 1.4, where $r = 10$ and $a_i \in \{0,1,2,3,4,5,6,7,8,9\}$ for $-m \le i \le n - 1$. That is,

$$\begin{array}{cccc} 10^3 & 10^2 & 10^1 & 10^0 \\ 6 & 3 & 5 & 7_{10} \end{array} = (6 \times 10^3) + (3 \times 10^2) + (5 \times 10^1) + (7 \times 10^0)$$

When a count of 1 is added to decimal 9, the sum is zero and a carry of 1 is added to the next higher-order column on the left. The following example contains both an integer and a fraction:

$$\begin{array}{ccccccc} 10^3 & 10^2 & 10^1 & 10^0 & . & 10^{-1} & \\ 5 & 4 & 3 & 6 & . & 5 & = \end{array} (5 \times 10^3) + (4 \times 10^2) + (3 \times 10^1) + (6 \times 10^0) + (5 \times 10^{-1})$$

Binary-coded decimal Each decimal digit can be encoded into a corresponding binary number; however, only ten decimal digits are valid. The highest-valued decimal digit is 9, which requires four bits in the binary representation. Therefore, four binary digits are required to represent each decimal digit. This is shown in Table 1.9,

which lists the ten decimal digits (0 through 9) and indicates the corresponding binary-coded decimal (BCD) digits. Table 1.9 also shows BCD numbers of more than one decimal digit.

Table 1.9 Binary-Coded Decimal Numbers

Decimal	Binary-Coded Decimal
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0001 0000
11	0001 0001
12	0001 0010
…	…
124	0001 0010 0100
…	…
365	0011 0110 0101

1.1.4 Hexadecimal Number System

The radix is 16 in the *hexadecimal number system*; therefore, 16 digits are used, 0 through 9 and A through F, where by convention A, B, C, D, E, and F correspond to decimal 10, 11, 12, 13, 14, and 15, respectively. The low-value digit is 0 and the high-value digit is $(r - 1) = 15$ (F). The weight assigned to each position of a hexadecimal number is as follows:

$$16^{n-1}16^{n-2} \ldots \; 16^3 \; 16^2 \; 16^1 \; 16^0 . \; 16^{-1}16^{-2}16^{-3} \; \ldots \; 16^{-m}$$

where the integer and fraction are separated by the radix point (hexadecimal point). The decimal value of the hexadecimal number $6A8C.D416_{16}$ is obtained by using Equation 1.4, where $r = 16$ and $a_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ for $-m \leq i \leq n - 1$. Therefore,

$$
\begin{array}{cccccccccccl}
16^3 & 16^2 & 16^1 & 16^0 & . & 16^{-1} & 16^{-2} & 16^{-3} & 16^{-4} & & & \\
6 & A & 8 & C & . & D & 4 & 1 & 6 & = & (6 \times 16^3) + (10 \times 16^2) + (8 \times 16^1) \\
 & & & & & & & & & & + (12 \times 16^0) + (13 \times 16^{-1}) + (4 \times 16^{-2}) \\
 & & & & & & & & & & + (1 \times 16^{-3}) + (6 \times 16^{-4}) \\
 & & & & & & & & & = & 27{,}276.82846_{10}
\end{array}
$$

When a count of 1 is added to hexadecimal F, the sum is zero and a carry of 1 is added to the next higher-order column on the left.

Binary-coded hexadecimal Each hexadecimal digit corresponds to a 4-bit binary number as shown in Table 1.10. All 2^4 values of the four binary bits are used to represent the 16 hexadecimal digits. Table 1.10 also indicates hexadecimal numbers of more than one digit. Counting in hexadecimal is shown in Table 1.11. Table 1.12 summarizes the characters used in the four number systems: binary, octal, decimal, and hexadecimal.

Table 1.10 Binary-Coded Hexadecimal Numbers

Decimal	Hexadecimal	Binary-Coded Hexadecimal
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
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
…	…	…
124	7C	0111 1100
…	…	…
365	16D	0001 0110 1101

Table 1.11 Counting in Hexadecimal

Decimal	Hexadecimal		
	256	16	1
	16^2	16^1	16^0
0	0	0	0
1	0	0	1
2	0	0	2
3	0	0	3
4	0	0	4
5	0	0	5
6	0	0	6
7	0	0	7
8	0	0	8
9	0	0	9
10	0	0	A
11	0	0	B
12	0	0	C
13	0	0	D
14	0	0	E
15	0	0	F
16	0	1	0
17	0	1	1
…		…	
26	0	1	A
27	0	1	B
…		…	
30	0	1	E
31	0	1	F
…		…	
256	1	0	0
…		…	
285	1	1	D
…		…	
1214	4	B	E

1.1.5 Arithmetic Operations

The arithmetic operations of addition, subtraction, multiplication, and division in any radix can be performed using identical procedures to those used for decimal arithmetic. The operands for the four operations are shown in Table 1.13.

Table 1.12 Digits Used for Binary, Octal, Decimal, and Hexadecimal Number Systems

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary															
Octal															
Decimal															
Hexadecimal															

Table 1.13 Operands Used for Arithmetic Operations

	Addition		Subtraction		Multiplication		Division
	Augend		Minuend		Multiplicand		Dividend
+)	Addend	–)	Subtrahend	×)	Multiplier	÷)	Divisor
	Sum		Difference		Product		Quotient, Remainder

Radix 2 addition Figure 1.2 illustrates binary addition of unsigned operands. The sum of column 1 is 2_{10} (10_2); therefore, the sum is 0 with a carry of 1 to column 2. The sum of column 2 is 4_{10} (100_2); therefore, the sum is 0 with a carry of 0 to column 3 and a carry of 1 to column 4. The sum of column 3 is 3_{10} (11_2); therefore, the sum is 1 with a carry of 1 to column 4. The sum of column 4 is 4_{10} (100_2); therefore, the sum is 0 with a carry of 0 to column 5 and a carry of 1 to column 6.

The unsigned values of the binary operands are shown in the rightmost column together with the resulting sum.

Column	6	5	4	3	2	1	Radix 10 values
			1	1	1	0	14
			0	1	1	1	7
			1	0	1	0	10
+)			0_{11}	1_0	0_1	1	5
		0	0	1	0	0	36

Figure 1.2 Example of binary addition.

Radix 2 subtraction The rules for subtraction in radix 2 are as follows:

0 – 0 =	0
0 – 1 =	1 with a borrow from the next higher-order minuend
1 – 0 =	1
1 – 1 =	0

Figure 1.3 provides an example of binary subtraction using the above rules for unsigned operands. An alternative method for subtraction — used in computers — will be given in Section 1.2 when number representations are presented. In Figure 1.3 column 3, the difference is 1 with a borrow from the minuend in column 4, which changes the minuend in column 4 to 0.

Column	4	3	2	1	Radix 10 values
	1^0	0	1	1	11
–)	0	1	0	1	5
		1	1	0	6

Figure 1.3 Example of binary subtraction.

Radix 2 multiplication Multiplying in binary is similar to multiplying in decimal. Two n-bit operands produce a $2n$-bit product. Figure 1.4 shows an example of binary multiplication using unsigned operands, where the multiplicand is 7_{10} and the multiplier is 14_{10}. The multiplicand is multiplied by the low-order multiplier bit (0) producing a partial product of all zeroes. Then the multiplicand is multiplied by the next higher-order multiplier bit (1) producing a left-shifted partial product of 0000 111. The process repeats until all bits of the multiplier have been used.

				0	1	1	1	7
			×)	1	1	1	0	14
0	0	0	0	0	0	0	0	
0	0	0	0	1	1	1		
0	0	0	1	1	1			
0	0_1	1_{11}	1_0	1_1				
	1	1	0	0	0	1	0	98

Figure 1.4 Example of binary multiplication.

Radix 2 division The division process is shown in Figure 1.5, where the divisor is n bits and the dividend is $2n$ bits. The division procedure uses a sequential shift-subtract-restore technique. Figure 1.5 shows a divisor of 5_{10} (0101_2) and a dividend of 13_{10} ($0000\ 1101_2$), resulting in a quotient of 2_{10} (0010_2) and a remainder of 3_2 (0011_2).

The divisor is subtracted from the high-order four bits of the dividend. The result is a partial remainder that is negative — the leftmost bit is 1 — indicating that the divisor is greater than the four high-order bits of the dividend. Therefore, a 0 is placed

in the high-order bit position of the quotient. The dividend bits are then restored to their previous values with the next lower-order bit (1) of the dividend being appended to the right of the partial product. The divisor is shifted right one bit position and again subtracted from the dividend bits.

								0	0	0	1	0	Quotient
	0	1	0	1	0	0	0	0	1	1	0	1	
Subtract					0	1	0	1					
					1	0	1	1					
Restore					0	0	0	0	1				
Shift-subtract						0	1	0	1				
						1	1	0	0				
Restore					0	0	0	0	1	1			
Shift-subtract							0	1	0	1			
							1	1	1	0			
Restore					0	0	0	0	1	1	0		
Shift-subtract								0	1	0	1		
								0	0	0	1		
No restore					0	0	0	0	0	0	1	1	
Shift-subtract									0	1	0	1	
									1	1	1	0	
Restore					0	0	0	0	0	0	1	1	Remainder

Figure 1.5 Example of binary division.

This restore-shift-subtract cycle repeats for a total of three cycles until the partial remainder is positive — the leftmost bit is 0, indicating that the divisor is less than the corresponding dividend bits. This results in a no-restore cycle in which the previous partial remainder (0001) is not restored. A 1 bit is placed in the next lower-order quotient bit and the next lower-order dividend bit is appended to the right of the partial remainder. The divisor is again subtracted, resulting in a negative partial remainder, which is again restored by adding the divisor. The 4-bit quotient is 0010 and the 4-bit remainder is 0011.

The results can be verified by multiplying the quotient (0010) by the divisor (0101) and adding the remainder (0011) to obtain the dividend. Thus, $0010 \times 0101 = 1010 + 0011 = 1101$.

Radix 8 addition Figure 1.6 illustrates octal addition. The result of adding column 1 is 17_8, which is a sum of 1 with a carry of 2. The result of adding column 2 is 11_8, which is a sum of 3 with a carry of 1. The remaining columns are added in a similar manner, yielding a result of 21631_8 or 9113_{10}.

Column	4	3	2	1	Radix 10 value
	7	6	5	4	4012
	6	5	4	7	3431
+)	3_1	2_1	0_2	6	1670
	1	6	3	1	9113

Figure 1.6 Example of octal addition.

Radix 8 subtraction Octal subtraction is slightly more complex than octal addition. Figure 1.7 provides an example of octal subtraction. In column 2 (8^1), a 1 is subtracted from minuend 5_8 leaving a value of 4_8; the 1 is then added to the minuend in column 1 (2_8). This results in a difference of 6_8 in column 1, as shown below.

$(1 \times 8^1) + (2 \times 8^0) = 10_{10}$
Therefore, $10 - 4 = 6$

In a similar manner, in column 4 (8^3), a 1 is subtracted from minuend 6_8 leaving a value of 5_8; the 1 is then added to the minuend in column 3 (1_8), leaving a difference of $9 - 5 = 4$, as shown below.

$(1 \times 8^3) + (1 \times 8^2) = 1100_8$
Consider only the 11 of 1100_8, where $(1 \times 8^1) + (1 \times 8^0) = 9_{10}$
Therefore, $9 - 5 = 4$

	8^3	8^2	8^1	8^0
Column	4	3	2	1
	6	1	5	2
–)	5	5	3	4
		4	1	6

Figure 1.7 Example of octal subtraction.

Consider another example of octal subtraction shown in Figure 1.8, which shows a slightly different approach. A 1 is subtracted from the minuend in column 4 and

added to the minuend in column 3. This results in a value of 13_8 in column 3 or 001 011 in binary-coded octal (also radix 2). Therefore, 001 011 – 101 = 000 110 = 6.

	8^3	8^2	8^1	8^0
Column	4	3	2	1
	6	3	7	2
–)	4	5	0	1
		6	7	1

Figure 1.8 Example of octal subtraction.

Radix 8 multiplication An example of octal multiplication is shown in Figure 1.9. The multiplicand is multiplied by each multiplier digit in turn to obtain a partial product. Except for the first partial product, each successive partial product is shifted left one digit. The subscripts in partial products 3 and 4 represent carries obtained from multiplying the multiplicand by the multiplier digits. When all of the partial products are obtained, the partial products are added following the rules for octal addition.

					7	4	6	3
				×)	5	2	1	0
Partial product 1	0	0	0	0	0	0	0	0
Partial product 2	0	0	0	7	4	6	3	
Partial product 3	0	0_1	6_1	0_1	4	6		
Partial product 4	0_4	3_2	4_3	6_1	7			
Carries from addition	1	2	2	2	1			
		0	0	1	0	4	3	0

Figure 1.9 Example of octal multiplication.

Radix 8 division An example of octal division is shown in Figure 1.10. The first quotient digit is 3_8 which, when multiplied by the divisor 17_8, yields a result of 55_8. This can be verified as shown Figure 1.11, where $3_8 \times 7_8 = 25_8$, resulting in a product of 5_8 and a carry of 2_8. Another approach is as follows: $3_{10} \times 7_{10} = 21_{10}$, which is 5 away from 2×16; that is, a product of 5_8 and a carry of 2_8. Subtraction of the partial remainder and multiplication of the quotient digit times the divisor are accomplished using the rules stated above for octal arithmetic.

				2	3
1	7	6	1	4	5
		5	5		
			4	4	
			3	6	
				6	5
				5	5
					0

Figure 1.10 Example of octal division.

	1	7
×)		3
	$3_2 = 5$	5

Figure 1.11 Example of octal multiplication for the first partial remainder of Figure 1.10.

The results of Figure 1.10 can be verified as follows:

Dividend = (quotient × divisor) + remainder
= $(323_8 \times 17_8) + 10_8$
= 6145_8

Radix 10 addition Arithmetic operations in the decimal number system are widely used and need no introduction; however, they are included here to add completeness to the number system topic. An example of decimal addition is shown in Figure 1.12. The carries between columns are indicated by subscripted numbers.

	4	5	2
	7	6	5
	1	8	9
+)	2_3	9_2	7
	7	0	3

Figure 1.12 Example of decimal addition.

Radix 10 subtraction An example of decimal subtraction is shown in Figure 1.13. The superscripted numbers indicate the minuend result after a borrow is subtracted.

	7^{6}	6^{5}	3	9	4
–)	5	9	7	2	2
		6	6	7	2

Figure 1.13 Example of decimal subtraction.

Radix 10 multiplication An example of decimal multiplication is shown in Figure 1.14. The subscripted numbers indicate the carries.

			2	9	6
		×)	5	4	3
0	0	0	8	8	8
0	1	1	8	4	
1	4_1	8_1	0_1		
	6	0	7	2	8

Figure 1.14 Example of decimal multiplication.

Radix 10 division An example of decimal division is shown in Figure 1.15.

					8
7	2	1	3	4	9
			7	2	
			6	2	9
			5	7	6
					3

Figure 1.15 Example of decimal division.

Radix 16 addition An example of hexadecimal addition is shown in Figure 1.16. The subscripted numbers indicate carries from the previous column. The decimal value of the hexadecimal addition of each column is also shown. To obtain the hexadecimal value of the column, a multiple of 16_{10} is subtracted from decimal value and the difference is the hexadecimal value and the multiple of 16_{10} is the carry. For exam-

ple, the decimal sum of column 1 is 28. Therefore, 28 – 16 = 12 (C_{16}) with a carry of 1 to column 2. In a similar manner, the decimal sum of column 2 is 40 + 1 (carry) = 41. Therefore, 41 – 32 = 9 (9_{16}) with a carry of 2 to column 3.

Column	4	3	2	1
	A	B	C	D
	9	8	7	6
	E	F	9	4
+)	9_2	A_2	C_1	5
Radix 10 =	44	46	41	28
	C	E	9	C

Figure 1.16 Example of hexadecimal addition.

Radix 16 subtraction Hexadecimal subtraction is similar to subtraction in any other radix. An example of hexadecimal subtraction is shown in Figure 1.17.

Column	4	3	2	1
	C^1	2^1	8	D
–)	8	F	E	9
		2	A	4

Figure 1.17 Example of hexadecimal subtraction.

The superscripted numbers indicate borrows from the minuends. For example, the minuend in column 2 borrows a 1 from the minuend in column 3; therefore, column 2 becomes $18_{16} – E_{16} = A_{16}$. This is more readily apparent if the hexadecimal numbers are represented as binary numbers, as shown below.

	1	8			0	0	0	1		1	0	0	0
–)		E	→	–)	0	0	0	0		1	1	1	0
					0	0	0	0		1	0	1	0

In a similar manner, column 3 becomes $11_{16} – F_{16} = 2_{16}$ with a borrow from column 4. Column 4 becomes $B_{16} – 8_{16} = 3_{16}$.

Radix 16 multiplication Figure 1.18 shows an example of hexadecimal multiplication. Multiplication in radix 16 is slightly more complex than multiplication in other radices. Each multiplicand is multiplied by multiplier digit in turn to form a partial product. Except for the first partial product, each partial product is shifted left one digit position. The subscripted digits in Figure 1.18 indicate the carries formed when multiplying the multiplicand by the multiplier digits.

Consider the first row of Figure 1.18 — the row above partial product 1.

$10_{10} \times 4_{10} = 40_{10} = 8_{16}$ with a carry of 2_{16}
$10_{10} \times 13_{10} = 130_{10} = 2_{16}$ with a carry of 8_{16}
$10_{10} \times 9_{10} = 90_{10} = A_{16}$ with a carry of 5_{16}
$10_{10} \times 12_{10} = 120_{10} = 8_{16}$ with a carry of 7_{16}

In a similar manner, the remaining partial products are obtained. Each column of partial products is then added to obtain the product.

					C	9	D	4
				×)	7	8	B	A
				7	8_{51}	A_8	2_2	8
Partial product 1	0	0	0	7	E	2	4	8
			8	4_6	3_8	F_2	C	
Partial product 2	0	0	8	A	C	1	C	
	0	6	4	8_6	8_2	0		
Partial product 3	0	6	4	E	A	0		
	5	4_3	F_5	B_1	C			
Partial product 4	5	8	4	C	C			
Carries from addition		1	2	3		1		
Product	5	F	2	E	0	4	0	8

Figure 1.18 Example of hexadecimal multiplication.

Radix 16 division Figure 1.19 (a) and Figure 1.19 (b) show two examples of hexadecimal division. The results of Figure 1.19 can be verified as follows:

Dividend = (quotient × divisor) + remainder

For Figure 1.19 (a): Dividend = $(F0F_{16} \times 11_{16}) + 0 = FFFF_{16}$
For Figure 1.19 (b): Dividend = $(787_{16} \times 22_{16}) + 11_{16} = FFFF_{16}$

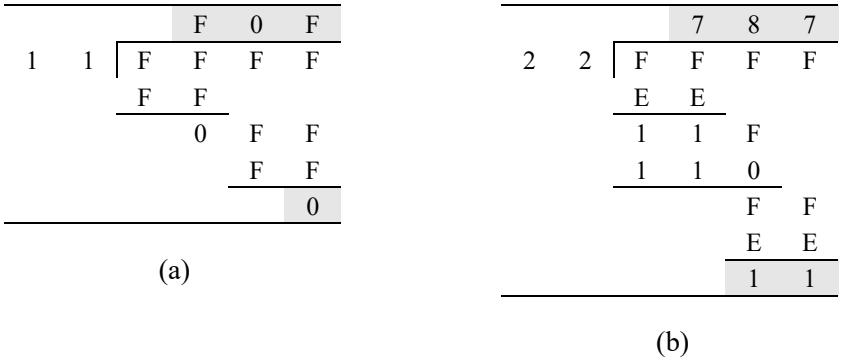


Figure 1.19 Examples of hexadecimal division.

1.1.6 Conversion Between Radices

Methods to convert a number in radix r_i to radix r_j will be presented in this section. The following conversion methods will be presented:

Binary	→	Decimal
Octal	→	Decimal
Hexadecimal	→	Decimal
Decimal	→	Binary
Decimal	→	Octal
Decimal	→	Hexadecimal
Binary	→	Octal
Binary	→	Hexadecimal
Octal	→	Binary
Octal	→	Hexadecimal
Hexadecimal	→	Binary
Hexadecimal	→	Octal
Octal	→	Binary-coded octal
Hexadecimal	→	Binary-coded hexadecimal
Decimal	→	Binary-coded decimal

Comparison between the following formats will also be examined:

octal → binary-coded octal and octal → binary
hexadecimal → binary-coded hexadecimal and hexadecimal → binary
decimal → binary-coded decimal and decimal → binary

There will also be an example to illustrate converting between two nonstandard radices and an example to determine the value of an unknown radix for a given radix 10 number.

Binary to decimal Conversion from any radix r to radix 10 is easily accomplished by using Equation 1.2 or 1.3 for integers, or Equation 1.4 or 1.5 for numbers consisting of integers and fractions. The binary number 1111000.101_2 will be converted to an equivalent decimal number. The weight by position is as follows:

2^6	2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
1	1	1	1	0	0	0	.	1	0	1

$$\begin{aligned}\text{Therefore, } 1111000.101_2 = {} & (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + \\ & (0 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) + \\ & (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) \\ = {} & 64 + 32 + 16 + 8 + 0.5 + 0.125 \\ = {} & 120.625_{10}\end{aligned}$$

Octal to decimal The octal number 217.65_8 will be converted to an equivalent decimal number. The weight by position is as follows:

8^2	8^1	8^0		8^{-1}	8^{-2}
2	1	7	.	6	5

$$\begin{aligned}\text{Therefore, } 217.65_8 = {} & (2 \times 8^2) + (1 \times 8^1) + (7 \times 8^0) + (6 \times 8^{-1})\,(5 \times 8^{-2}) \\ = {} & 128 + 8 + 7 + 0.75 + 0.078125 \\ = {} & 143.828125_{10}\end{aligned}$$

Hexadecimal to decimal The hexadecimal number $5C2.4D_{16}$ will be converted to an equivalent decimal number. The weight by position is as follows:

16^2	16^1	16^0		16^{-1}	16^{-2}
5	C	2	.	4	D

$$\begin{aligned}\text{Therefore, } 5C2.4D_{16} = {} & (5 \times 16^2) + (12 \times 16^1) + (2 \times 16^0) + (4 \times 16^{-1}) + (13 \times 16^{-2}) \\ = {} & 1280 + 192 + 2 + 0.25 + 0.05078125 \\ = {} & 1474.300781_{10}\end{aligned}$$

Decimal to binary To convert a number in radix 10 to any other radix *r*, repeatedly divide the integer by radix *r*, then repeatedly multiply the fraction by radix *r*. The first remainder obtained when dividing the integer is the low-order digit. The first integer obtained when multiplying the fraction is the high-order digit.

The decimal number 186.625_{10} will be converted to an equivalent binary number. The process is partitioned into two parts: divide the integer 186_{10} repeatedly by 2 until the quotient equals zero; multiply the fraction 0.625 repeatedly by 2 until a zero result is obtained or until a certain precision is reached.

186 ÷ 2 =	quotient = 93,	remainder = 0	(0 is the low-order digit)
93 ÷ 2 =	quotient = 46,	remainder = 1	
46 ÷ 2 =	quotient = 23,	remainder = 0	
23 ÷ 2 =	quotient = 11,	remainder = 1	
11 ÷ 2 =	quotient = 5,	remainder = 1	
5 ÷ 2 =	quotient = 2,	remainder = 1	
2 ÷ 2 =	quotient = 1,	remainder = 0	
1 ÷ 2 =	quotient = 0,	remainder = 1	

0.625 × 2 =	1.25	1	(1 is the high-order digit)
0.25 × 2 =	0.5	0	
0.5 × 2 =	1.0	1	

Therefore, $186.625_{10} = 10111010.101_2$.

The decimal number 267_{10} will be converted to binary-coded decimal (BCD) and binary. The binary bit configuration for BCD is

$$267_{10} = 0010\ \ 0110\ \ 0111_{BCD}$$

The bit configuration for binary is

$$267_{10} = 0001\ \ 0000\ \ 1011_2$$

The two results are not equal because the BCD number system does not use all sixteen combinations of four bits. BCD uses only ten combinations — 0 through 9.

Decimal to octal The decimal number 219.62_{10} will be converted to an equivalent octal number. The integer 219_{10} is divided by 8 repeatedly and the fraction 0.62_{10} is multiplied by 8 repeatedly to a precision of three digits.

$219 \div 8 =$	quotient = 27,	remainder = 3	(3 is the low-order digit)
$27 \div 8 =$	quotient = 3,	remainder = 3	
$3 \div 8 =$	quotient = 0,	remainder = 3	

$0.62 \times 8 =$	4.96	4	(4 is the high-order digit)
$0.96 \times 8 =$	7.68	7	
$0.68 \times 8 =$	5.44	5	

Therefore, $219.62_{10} = 333.475_8$.

Decimal to hexadecimal The decimal number 195.828125_{10} will be converted to an equivalent hexadecimal number. The integer is divided by 16 repeatedly and the fraction is multiplied by 16 repeatedly.

$195 \div 16 =$	quotient = 12,	remainder = 3	(3 is the low-order digit)
$12 \div 16 =$	quotient = 0,	remainder = 12 (C)	

$0.828125 \times 16 =$	13.250000	13 (D)	(D is the high-order digit)
$0.250000 \times 16 =$	4.000000	4	

Therefore, $195.828125_{10} = C3.D4_{16}$.

Binary to octal When converting a binary number to octal, the binary number is partitioned into groups of three bits as the number is scanned right to left for integers and scanned left to right for fractions. If the leftmost group of the integer does not contain three bits, then leading zeroes are added to produce a 3-bit octal digit; if the rightmost group of the fraction does not contain three bits, then trailing zeroes are added to produce a 3-bit octal digit. The binary number 10110100011.11101_2 will be converted to an octal number as shown below.

0 1 0	1 1 0	1 0 0	0 1 1	.	1 1 1	0 1 0
2	6	4	3	.	7	2

Binary to hexadecimal When converting a binary number to hexadecimal, the binary number is partitioned into groups of four bits as the number is scanned right to left for integers and scanned left to right for fractions. If the leftmost group of the

integer does not contain four bits, then leading zeroes are added to produce a 4-bit hexadecimal digit; if the rightmost group of the fraction does not contain four bits, then trailing zeroes are added to produce a 4-bit hexadecimal digit. The binary number 11010101000.1111010111_2 will be converted to a hexadecimal number as shown below.

0 1 1 0	1 0 1 0	1 0 0 0	.	1 1 1 1	0 1 0 1	1 1 0 0
6	A	8	.	F	5	C

Octal to binary When converting an octal number to binary, three binary digits are entered that correspond to each octal digit, as shown below.

2	7	5	4	.	3	6
0 1 0	1 1 1	1 0 1	1 0 0	.	0 1 1	1 1 0

When converting from octal to binary-coded octal (BCO) and from octal to binary, the binary bit configurations are identical. This is because the octal number system uses all eight combinations of three bits. An example is shown below in which the octal number 217_8 is converted to binary-coded octal and to binary.

The binary bit configuration for BCO is

$$217_8 = 010\ \ 001\ \ 111_{BCO}$$

The bit configuration for binary is

$$217_8 = (2 \times 8^2) + (1 \times 8^1) + (7 \times 8^0) = 143_{10}$$
$$143_{10} = 010\ 001\ 111_2$$

Octal to hexadecimal To convert from octal to hexadecimal, the octal number is first converted to BCO then partitioned into 4-bit segments to form binary-coded hexadecimal (BCH). The BCH notation is then easily changed to hexadecimal, as shown below.

7			6			3			5			.	4			6				
1	1	1	1	1	0	0	1	1	1	0	1	.	1	0	0	1	1	0	0	0
F			9			D						.	9			8				

Hexadecimal to binary To covert from hexadecimal to binary, substitute the four binary bits for the hexadecimal digits according to Table 1.10 as shown below.

F	A	9	7	.	B	6
1 1 1 1	1 0 1 0	1 0 0 1	0 1 1 1	.	1 0 1 1	0 1 1 0

When converting from hexadecimal to BCH and from hexadecimal to binary, the binary bit configurations are identical. This is because the hexadecimal number system uses all sixteen combinations of four bits. An example is shown below in which the hexadecimal number $1E2_{16}$ is converted to BCH and to binary.

The binary bit configuration for BCH is

$$1E2_8 = 0001\ \ 1110\ \ 0010_{BCH}$$

The bit configuration for binary is

$$1E2_{16} = (1 \times 16^2) + (14 \times 16^1) + (2 \times 16^0) = 482_{10}$$
$$482_{10} = 0001\ 1110\ 0010_2$$

Hexadecimal to octal When converting from hexadecimal to octal, the hexadecimal digits are first converted to binary. Then the binary bits are partitioned into 3-bit segments to obtain the octal digits, as shown below.

B				8				E				.	4				D				
1	0	1	1	1	0	0	0	1	1	1	0	.	0	1	0	0	1	1	0	1	0
5				6				1				.	2				3			2	

Conversion from a nonconventional radix to radix 10 Equation 1.4 will be used to convert the following radix 5 number to an equivalent radix 10 number: 2134.43_5.

$$\begin{aligned} 2134.43_5 &= (2 \times 5^3) + (1 \times 5^2) + (3 \times 5^1) + (4 \times 5^0)\ .\ (4 \times 5^{-1}) + (3 \times 5^{-2}) \\ &= 250 + 25 + 15 + 4 + 0.8 + 0.12 \\ &= 294.92_{10} \end{aligned}$$

Convert from radix *ri* to any other radix *rj* To convert any nondecimal number A_{ri} in radix *ri* to another nondecimal number A_{rj} in radix *rj*, first convert the number A_{ri} to decimal using Equation 1.4, then convert the decimal number to radix *rj* by using repeated division and/or repeated multiplication. The radix 9 number 125_9 will be converted to an equivalent radix 7 number. First 125_9 is converted to radix 10.

$$\begin{aligned} 125_9 &= (1 \times 9^2) + (2 \times 9^1) + (5 \times 9^0) \\ &= 104_{10} \end{aligned}$$

Then, convert 104_{10} to radix 7.

$104 \div 7 =$	quotient = 14,	remainder = 6	(6 is the low-order digit)
$14 \div 7 =$	quotient = 2,	remainder = 0	
$2 \div 7 =$	quotient = 0,	remainder = 2	

Verify the answer.

$$\begin{aligned} 125_9 = 206_7 &= (2 \times 7^2) + (0 \times 7^1) + (6 \times 7^0) \\ &= 104_{10} \end{aligned}$$

Determine the value of an unknown radix The equation shown below has an unknown radix *a*. This example will determine the value of radix *a*.

$$\begin{aligned} 44_a{}^{0.5} &= 6_{10} \\ 44_a &= 36_{10} \\ (4 \times a^1) + (4 \times a^0) &= (3 \times 10^1) + (6 \times 10^0) \\ 4a + 4 &= 30 + 6 \\ 4a &= 32 \\ a &= 8 \end{aligned}$$

Verify the answer.

$$\begin{aligned} 44_8 &= (4 \times 8^1) + (4 \times 8^0) \\ &= 36_{10} \end{aligned}$$

1.2 Number Representations

The material presented thus far covered only positive numbers. However, computers use both positive and negative numbers. Since a computer cannot recognize a plus (+) or a minus (–) sign, an encoding method must be established to represent the sign of a number in which both positive and negative numbers are distributed as evenly as possible.

There must also be a method to differentiate between positive and negative numbers; that is, there must be an easy way to test the sign of a number. Detection of a number with a zero value must be straightforward. The leftmost (high-order) digit is usually reserved for the sign of the number. Consider the following number A with radix r:

$$A = (a_{n-1}\ a_{n-2}\ a_{n-3}\ \ldots\ a_2\ a_1\ a_0)_r$$

where digit a_{n-1} has the following value:

$$A = \begin{cases} 0 & \text{if } A \geq 0 \\ r-1 & \text{if } A < 0 \end{cases} \tag{1.6}$$

The remaining digits of A indicate either the true magnitude or the magnitude in a complemented form. There are three conventional ways to represent positive and negative numbers in a positional number system: sign magnitude, diminished-radix complement, and radix complement. In all three number representations, the high-order digit is the sign of the number, according to Equation 1.6.

$$\begin{aligned} 0 &= \text{ positive} \\ r-1 &= \text{ negative} \end{aligned}$$

1.2.1 Sign Magnitude

In this representation, an integer has the following decimal range:

$$-(r^{n-1}-1) \text{ to } +(r^{n-1}-1) \tag{1.7}$$

where the number zero is considered to be positive. Thus, a positive number A is represented as

$$A = (0\ a_{n-2}a_{n-3}\ \ldots\ a_1a_0)_r \tag{1.8}$$

and a negative number with the same absolute value as

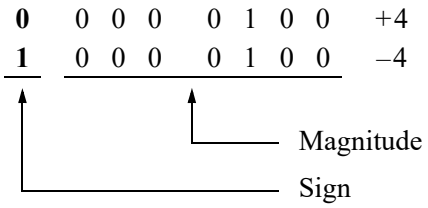
$$A' = [(r-1)\, a_{n-2}a_{n-3} \ \ldots \ a_1a_0]_r \tag{1.9}$$

In sign-magnitude notation, the positive version $+A$ differs from the negative version $-A$ only in the sign digit position. The magnitude portion $a_{n-2}a_{n-3} \ \ldots \ a_1a_0$ is identical for both positive and negative numbers of the same absolute value.

There are two problems with sign-magnitude representation. First, there are two representations for the number 0; specifically +0 and –0; ideally there should be a unique representation for the number 0. Second, when adding two numbers of opposite signs, the magnitudes of the numbers must be compared to determine the sign of the result. This is not necessary in the other two methods that are presented in subsequent sections. Sign-magnitude notation is used primarily for representing fractions in floating-point notation.

Examples of sign-magnitude notation are shown below using 8-bit binary numbers and decimal numbers that represent both positive and negative values. Notice that the magnitude parts are identical for both positive and negative numbers for the same radix.

Radix 2



0	0	0	0	1	1	0	1	.	1	0	1	+13.625
1	0	0	0	1	1	0	1	.	1	0	1	–13.625

0	1	0	1	0	1	1	0	.	0	1	1	+86.375
1	1	0	1	0	1	1	0	.	0	1	1	–86.375

Radix 10

0	7	4	3	+743
9	7	4	3	–743

where 0 represents a positive number in radix 10, and 9 ($r - 1$) represents a negative number in radix 10. Again, the magnitudes of both numbers are identical.

0	6	7	8	4	+6784
9	6	7	8	4	–6784

1.2.2 Diminished-Radix Complement

This is the *(r – 1) complement* in which the radix is diminished by 1 and an integer has the following decimal range:

$$-(r^{n-1} - 1) \text{ to } +(r^{n-1} - 1) \tag{1.10}$$

which is the same as the range for sign-magnitude integers, although the numbers are represented differently, and where the number zero is considered to be positive. Thus, a positive number A is represented as

$$A = (0\ a_{n-2}a_{n-3}\ \ \ldots\ \ a_1a_0)_r \tag{1.11}$$

and a negative number as

$$A' = [(r-1)\ a_{n-2}'a_{n-3}'\ \ \ldots\ \ a_1'a_0']_r \tag{1.12}$$

where

$$a_i' = (r-1) - a_i \tag{1.13}$$

In binary notation ($r = 2$), the diminished-radix complement ($r - 1 = 2 - 1 = 1$) is the 1s complement. Positive and negative integers have the ranges shown below and are represented as shown in Equation 1.11 and Equation 1.12, respectively.

Positive integers: 0 to $2^{n-1} - 1$
Negative integers: 0 to $-(2^{n-1} - 1)$

To obtain the 1s complement of a binary number, simply complement (invert) all the bits. Thus, $0011\ 1100_2$ ($+60_{10}$) becomes $1100\ 0011_2$ (-60_{10}). To obtain the value of a positive binary number, the 1s are evaluated according to their weights in the positional number system, as shown below.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
0	0	1	1	1	1	0	0	$+60_{10}$

To obtain the value of a negative binary number, the 0s are evaluated according to their weights in the positional number system, as shown below.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	1	0	0	0	0	1	1	-60_{10}

When performing arithmetic operations on two operands, comparing the signs is straightforward, because the leftmost bit is a 0 for positive numbers and a 1 for negative numbers. There is, however, a problem when using the diminished-radix complement. There is a dual representation of the number zero, because a word of all 0s (+0) becomes a word of all 1s (–0) when complemented. This does not allow the requirement of having a unique representation for the number zero to be attained. The examples shown below represent the diminished-radix complement for different radices.

Example 1.3 The binary number 1101_2 will be 1s complemented. The number has a decimal value of –2. To obtain the 1s complement, subtract each digit in turn from 1 (the highest number in the radix), as shown below (Refer to Equation 1.12 and Equation 1.13). Or in the case of binary, simply invert each bit. Therefore, the 1s complement of 1101_2 is 0010_2, which has a decimal value of +2.

1	1	1	1
− 1	− 1	− 0	− 1
0	0	1	0

To verify the operation, add the negative and positive numbers to obtain 1111_2, which is zero in 1s complement notation.

	1	1	0	1
+)	0	0	1	0
	1	1	1	1

Example 1.4 Obtain the diminished-radix complement (9s complement) of 08752.43_{10}, where 0 is the sign digit indicating a positive number. The 9s complement is obtained by using Equation 1.12 and Equation 1.13. When a number is

complemented in any form, the number is negated. Therefore, the sign of the complemented radix 10 number is $(r - 1) = 9$. The remaining digits of the number are obtained by using Equation 1.13, such that each digit in the complemented number is obtained by subtracting the given digit from 9. Therefore, the 9s complement of 08752.43_{10} is

$$\frac{9-0}{9}\ \frac{9-8}{1}\ \frac{9-7}{2}\ \frac{9-5}{4}\ \frac{9-2}{7}\ .\ \frac{9-4}{5}\ \frac{9-3}{6}$$

where the sign digit is $(r - 1) = 9$. If the above answer is negated, then the original number will be obtained. Thus, the 9s complement of $91247.56_{10} = 08752.43_{10}$; that is, the 9s complement of -1247.56_{10} is $+8752.43_{10}$, as written in conventional sign magnitude notation for radix 10.

Example 1.5 The diminished-radix complement of the positive decimal number 06784_{10} will be 9s complemented. To obtain the 9s complement, subtract each digit in turn from 9 (the highest number in the radix), as shown below to obtain the negative number with the same absolute value. The sign of the positive number is 0 and the sign of the negative number is 9 (refer to Equation 1.11 and Equation 1.12).

$$\frac{9-0}{9}\ \frac{9-6}{3}\ \frac{9-7}{2}\ \frac{9-8}{1}\ \frac{9-4}{5}$$

To verify the operation, add the negative and positive numbers to obtain 99999_{10}, which is zero in 9s complement notation.

$$\begin{array}{rrrrrr} & 0 & 6 & 7 & 8 & 4 \\ +) & 9 & 3 & 2 & 1 & 5 \\ \hline & 9 & 9 & 9 & 9 & 9 \end{array}$$

Example 1.6 The diminished-radix complement of the positive radix 8 number 05734_8 will be 7s complemented. To obtain the 7s complement, subtract each digit in turn from 7 (the highest number in the radix), as shown below to obtain the negative number with the same absolute value. The sign of the positive number is 0 and the sign of the negative number is 7 (refer to Equation 1.11 and Equation 1.12).

$$\frac{7-0}{7}\ \frac{7-5}{2}\ \frac{7-7}{0}\ \frac{7-3}{4}\ \frac{7-4}{3}$$

To verify the operation, add the negative and positive numbers to obtain 77777_8, which is zero in 7s complement notation.

$$\begin{array}{rrrrrr} & 0 & 5 & 7 & 3 & 4 \\ +) & 7 & 2 & 0 & 4 & 3 \\ \hline & 7 & 7 & 7 & 7 & 7 \end{array}$$

Example 1.7 The diminished-radix complement of the positive radix 16 number $0A7C4_{16}$ will be 15s complemented. To obtain the 15s complement, subtract each digit in turn from 15 (the highest number in the radix), as shown below to obtain the negative number with the same absolute value. The sign of the positive number is 0 and the sign of the negative number is F (refer to Equation 1.11 and Equation 1.12).

$$\begin{array}{ccccc} \underline{F-0} & \underline{F-A} & \underline{F-7} & \underline{F-C} & \underline{F-4} \\ F & 5 & 8 & 3 & B \end{array}$$

To verify the operation, add the negative and positive numbers to obtain $FFFFF_{16}$, which is zero in 15s complement notation.

$$\begin{array}{rccccc} & 0 & A & 7 & C & 4 \\ +) & F & 5 & 8 & 3 & B \\ \hline & F & F & F & F & F \end{array}$$

1.2.3 Radix Complement

This is the *r complement*, where an integer has the following decimal range:

$$-(r^{n-1}) \text{ to } +(r^{n-1} - 1) \tag{1.14}$$

where the number zero is positive. A positive number A is represented as

$$A = (0\ a_{n-2}a_{n-3}\ \ldots\ a_1a_0)_r \tag{1.15}$$

and a negative number as

$$(A')_{+1} = \{[(r-1)\ a_{n-2}'a_{n-3}'\ \ldots\ a_1'a_0'] + 1\}_r \tag{1.16}$$

where A' is the diminished-radix complement. Thus, the radix complement is obtained by adding 1 to the diminished-radix complement; that is, $(r-1) + 1 = r$. Note that all three number representations have the same format for positive numbers and differ only in the way that negative numbers are represented, as shown in Table 1.14.

Table 1.14 Number Representations for Positive and Negative Integers of the Same Absolute Value for Radix *r*

Number Representation	Positive Numbers	Negative Numbers
Sign magnitude	$0\ a_{n-2}a_{n-3}\ \ldots\ a_1a_0$	$(r-1)\ a_{n-2}a_{n-3}\ \ldots\ a_1a_0$
Diminished-radix complement	$0\ a_{n-2}a_{n-3}\ \ldots\ a_1a_0$	$(r-1)\ a_{n-2}'a_{n-3}'\ \ldots\ a_1'a_0'$
Radix complement	$0\ a_{n-2}a_{n-3}\ \ldots\ a_1a_0$	$(r-1)\ a_{n-2}'a_{n-3}'\ \ldots\ a_1'a_0' + 1$

Another way to define the radix complement of a number is shown in Equation 1.17, where n is the number of digits in A.

$$(A')_{+1} = r^n - A_r \tag{1.17}$$

For example, assume that $A = 0101\ 0100_2$ ($+84_{10}$). Then, using Equation 1.17,

$2^8 = 256_{10} = 10000\ 0000_2$. Thus, $256_{10} - 84_{10} = 172_{10}$

$(A')_{+1} = 2^8 - (0101\ 0100)$

$$\begin{array}{rcccccccc} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -) & & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ \hline & & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{array}$$

As can be seen from the above example, to generate the radix complement for a radix 2 number, keep the low-order 0s and the first 1 unchanged and complement (invert) the remaining high-order bits. To obtain the value of a negative number in radix 2, the 0s are evaluated according to their weights in the positional number system, then add 1 to the value obtained.

$$\begin{array}{cccc|cccc} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & \\ \hline \multicolumn{4}{c|}{80} & \multicolumn{4}{c|}{3 + 1 = 4} & -84_{10} \end{array}$$

Table 1.15 and Table 1.16 show examples of the three number representations for positive and negative numbers in radix 2. Note that the positive numbers are identical for all three number representations; only the negative numbers change.

Table 1.15 Number Representations for Positive and Negative Integers in Radix 2

Number Representation	$+127_{10}$	-127_{10}
Sign magnitude	**0** 111 1111	**1** 111 1111
Diminished-radix complement (1s)	**0** 111 1111	**1** 000 0000
Radix complement (2s)	**0** 111 1111	**1** 000 0001

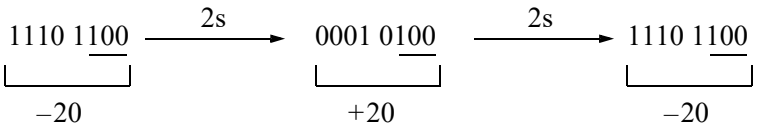
Table 1.16 Number Representations for Positive and Negative Integers in Radix 2

Number Representation	$+54_{10}$	-54_{10}
Sign magnitude	**0** 011 0110	**1** 011 0110
Diminished-radix complement (1s)	**0** 011 0110	**1** 100 1001
Radix complement (2s)	**0** 011 0110	**1** 100 1010

There is a unique zero for binary numbers in radix complement, as shown below. When the number zero is 2s complemented, the bit configuration does not change. The 2s complement is formed by adding 1 to the 1s complement.

$$\begin{array}{rcccccccc} \text{Zero in 2s complement} = & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{Form the 1s complement} = & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \text{Add 1} = & & & & & & & & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Example 1.8 Convert -20_{10} to binary and obtain the 2s complement. Then obtain the 2s complement of $+20_{10}$ using the fast method of keeping the low-order 0s and the first 1 unchanged as the number is scanned from right to left, then inverting all remaining bits.



Example 1.9 Obtain the radix complement (10s complement) of the positive number 08752.43_{10}. Determine the 9s complement as in Example 1.5, then add 1. The 10s complement of 08752.43_{10} is the negative number 91247.57_{10}.

$$\begin{array}{rccccccc} & \frac{9-0}{9} & \frac{9-8}{1} & \frac{9-7}{2} & \frac{9-5}{4} & \frac{9-2}{7} & . \frac{9-4}{5} & \frac{9-3}{6} \\ +) & & & & & & & 1 \\ \hline & 9 & 1 & 2 & 4 & 7 & \cdot 5 & 7 \end{array}$$

Adding 1 to the 9s complement in this example is the same as adding 10^{-2} $(.01_{10})$. To verify that the radix complement of 08752.43_{10} is 91247.57_{10}, the sum of the two numbers should equal zero for radix 10. This is indeed the case, as shown below.

$$\begin{array}{rl} & 08752.43 \\ +) & \underline{91247.57} \\ & 00000.00 \end{array}$$

Example 1.10 Obtain the 10s complement of 0.4572_{10} by adding a power of ten to the 9s complement of the number. The 9s complement of 0.4572_{10} is

$$9-0=9 \quad 9-4=5 \quad 9-5=4 \quad 9-7=2 \quad 9-2=7$$

Therefore, the 10s complement of $0.4572_{10} = 9.5427_{10} + 10^{-4} = 9.5428_{10}$. Adding the positive and negative numbers again produces a zero result.

Example 1.11 Obtain the radix complement of $1111\ 1111_2$ (-1_{10}). The answer can be obtained using two methods: add 1 to the 1s complement or keep the low-order 0s and the first 1 unchanged, then invert all remaining bits. Since there are no low-order 0s, only the rightmost 1 is unchanged. Therefore, the 2s complement of $1111\ 1111_2$ is $0000\ 0001_2$ $(+1_{10})$.

Example 1.12 Obtain the 8s complement of 04360_8. First form the 7s complement, then add 1. The 7s complement of 04360_8 is 73417_8. Therefore, the 8s complement of 04360_8 is $73417_8 + 1$, as shown below, using the rules for octal addition. Adding the positive and negative numbers results in a sum of zero.

$$\begin{array}{rccccc} & 7 & 3 & 4 & 1 & 7 \\ +) & & & & & 1 \\ \hline & 7 & 3 & 4 & 2 & 0 \end{array}$$

Example 1.13 Obtain the 16s complement of $F8A5_{16}$. First form the 15s complement, then add 1. The 15s complement of $F8A5_{16}$ is $075A_{16}$. Therefore, the 16s complement of $F8A5_{16} = 075A_{16} + 1 = 075B_{16}$. Adding the positive and negative numbers results in a sum of zero.

Example 1.14 Obtain the 4s complement of 0231_4. The rules for obtaining the radix complement are the same for any radix: generate the diminished-radix complement, then add 1. Therefore, the 4s complement of $0231_4 = 3102_4 + 1 = 3103_4$. To verify the result, add $0231_4 + 3103_4 = 0000_4$, as shown below using the rules for radix 4 addition.

$$\begin{array}{rcccc} & 0 & 2 & 3 & 1 \\ +) & 3 & 1 & 0 & 3 \\ \hline & 0 & 0 & 0 & 0 \end{array}$$

1.2.4 Arithmetic Operations

This section will concentrate on fixed-point binary and binary-coded decimal operations, since these are the dominant number representations in computers — floating-point operations will be discussed in the chapter on computer arithmetic. Examples of addition, subtraction, multiplication, and division will be presented for fixed-point binary using radix complementation and for binary-coded decimal number representations.

Binary Addition Numbers in radix complement representation are designated as signed numbers, specifically as 2s complement numbers in binary. The sign of a binary number can be extended to the left indefinitely without changing the value of the number. For example, the numbers 0000000001010_2 and 00001010_2 both represent a value of $+10_{10}$; the numbers 1111111110110_2 and 11110110_2 both represent a value of -10_{10}.

Thus, when an operand must have its sign extended to the left, the expansion is achieved by setting the extended bits equal to the leftmost (sign) bit. The maximum positive number consists of a 0 followed by a field of all 1s, dependent on the word size of the operand. Similarly, the maximum negative number consists of a 1 followed by a field of all 0s, dependent on the word size of the operand.

The radix (or binary) point can be in any fixed position in the number — thus the radix point is referred to as *fixed-point*. For integers, however, the radix point is positioned to the immediate right of the low-order bit position. There are normally two operands for addition, as shown below.

$$\begin{array}{rcl} & A = & \text{Augend} \\ +) & B = & \underline{\text{Addend}} \\ & & \text{Sum} \end{array}$$

The rules for binary addition are as follows:

+	0	1
0	0	1
1	1	0 *

* 1 + 1 = 0 with a carry to the next column on the left. The sum of $1_2 + 1_2$ requires a 2-bit vector 10_2 to represent the value of 2_{10}.

Example 1.15 Add $+51_{10}$ and $+32_{10}$ to yield $+83_{10}$. The sum is also referred to as *true addition*, because the result is the sum of the two numbers, ignoring the sign of the numbers. The carry-out is 0, which can be ignored, since it is the sign extension of the sum.

	A =	0	0	1	1	0	0	1	1	(+51)
+)	B =	0	0	1	0	0	0	0	0	(+32)
0 ←		0	1	0	1		0	1	1	(+83)

Example 1.16 Add -51_{10} and -32_{10} to yield -83_{10}. The sum is also referred to as *true addition*, because the result is the sum of the two numbers, ignoring the sign of the numbers. The carry-out is 1, which can be ignored, since it is the sign extension of the sum.

	A =	1	1	0	0	1	1	0	1	(–51)
+)	B =	1	1	1	0	0	0	0	0	(–32)
1 ←		1	0	1	0		1	0	1	(–83)

Example 1.17 Add $+51_{10}$ and -32_{10} to yield $+19_{10}$. The sum is also referred to as *true subtraction*, because the result is the difference of the two numbers, ignoring the sign of the numbers. The carry-out is 1, which can be ignored, since it does not constitute an overflow when adding two numbers of opposite signs.

	A =	0	0	1	1	0	0	1	1	(+51)
+)	B =	1	1	1	0	0	0	0	0	(–32)
1 ←		0	0	0	1		0	1	1	(+19)

Example 1.18 Add -51_{10} and $+32_{10}$ to yield -19_{10}. The sum is also referred to as *true subtraction*, because the result is the difference of the two numbers, ignoring the sign of the numbers. The carry-out is 0, which can be ignored, since it does not constitute an overflow when adding two numbers of opposite signs.

	$A =$	1	1	0	0	1	1	0	1	(–51)
+)	$B =$	0	0	1	0	0	0	0	0	(+32)
0	←	1	1	1	0		1	0	1	(–19)

Overflow Overflow occurs when the result of an arithmetic operation (usually addition) exceeds the word size of the machine; that is, the sum is not within the representable range of numbers provided by the number representation. For numbers in 2s complement representation, the range is from -2^{n-1} to $+2^{n-1} - 1$. For two n-bit numbers

$$A = a_{n-1}a_{n-2}a_{n-3} \ldots a_1a_0$$
$$B = b_{n-1}\, b_{n-2}\, b_{n-3} \ldots b_1b_0$$

a_{n-1} and b_{n-1} are the sign bits of operands A and B, respectively. Overflow can be detected by either of the following two equations:

$$\text{Overflow} = (a_{n-1} \bullet b_{n-1} \bullet s_{n-1}') \; + \; (a_{n-1}' \bullet b_{n-1}' \bullet s_{n-1})$$
$$\text{Overflow} = c_{n-1} \oplus c_{n-2} \tag{1.18}$$

where the symbol "•" is the logical AND operator, the symbol "+" is the logical OR operator, the symbol "⊕" is the exclusive-OR operator as defined below, and c_{n-1} and c_{n-2} are the carry bits out of positions $n - 1$ and $n - 2$, respectively.

⊕	0	1
0	0	1
1	1	0

Therefore, overflow $= c_{n-1} \oplus c_{n-2}$

$$= (c_{n-1} \bullet c_{n-2}') + (c_{n-1}' \bullet c_{n-2})$$

Thus, overflow produces an erroneous sign reversal and is possible only when both operands have the same sign. An overflow cannot occur when adding two operands of different signs, since adding a positive number to a negative number produces a result that falls within the limit specified by the two numbers.

Example 1.19 Given the following two positive numbers in radix complementation for radix 2, perform an add operation:

$A = 0100\ 0100\ (+68)$
$B = 0101\ 0110\ (+86)$

	$A =$	0100	0100	(+68)	
+)	$B =$	0101	0110	(+86)	
		1001	1010	(+154)	Overflow. +154 takes 9 bits

If 9 bits are used for the result, then the answer is $+154_{10}$. However, the value of +154 cannot be contained in only 8 bits; therefore, an overflow has occurred.

Example 1.20 Given the following two negative numbers in radix complementation for radix 2, perform an add operation:

$A = 1011\ 1100\ (-68)$
$B = 1010\ 1010\ (-86)$

	$A =$	1011	1100	(–68)	
+)	$B =$	1010	1010	(–86)	
		0110	0110	(–154)	Overflow. –154 takes 9 bits

If all 9 bits are used for the result, then the answer is -154_{10}. However, the value of -154_{10} cannot be contained in only 8 bits; therefore, an overflow has occurred. The range for binary numbers in 2s complement representation is

$$-2^{n-1} \text{ to } +2^{n-1} - 1$$

For $n = 8$: -128 to $+127$
In binary: 1000 0000 to 0111 1111

In the above two examples, Equation 1.18 holds true, indicating that an overflow has occurred. That is, the signs of the two operands are the same, but the sign of the

result is different. Also, the carry out of the second high-order bit position and the carry out of the high-order bit position are different.

Binary subtraction The operands used for subtraction are the minuend and subtrahend, as shown below. The rules for binary subtraction presented in Section 1.1.5, sometimes referred to as the *paper-and-pencil* method, are not easily applicable to computer subtraction. The method used by most processors is to add the 2s complement of the subtrahend to the minuend; that is, change the sign of the minuend and add the resulting two operands. This can be described formally as shown in Equation 1.19.

$$\begin{array}{rl} A = & \text{Minuend} \\ -)\ B = & \underline{\text{Subtrahend}} \\ & \text{Difference} \end{array}$$

$$\begin{aligned} \text{Difference} &= A - B \\ &= A - r^n + r^n - B \\ &= A - r^n + (r^n - B) \\ &= A - r^n + (B' + 1) \end{aligned} \tag{1.19}$$

where B' is the diminished-radix complement (1s complement) of the subtrahend B. An example using Equation 1.19 is shown below.

$$\begin{array}{rl} A = & 0111 \\ -)\ B = & \underline{0011} \end{array} \longrightarrow \begin{array}{rl} A = & 0111 \\ +)\ B = & \underline{1101} \\ & 0100 \end{array}$$

Using Equation 1.19 where $n = 4$ and $r = 2$, the following result is observed:

$$\begin{aligned} \text{Difference} &= A - r^n + (B' + 1) \\ &= 7 - 2^4 + (12 + 1) \\ &= 7 - 16 + 13 \\ &= 4 \end{aligned}$$

Example 1.21 Perform the operation $A - B$ for the following two positive operands:

$A = 0001\ 0000\ (+16)$
$B = 0000\ 1010\ (+10)$

$A - B = A + (B' + 1)$

	$A =$	0001 0000	(+16)	
+)	$B' + 1 =$	1111 0110	(–10)	2s complement of B
	$A - B =$	0000 0110	(+6)	

Example 1.22 Perform the operation $B - A$ for the following two positive operands:

$A = 0001\ 0000\ (+16)$
$B = 0000\ 1010\ (+10)$

$B - A = B + (A' + 1)$

	$B =$	0000 1010	(+10)	
+)	$A' + 1 =$	1111 0000	(–16)	2s complement of A
	$B - A =$	1111 1010	(–6)	

Example 1.23 Perform the operation $A - B$ for the following two negative operands:

$A = 1111\ 0000\ (-16)$
$B = 1111\ 0110\ (-10)$

$A - B = A + (B' + 1)$

	$A =$	1111 0000	(–16)	
+)	$B' + 1 =$	0000 1010	(+10)	2s complement of B
	$A - B =$	1111 1010	(–6)	

Example 1.24 Perform the operation $B - A$ for the following two negative operands:

$A = 1111\ 0000\ (-16)$
$B = 1111\ 0110\ (-10)$

$B - A = B + (A' + 1)$

	$B =$	1111 0110	(–10)	
+)	$A' + 1 =$	0001 0000	(+16)	2s complement of A
	$B - A =$	0000 0110	(+6)	

Example 1.25 As stated previously, the diminished-radix complement is rarely used in arithmetic applications because of the dual interpretation of the number zero. This example illustrates another disadvantage of the diminished-radix complement. Given the two radix 2 numbers shown below in 1s complement representation, obtain the difference.

$A = 1111\ 1001$ (–6)
$B = 1110\ 1101$ (–18)
$A - B = A + B'$ (1s complement of B)

	$A =$	1111 1001	(–6)	
+)	$B' =$	0001 0010	(+18)	1s complement of B
	1 ←	0000 1011	(+11)	Incorrect result
		→ 1		End-around carry
	$A - B =$	0000 1100	(+12)	

When performing a subtract operation using 1s complement operands, an end-around carry will result if at least one operand is negative. As can be seen above, the result will be incorrect (+11) if the carry is not added to the intermediate result. Although 1s complementation may seem easier than 2s complementation, the result that is obtained after an add operation is not always correct. In 1s complement notation, the final carry-out c_{n-1} cannot be ignored. If the carry-out is zero, then the result is correct. Thus, 1s complement subtraction may result in an extra add cycle to obtain the correct result.

Binary multiplication The multiplication of two n-bit operands results in a product of $2n$ bits, as shown below.

$A = a_{n-1}\, a_{n-2}\, a_{n-3} \ldots a_1\, a_0$
$B = b_{n-1}\, b_{n-2}\, b_{n-3} \ldots b_1\, b_0$

Product of $A \times B = p_{2n-1}\, p_{2n-2}\, p_{2n-3} \ldots p_1\, p_0$

	$A =$	Multiplicand (n bits)
×	$B =$	Multiplier (n bits)
	$P =$	Product ($2n$ bits)

Multiplication of two fixed-point binary numbers in 2s complement representation is done by a process of successive add and shift operations. The process consists of multiplying the multiplicand by each multiplier bit as the multiplier is scanned right to left. If the multiplier bit is a 1, then the multiplicand is copied as a partial product; otherwise, 0s are inserted as a partial product. The partial products inserted into successive lines are shifted left one bit position from the previous partial product. The partial products are then added to form the product.

The sign of the product is determined by the signs of the operands: If the signs are the same, then the sign of the product is plus; if the signs are different, then the sign of the product is minus. In this sequential add-shift technique, however, the multiplier must be positive. When the multiplier is positive, the bits are treated the same as in the sign-magnitude representation. When the multiplier is negative, the low-order 0s and the first 1 are treated the same as a positive multiplier, but the remaining high-order sign bits are treated as 1s and not the sign bits. Therefore, the algorithm treats the multiplier as unsigned, or positive.

The problem is easily solved by forming the 2s complement of both the multiplier and the multiplicand. An alternative approach is to 2s complement the negative multiplier leaving the multiplicand unchanged. Depending on the signs of the initial operands, it may be necessary to complement the product.

Example 1.26 Multiply the positive operands of 0111_2 ($+7_{10}$) and 0101_2 ($+5_{10}$). Since both operands are positive, the product will be positive ($+35_{10}$).

				0	1	1	1	(+7)
			×	0	1	0	1	(+5)
0	0	0	0	0	1	1	1	
0	0	0	0	0	0	0		
0	0	0	1	1	1			
0	0	0	0	0				
	0	1	0	0	0	1	1	(+35)

Example 1.27 Multiply a negative multiplicand 1100_2 (–4) by a positive multiplier 0011_2 (+3) to obtain a product of 1111 0100_2 (–12). Since the multiplier is positive, the sign of the product will be correct.

				1	1	0	0	(–4)
			×	0	0	1	1	(+3)
1	1	1	1	1	1	0	0	
1	1	1	1	1	0	0		
0	0	0	0	0	0			
0	0	0	0	0				
	1	1	1	0	1	0	0	(–12)

Example 1.28 In this example, a positive multiplicand 0100_2 ($+4_{10}$) will be multiplied by a negative multiplier 1101_2 (-3_{10}). The product should be -12_{10}; however, the negative multiplier is treated as an unsigned (or positive) integer of $+13_{10}$ by the algorithm. Therefore, both the multiplicand and the multiplier will be 2s complemented (negated) to provide a positive multiplier. This will result in a correctly signed product of -12_{10}.

				0	1	0	0	(+4)
			×)	1	1	0	1	(–3)
0	0	0	0	0	1	0	0	
0	0	0	0	0	0	0		
0	0	0	1	0	0			
0	0	1	0	0				
	0	1	1	0	1	0	0	(+52)

Both the multiplicand and the multiplier will now be 2s complemented so that the multiplier will be positive. Since the multiplicand can be either positive or negative, the product will have the correct sign.

				1	1	0	0	(–4)
			×)	0	0	1	1	(+3)
1	1	1	1	1	1	0	0	
1	1	1	1	1	0	0		
0	0	0	0	0	0			
0	0	0	0	0				
	1	1	1	0	1	0	0	(–12)

Binary division Division has two operands that produce two results, as shown below. Unlike multiplication, division is not commutative; that is, $A \div B \neq B \div A$, except when $A = B$.

$$\frac{\text{Dividend}}{\text{Divisor}} = \text{Quotient} + \text{Remainder}$$

All operands for a division operation comply with the following equation:

$$\text{Dividend} = (\text{Quotient} \times \text{Divisor}) + \text{Remainder} \tag{1.20}$$

The remainder has a smaller value than the divisor and has the same sign as the dividend. If the divisor B has n bits, then the dividend A has $2n$ bits and the quotient Q and remainder R both have n bits, as shown below.

$$A = a_{2n-1}\, a_{2n-2} \ldots a_n\, a_{n-1} \ldots a_1\, a_0$$

$$B = b_{n-1}\, b_{n-2} \ldots b_1\, b_0$$

$$Q = q_{n-1}\, q_{n-2} \cdots q_1\, q_0$$

$$R = r_{n-1}\, r_{n-2} \ldots r_1\, r_0$$

The sign of the quotient is q_{n-1} and is determined by the rules of algebra; that is,

$$q_{n-1} = a_{2n-1} \oplus b_{n-1}$$

Multiplication is a shift-add multiplicand operation, whereas division is a shift-subtract divisor operation. The result of a shift-subtract operation determines the next operation in the division sequence. If the partial remainder is negative, then the carry-out is 0 and becomes the low-order quotient bit q_0. The partial remainder thus obtained is restored to the value of the previous partial remainder. If the partial remainder is positive, then the carry-out is 1 and becomes the low-order quotient bit q_0. The partial remainder is not restored. This technique is referred to as *restoring division*.

Example 1.29 An example of restoring division using a hardware algorithm is shown in Figure 1.20, where the dividend is in register-pair A Q is $0000\ 0111_2$ $(+7_{10})$ and the divisor is in register B is 0011_2 $(+3_{10})$. The algorithm is implemented with a subtractor and a $2n$-bit shift register.

The first operation in Figure 1.20 is to shift the dividend left 1 bit position, then subtract the divisor, which is accomplished by adding the 2s complement of the divisor. Since the result of the subtraction is negative, the dividend is restored by adding back the divisor, and the low-order quotient bit q_0 is set to 0. This sequence repeats for four cycles — the number of bits in the divisor. If the result of the subtraction was positive, then the partial remainder is placed in register A, the high-order half of the dividend, and q_0 is set to 1.

The division algorithm is slightly more complicated when one or both of the operands are negative. The operands can be preprocessed and/or the results can be post-processed to achieved the desired results. The negative operands are converted to positive numbers by 2s complementation before the division process begins. The resulting quotient is then 2s complemented, if necessary.

Unlike multiplication, overflow can occur in division. This happens when the high-order half of the dividend is greater than or equal to the divisor. Also, division by zero must be avoided. Both of these problems can be detected before the division process begins by subtracting the divisor from the high-order half of the dividend. If the difference is positive, then an overflow or a divide by zero has been detected.

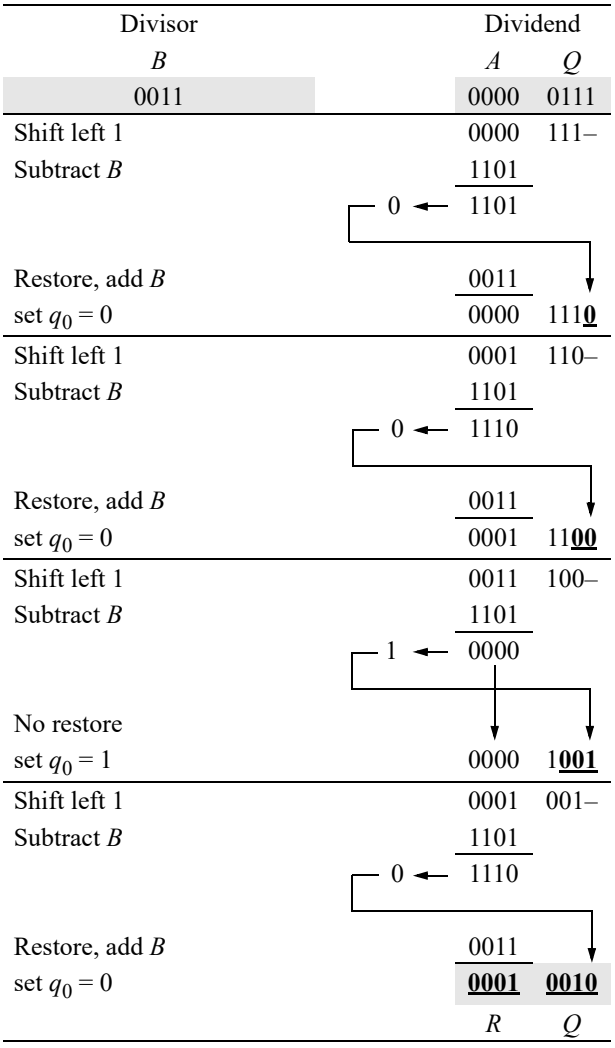


Figure 1.20 Example of binary division using two positive operands.

Figure 1.21 illustrates another example of binary division, this time using a positive dividend $0000\ 0111_2$ ($+7_{10}$) and a negative divisor 1101_2 (-3_{10}). This will result in a quotient of 1110_2 (-2_{10}) and a remainder of 0001_2 ($+1_{10}$). The results can be verified using Equation 1.20. Because the divisor is negative, it is added to the partial remainder, not subtracted from the partial remainder, as was the case in the division example of Figure 1.20.

Divisor		Dividend	
B		*A*	*Q*
1101		0000	0111
Shift left 1		0000	111–
Add *B*		1101	
	0 ←	1101	
Restore, add *B*'		0011	
set $q_0 = 0$		0000	1110
Shift left 1		0001	110–
Add *B*		1101	
	0 ←	1110	
Restore, add *B*'		0011	
set $q_0 = 0$		0001	1100
Shift left 1		0011	100–
Add *B*		1101	
	1 ←	0000	
No restore			
set $q_0 = 1$		0000	1001
Shift left 1		0001	001–
Add *B*		1101	
	0 ←	1110	
Restore, add *B*'		0011	
set $q_0 = 0$		0001	0010
		R	
Because the signs of the dividend and divisor are different, the quotient is 2s complemented			1110 *Q*

Figure 1.21 Example of binary division using a positive dividend and a negative divisor.

Use Equation 1.20 to verify the results.

$$\begin{aligned} \text{Dividend} &= (\text{Quotient} \times \text{Divisor}) + \text{Remainder} \\ 7 &= (-2 \times -3) + 1 \\ 7 &= 6 + 1 \end{aligned}$$

Binary-coded decimal addition When two binary-coded decimal (BCD) digits are added, the range is 0 to 18. If the carry-in $c_{in} = 1$, then the range is 0 to 19. If the sum digit is ≥ 10 (1010_2), then it must be adjusted by adding 6 (0110_2). This excess-6 technique generates the correct BCD sum and a carry to the next higher-order digit, as shown below in Figure 1.22 (a). A carry-out of a BCD sum will also cause an adjustment to be made to the sum — called the intermediate sum — even though the intermediate sum is a valid BCD digit, as shown in Figure 1.22 (b).

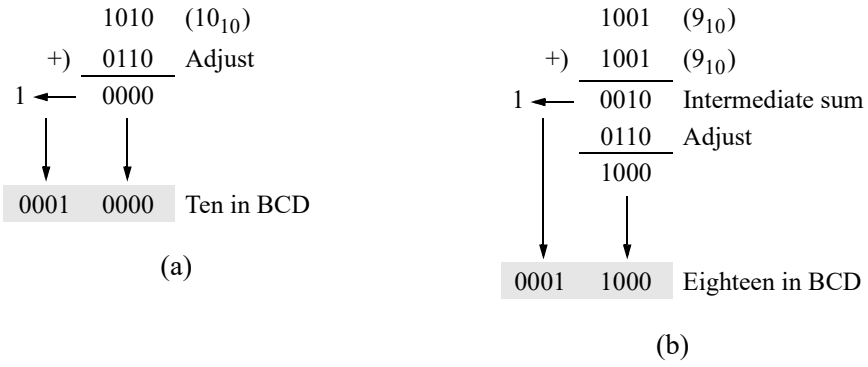


Figure 1.22 Example showing adjustment of a BCD sum.

There are three conditions that indicate when the sum of a BCD addition (the intermediate sum) should be adjusted by adding six.

1. Whenever bit positions 8 and 2 are both 1s.
2. Whenever bit positions 8 and 4 are both 1s.
3. Whenever the unadjusted sum produces a carry-out.

The examples of Figure 1.23, Figure 1.24, and Figure 1.25 illustrate these three conditions.

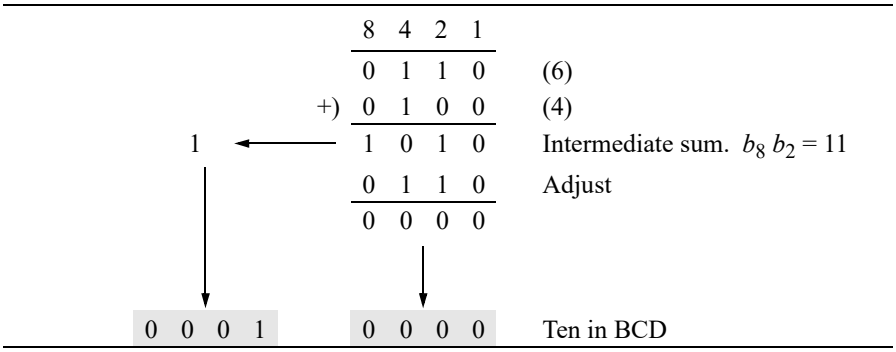


Figure 1.23 Example showing adjustment for BCD addition.

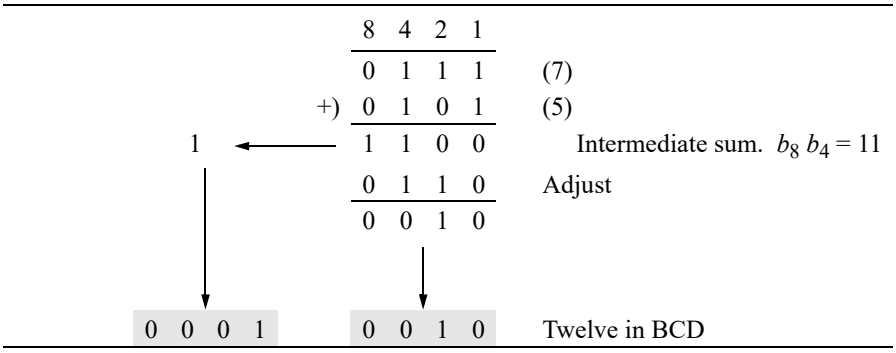


Figure 1.24 Example showing adjustment for BCD addition.

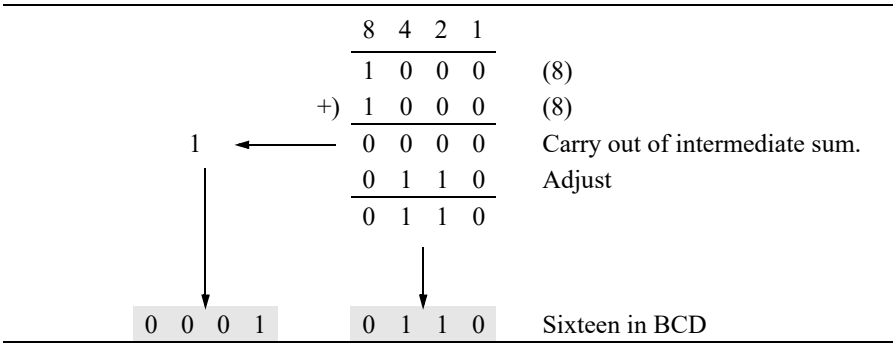


Figure 1.25 Example showing adjustment for BCD addition.

Therefore, all three conditions produce a carry-out to the next higher-order stage; that is,

$$\text{Carry} = c_8 + b_8 b_4 + b_8 b_2 \tag{1.21}$$

where c_8 is the carry-out of the high-order bit; $b_8 b_4$ and $b_8 b_2$ are bits of the unadjusted sum.

The algorithms used for BCD arithmetic are basically the same as those used for fixed-point arithmetic for radix 2. The main difference is that BCD arithmetic treats each digit as four bits, whereas fixed-point arithmetic treats each digit as a bit. Shifting operations are also different — decimal shifting is performed on 4-bit increments.

Binary-coded decimal numbers may also have a sign associated with them. This usually occurs in the *packed* BCD format, as shown below, where the sign digits are the low-order bytes 1100 (+) and 1101 (–). Comparing signs is straightforward. This sign notation is similar to the sign notation used in binary. Note that the low-order bit of the sign digit is 0 (+) or 1 (–).

+53	0 0 0 0	0 1 0 1	0 0 1 1	1 1 0 0 Sign is +
–25	0 0 0 0	0 0 1 0	0 1 0 1	1 1 0 1 Sign is –

Example 1.30 The following decimal numbers will be added using BCD arithmetic 26 and 18, as shown in Figure 1.26.

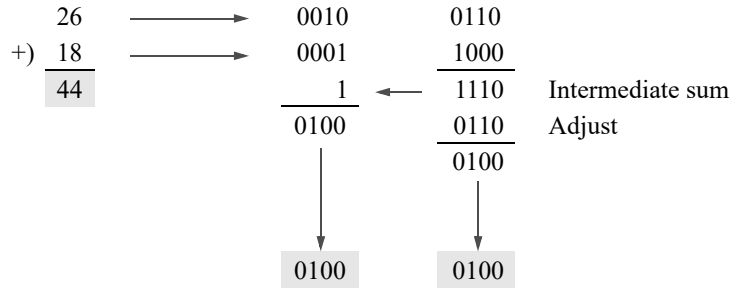


Figure 1.26 Example of BCD addition.

Example 1.31 The following decimal numbers will be added using BCD arithmetic 436 and 825, as shown in Figure 1.27.

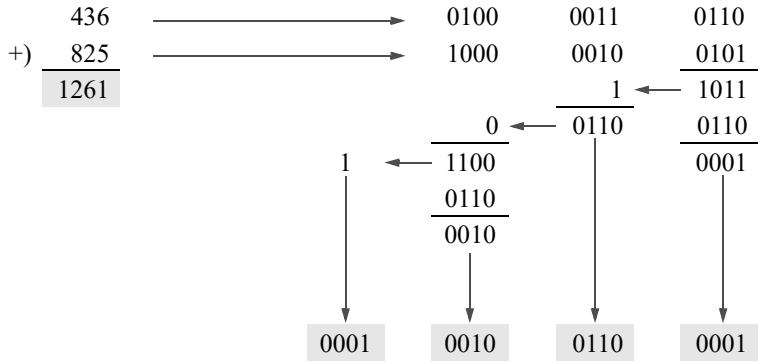


Figure 1.27 Example of BCD addition.

Binary-coded decimal subtraction Subtraction in BCD is essentially the same as in fixed-point binary: Add the *r*s complement of the subtrahend to the minuend, which for BCD is the 10s complement. The examples which follow show subtract operations using BCD numbers. Negative results can remain in 10s complement notation or be recomplemented to sign-magnitude notation with a negative sign.

Example 1.32 The following decimal numbers will be subtracted using BCD arithmetic: +30 and +20, as shown in Figure 1.28. This can be considered as true subtraction, because the result is the difference of the two numbers, ignoring the signs. A carry-out of 1 from the high-order decade indicates a positive number. A carry-out of 0 from the high-order decade indicates a negative number in 10s complement notation. The number can be changed to an absolute value by recomplementing the number and changing the sign to indicate a negative value.

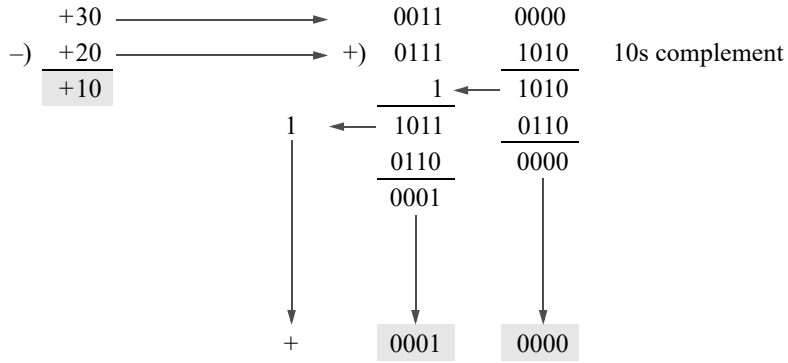


Figure 1.28 Example of BCD subtraction.

Example 1.33 The following decimal numbers will be added using BCD arithmetic: +28 and –20, as shown in Figure 1.29. This can be considered as true subtraction, because the result is the difference of the two numbers, ignoring the signs.

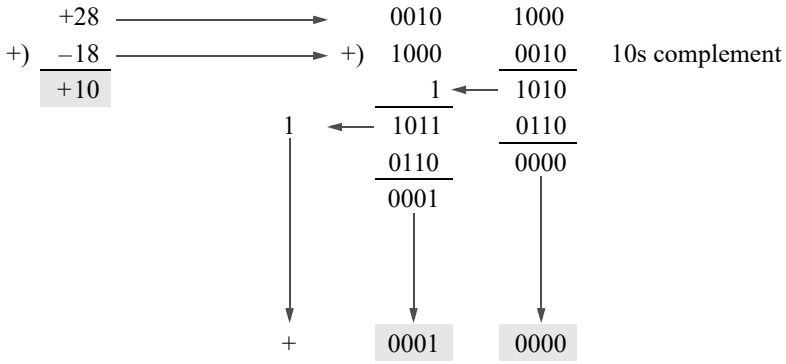


Figure 1.29 Example of BCD subtraction.

Example 1.34 The following decimal numbers will be subtracted using BCD arithmetic: +482 and +627, resulting in a difference of –145, as shown in Figure 1.30.

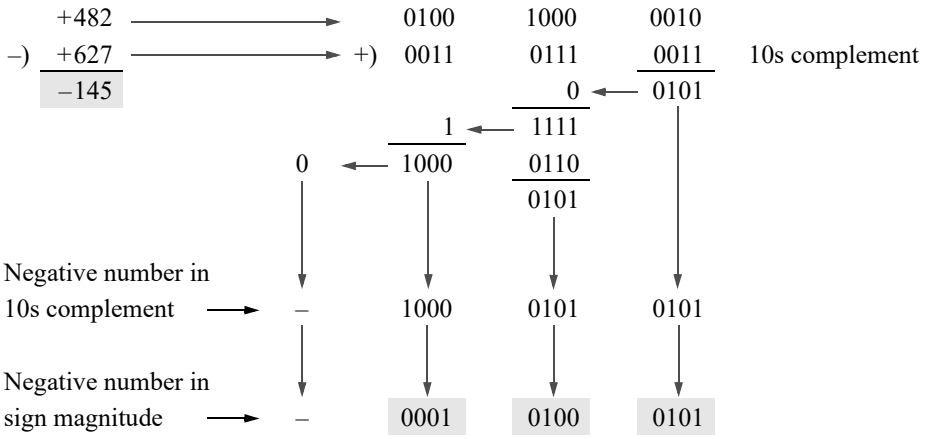


Figure 1.30 Example of BCD subtraction.

Binary-coded decimal multiplication The multiplication algorithms for decimal multiplication are similar to those for fixed-point binary multiplication except in

the way that the partial products are formed. In binary multiplication, the multiplicand is added to the previous partial product if the multiplier bit is a 1. In BCD multiplication, the multiplicand is multiplied by each digit of the multiplier and these subpartial products are aligned and added to form a partial product. When adding digits to obtain a partial product, adjustment may be required to form a valid BCD digit. Two examples of BCD multiplication are shown below.

Example 1.35 The following decimal numbers will be multiplied using BCD arithmetic: +67 and +9, resulting in a product of +603, as shown in Figure 1.31.

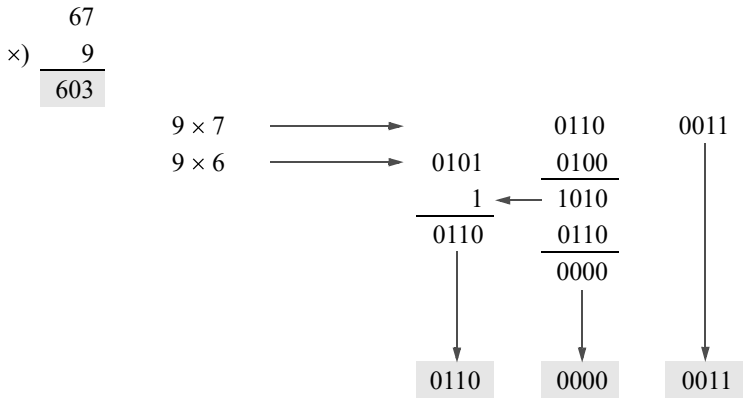


Figure 1.31 Example of BCD multiplication.

Example 1.36 The following decimal numbers will be multiplied using BCD arithmetic: +6875 and +46, resulting in a product of +316250, as shown Figure 1.32 using decimal numbers and in Figure 1.33 using BCD notation.

			6	8	7	5
		×)			4	6
Partial product 1 (pp1)		4	1	2	5	0
Partial product 1 (pp2)	2	7	5	0	0	
Product	3	1	6	2	5	0

Figure 1.32 Example of radix 10 multiplication using decimal numbers.

Each digit in the multiplicand is multiplied by each multiplier digit in turn to produce eight subpartial products, one for each multiplicand digit, as shown in Figure 1.33.

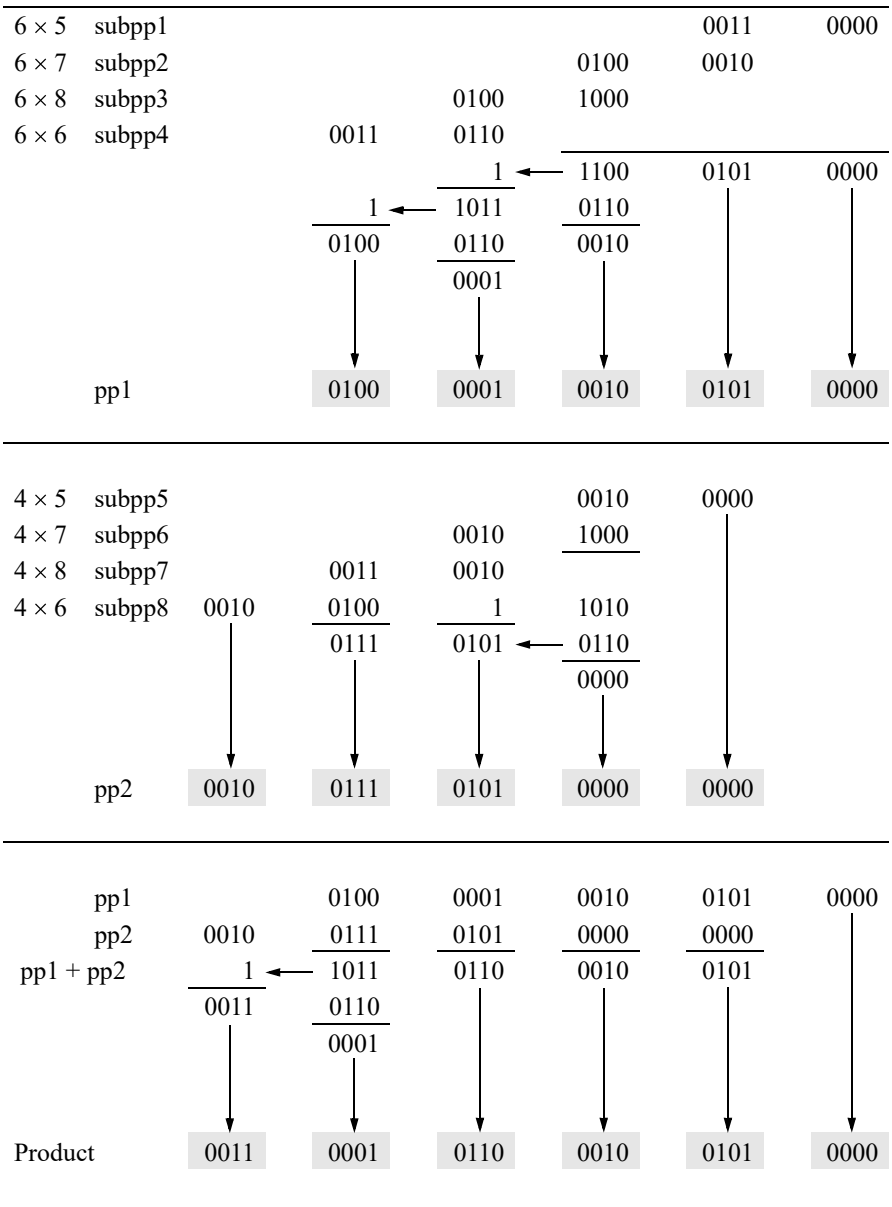


Figure 1.33 Example of decimal multiplication using BCD numbers.

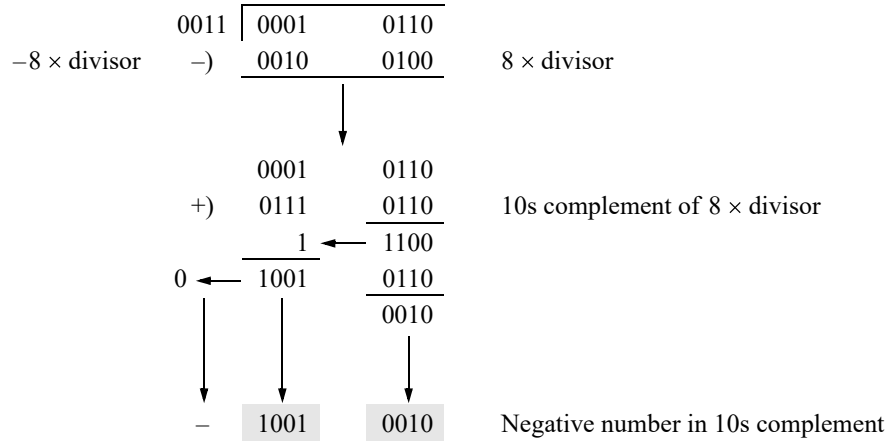
Binary-coded decimal division The method of decimal division presented here is analogous to the binary search method used in programming, which is a systematic way of searching an ordered database. The method begins by examining the middle of

the database. For division, the method adds or subtracts multiples of the divisor or partial remainder. The arithmetic operation is always in the following order:

$-8 \times$ the divisor
$\pm 4 \times$ the divisor
$\pm 2 \times$ the divisor
$\pm 1 \times$ the divisor

This method requires only four cycles for each quotient digit, regardless of the number of quotient digits. The first operation is $-8 \times$ the divisor. If the result is less than zero, then $4 \times$ the divisor is added to the partial remainder; if the result is greater than or equal to zero, then 8 is added to a quotient counter and $4 \times$ the divisor is subtracted from the partial remainder. The process repeats for $\pm 4 \times$ the divisor, $\pm 2 \times$ the divisor, and $\pm 1 \times$ the divisor. Whenever +8, +4, +2, or +1 is added to the quotient counter, the sum of the corresponding additions is the quotient digit.

Whenever a partial remainder is negative, the next version of the divisor is added to the partial remainder; whenever a partial remainder is positive, the next version of the divisor is subtracted from the partial remainder. The example shown in Figure 1.34 illustrates this technique, where $16 \div 3$. The positive versions of the divisor are $+4 \times$ the divisor and $+1 \times$ the divisor; therefore, the quotient is 5. The remainder is the result of the last cycle, in this case 1.



(Continued on next page)

Figure 1.34 Example of BCD division.

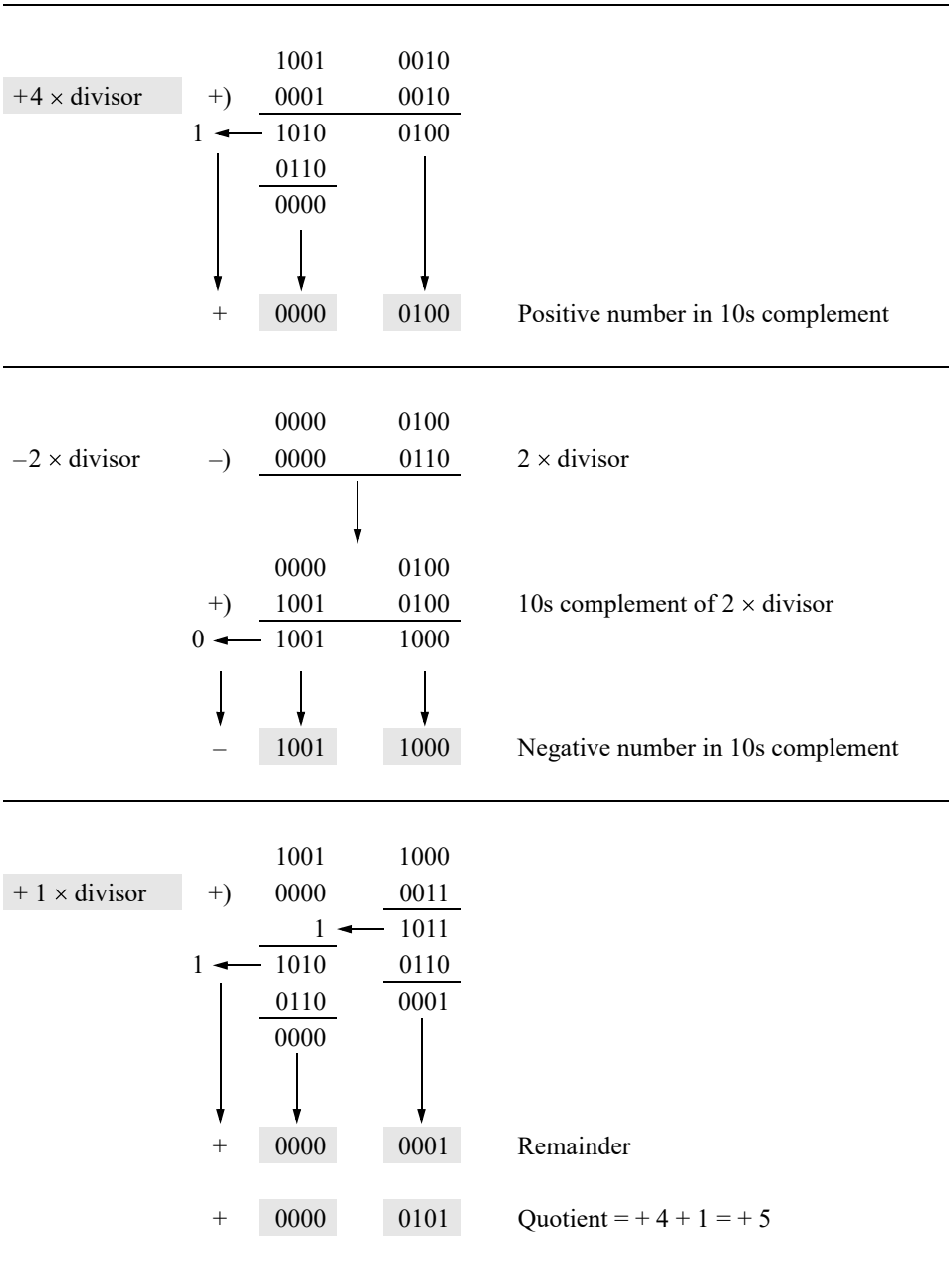


Figure 1.34 (Continued)

1.3 Binary Codes

There are many types of binary codes, many of which will be covered in the following sections. A *code* is simply a set of *n*-bit vectors that can represent numerical values, alphanumerical values, or some other type of coded information. An unsigned vector of *n* bits can represent 2^n unique combinations of the *n* bits and has values that range from 0 to $2^n - 1$. For example, a 3-bit vector has a range of 0 to $2^3 - 1$, or 000 to 111, as shown in Table 1.17 with appropriate weights assigned to each bit position. Although the weights assigned to the individual bits of this code are unique, in general any weight can be assigned.

Table 1.17 Three-Bit Binary Code

	Three-Bit Code		
Decimal	2^2	2^1	2^0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

The bits represented by a fixed-length binary sequence are called a *code word*. This section will discuss additional numerical and nonnumerical codes, weighted and nonweighted codes, and error-detecting codes. If a code word has *n* bits, this does not necessarily mean that there are 2^n valid code words in the set, as is evident in the binary-coded decimal code, in which six of the sixteen code words are invalid.

1.3.1 Binary Weighted and Nonweighted Codes

The 8421 binary-coded decimal (BCD) code is a *weighted code*, because each bit of the 4-bit code words is assigned a weight. This code was presented in an earlier section and is used to represent the decimal digits 0 through 9. The decimal digit represented by the code word can be obtained by summing the weights associated with the bit positions containing 1s. Table 1.18 shows the 8421 BCD code and other BCD codes in which different weights are assigned to the bit positions. Note that some codes have multiple bit configurations for the same decimal number.

Table 1.18 Examples of Binary Codes for Decimal Digits

Decimal	8421	7421	2421	84–2–1	Excess-3
0	0000	0000	0000	0000	0011
1	0001	0001	0001	0111	0100
2	0010	0010	0010	0110	0101
3	0011	0011	0011	0101	0110
4	0100	0100	0100	0100	0111
5	0101	0101	1011	1011	1000
6	0110	0110	1100	1010	1001
7	0111	1000	1101	1001	1010
8	1000	1001	1110	1000	1011
9	1001	1010	1111	1111	1100

The excess-3 code is a *nonweighted code* and is obtained by adding three to the 8421 BCD code, as shown in Table 1.18. It is interesting to note that some of the codes of Table 1.18 are self-complementing. The excess-3 code is a *self-complementing code* in which the 1s complement of a code word is identical to the 9s complement of the corresponding 8421 BCD code word in excess-3 notation, as shown below for the decimal number 4. The 2421 BCD code is also self-complementing, as shown below for the decimal number 8.

Excess-3 BCD code	8421 BCD code
4	4
↓	↓
0111	0100
↓ 1s complement	↓ 9s complement
1000	0101
	↓ excess-3

2421 BCD code	8421 BCD code
8	8
↓	↓
1110	1000
↓ 1s complement	↓ 9s complement
0001	

An alternative method of explaining the self-complementing characteristic of the excess-3 code is as follows: The 1s complement of the excess-3 number yields the 9s complement of the same excess-3 number. For example, the decimal number 4 in excess-3 notation is 0111 (7_{10}); the 1s complement of 0111 is 1000, which is the excess-3 code for 5, which is the 9s complement of 4, the original decimal number. Two examples are shown below.

Excess-3 BCD code	
4	
↓	
0111	
↓ 1s complement	
	which is the excess-3 code for 5, which is the 9s complement of 4

Excess-3 BCD code	
8	
↓	
1011	
↓ 1s complement	
	which is the excess-3 code for 1, which is the 9s complement of 8

Addition in excess-3 Adding in excess-3 is relatively easy. Add the excess-3 numbers using the rules for binary addition, then subtract $(n - 1) \times 3$ from the result, where n is the number of decimal digits, as shown below. Alternatively, convert the decimal numbers to BCD, add the numbers using the rules for BCD arithmetic, then convert the result to excess-3, as shown below.

6	=	1001	
8	=	1011	
4	=	0111	
9	=	1100	
27	10	0111	(39 in excess-3)
– 9	00	1001	
		1110	(30 in excess-3, which is 27 in decimal)

	54	=	0101		0100	
+)	38	=	0011		1000	
	92		1	←	1100	
			1001		0110	
					0010	
			↓		↓	Excess-3

Subtraction in excess-3 Subtraction in excess-3 is similar to subtracting in 1s complement. Add the 1s complement of the excess-3 subtrahend to the minuend using the rules for 1s complement arithmetic. The result will be a binary number in 1s complement, as shown below for two cases. Then convert the result to excess-3.

	7	=	1010		
–)	3	=	0110		
	4		↓		
			1010		
		+)	1001		1s complement of subtrahend
	1	←	0011		
			1		End-around carry
			0100		Binary number in 1s complement (+4)
			↓		Excess-3
					Binary number in excess-3 (7)

	6	=	1001		
–)	9	=	1100		
	–3		↓		
			1001		
		+)	0011		1s complement of subtrahend
			1100		Binary number in 1s complement (–3)
			↓		Excess-3
					Binary number in excess-3 (0)

1.3.2 Binary-to-BCD Conversion

Converting from binary to BCD is accomplished by multiplying by the BCD number by two repeatedly. Multiplying by two is accomplished by a left shift of one bit position followed by an adjustment, if necessary. For example, a left shift of BCD 1001 (9_{10}) results in 1 0010 which is 18 in binary, but only 12 in BCD. Adding six to the low-order BCD digit results in 1 1000, which is the required value of 18.

Instead of adding six after the shift, the same result can be obtained by adding three before the shift since a left shift multiplies any number by two, as shown below. BCD digits in the range 0–4 do not require an adjustment after being shifted left, because the shifted number will be in the range 0–8, which can be contained in a 4-bit BCD digit. However, if the number to be shifted is in the range 5–9, then an adjustment will be required after the left shift, because the shifted number will be in the range 10–18, which requires two BCD digits. Therefore, three is added to the digit prior to being shifted left 1-bit position.

	1001	
+)	0011	
	1100	Shift left 1 yields $0001\ 1000_{BCD}$

Figure 1.35 shows the procedure for converting from binary 1 0101 1011_2 (347_{10}) to BCD. Since there are 9 bits in the binary number; therefore, nine left-shift operations are required, yielding the resulting BCD number of $0011\ 0100\ 0111_{BCD}$.

	10^2	10^1	10^0	Binary
347_{10}	0000	0000	0000	101011011
Shift left 1	0000	0000	0001	01011011
No addition	0000	0000	0001	
Shift left 1	0000	0000	0010	1011011
No addition	0000	0000	0010	
Shift left 1	0000	0000	0101	011011
Add 3	0000	0000	1000	
Shift left 1	0000	0001	0000	11011
No addition	0000	0001	0000	
Shift left 1	0000	0010	0001	1011
No addition	0000	0010	0001	

(Continued on next page)

Figure 1.35 Example of binary-to-BCD conversion.

	10^2	10^1	10^0	Binary
Shift left 1	0000	0100	0011	011
No addition	0000	0100	0011	
Shift left 1	0000	1000	0110	11
Add 3	0000	1011	1001	
Shift left 1	0001	0111	0011	1
Add 3	0001	1010	0011	
Shift left 1	0011	0100	0111	$= 347_{10}$

Figure 1.35 (Continued)

1.3.3 BCD-to-Binary Conversion

Conversion from BCD to binary is accomplished by dividing by two repeatedly; that is, by a right shift operation. The low-order bit that is shifted out is the remainder and becomes the low-order binary bit. The BCD digits may require an adjustment after the right shift operation.

An example of converting 22_{10} to binary is shown in Figure 1.36. After the first shift-right-1 operation, the low-order bit position of the tens decade is a 1, indicating a value of 10_{10}. After the second shift-right-1 operation, the resulting value in the units decade should be 5_{10}; however, the value is 8_{10}. Therefore, a value of 3 must be subtracted to correct the BCD number in the units position. In general, if the high-order bit in any decade is a 1 after the shift-right operation, then a value of 3 must be subtracted from the decade.

	10^1	10^0	Binary
22_{10}	0010	0010	
Shift right 1	0001	0001	0
Shift right 1	0000	1000	10
Subtract 3	0000	0101	
	0000	0101	10_2

Figure 1.36 Example of BCD-to-binary conversion.

A more extensive example is shown in Figure 1.37, which converts 479_{10} to binary. The conversion process is complete when all of the BCD decades contain zeroes. The binary equivalent of 479_{10} is 111011111_2.

	10^2	10^1	10^0	Binary
479_{10}	0100	0111	1001	
Shift right 1	0010	0011	1100	1
Subtract 3	0010	0011	1001	
Shift right 1	0001	0001	1100	11
Subtract 3	0001	0001	1001	
Shift right 1	0000	1000	1100	111
Subtract 3	0000	0101	1001	
Shift right 1	0000	0010	1100	1111
Subtract 3	0000	0010	1001	
Shift right 1	0000	0001	0100	11111
No subtraction	0000	0001	0100	
Shift right 1	0000	0000	1010	011111
Subtract 3	0000	0000	0111	
Shift right 1	0000	0000	0011	1011111
No subtraction	0000	0000	0011	
Shift right 1	0000	0000	0001	11011111
No subtraction	0000	0000	0001	
Shift right 1	0000	0000	0000	111011111
No subtraction	0000	0000	0000	

Figure 1.37 Example of BCD-to-binary conversion.

1.3.4 Gray Code

The Gray code is an nonweighted code that has the characteristic whereby only one bit changes between adjacent code words. The Gray code belongs to a class of cyclic codes called *reflective codes*, as can be seen in Table 1.19. Notice in the first four rows, that y_4 reflects across the reflecting axis; that is, y_4 in rows 2 and 3 is the mirror image of y_4 in rows 0 and 1. In the same manner, y_3 and y_4 reflect across the reflecting axis drawn under row 3. Thus, rows 4 through 7 reflect the state of rows 0 through 3 for y_3 and y_4. The same is true for y_2, y_3, and y_4 relative to rows 8 through 15 and rows 0 through 7.

Table 1.19 Binary 8421 Code and the Gray Code

	(Binary Code $b_3\ b_2\ b_1\ b_0$)				(Gray Code $g_3\ g_2\ g_1\ g_0$)				
Row	y_1	y_2	y_3	y_4	y_1	y_2	y_3	y_4	
0	0	0	0	0	0	0	0	0	
1	0	0	0	1	0	0	0	1	
2	0	0	1	0	0	0	1	1	← y_4 is reflected
3	0	0	1	1	0	0	1	0	
4	0	1	0	0	0	1	1	0	← y3 and y_4
5	0	1	0	1	0	1	1	1	are reflected
6	0	1	1	0	0	1	0	1	
7	0	1	1	1	0	1	0	0	
8	1	0	0	0	1	1	0	0	← y_2, y_3, and y_4
9	1	0	0	1	1	1	0	1	are reflected
10	1	0	1	0	1	1	1	1	
11	1	0	1	1	1	1	1	0	
12	1	1	0	0	1	0	1	0	
13	1	1	0	1	1	0	1	1	
14	1	1	1	0	1	0	0	1	
15	1	1	1	1	1	0	0	0	

Binary-to-Gray code conversion A procedure for converting from the binary 8421 code to the Gray code can be formulated. Let an n-bit binary code word be represented as

$$b_{n-1}\ b_{n-2}\ \cdots\ b_1\ b_0$$

and an n-bit Gray code word be represented as

$$g_{n-1}\ g_{n-2}\ \cdots\ g_1\ g_0$$

where b_0 and g_0 are the low-order bits of the binary and Gray codes, respectively. The ith Gray code bit g_i can be obtained from the corresponding binary code word by the following algorithm:

$$\begin{aligned} g_{n-1} &= b_{n-1} \\ g_i &= b_i \oplus b_{i+1} \end{aligned} \tag{1.22}$$

for $0 \le i \le n-2$, where the symbol $\oplus$ denotes modulo-2 addition defined as:

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

For example, using the algorithm of Equation 1.22, the 4-bit binary code word $b_3\ b_2\ b_1\ b_0 = 1010$ translates to the 4-bit Gray code word $g_3\ g_2\ g_1\ g_0 = 1111$ as follows:

$$\begin{aligned} g_3 &= b_3 &&= 1 \\ g_2 &= b_2 \oplus b_3 = 0 \oplus 1 &&= 1 \\ g_1 &= b_1 \oplus b_2 = 1 \oplus 0 &&= 1 \\ g_0 &= b_0 \oplus b_1 = 0 \oplus 1 &&= 1 \end{aligned}$$

Gray-to-binary code conversion The reverse algorithm to convert from the Gray code to the binary 8421 code is shown in Equation 1.23, where an n-bit binary code word is represented as

$$b_{n-1}\ b_{n-2}\ \cdots\ b_1\ b_0$$

and an n-bit Gray code word is represented as

$$g_{n-1}\ g_{n-2}\ \cdots\ g_1\ g_0$$

where b_0 and g_0 are the low-order bits of the binary and Gray codes, respectively.

$$\begin{aligned} b_{n-1} &= g_{n-1} \\ b_i &= b_{i+1} \oplus g_i \end{aligned} \tag{1.23}$$

For example, using the algorithm of Equation 1.23, the 4-bit Gray code word $g_3\ g_2\ g_1\ g_0 = 1001$ translates to the 4-bit binary code word $b_3\ b_2\ b_1\ b_0 = 1110$ as follows:

$$b_3 = g_3 = 1$$
$$b_2 = b_3 \oplus g_2 = 1 \oplus 0 = 1$$
$$b_1 = b_2 \oplus g_1 = 1 \oplus 0 = 1$$
$$b_0 = b_1 \oplus g_0 = 1 \oplus 1 = 0$$

1.4 Error Detection and Correction Codes

Transferring data within a computer or between computers is subject to error, either permanent or transient. Permanent errors can be caused by hardware malfunctions; transient errors can be caused by transmission errors due to noise. In either case, the data error must at least be detected and preferably corrected. The following error detecting codes and error correcting codes will be briefly discussed: parity, Hamming code, cyclic redundancy check (CRC) code, checksum, and two-out-of-five code.

1.4.1 Parity

An extra bit can be added to a message to make the overall parity of the code word either odd or even; that is, the number of 1s in the code word — message bits plus parity bit — will be either odd or even, as shown in Table 1.20 for both odd and even parity.

Table 1.20 Parity Bit Generation

Message	Parity Bit (odd)	Message	Parity Bit (even)
0000	1	0000	0
0001	0	0001	1
0010	0	0010	1
0011	1	0011	0
0100	0	0100	1
0101	1	0101	0
0110	1	0110	0
0111	0	0111	1
1000	0	1000	1

(Continued on next page)

Table 1.20 Parity Bit Generation

Message	Parity Bit (odd)	Message	Parity Bit (even)
1001	1	1001	0
1010	1	1010	0
1011	0	1011	1
1100	1	1100	0
1101	0	1101	1
1110	0	1110	1
1111	1	1111	0

The parity bit to maintain even parity for a 4-bit message $m_3\, m_2\, m_1\, m_0$ can be generated by modulo-2 addition as previously defined and shown in Equation 1.24. The parity bit for odd parity generation is the complement of Equation 1.24.

$$\text{Parity bit (even)} = m_3 \oplus m_2 \oplus m_1 \oplus m_0 \tag{1.24}$$

Parity implementation can detect an odd number of errors, but cannot correct the errors, because the bits in error cannot be determined. If a single error occurred, then an incorrect code word would be generated and the error would be detected. If two errors occurred, then parity would be unchanged and still correct. As can be seen in Table 1.20, every adjacent pair of code words — message plus parity — have a minimum distance of two; that is, they differ in two bit positions. This means that two bits must change to still maintain a correct code word. The distance-2 words shown below are adjacent and have odd parity, because there are an odd number of 1s.

Bit position	7	6	5	4	3	2	1	0	P
Distance 2 →	1	0	1	0	1	0	1	0	1
→	1	0	1	0	1	0	1	1	0

When a word is to be transmitted, the parity bit is generated (PG) and the nine bits are transmitted along the 9-bit parallel bus, as shown in Figure 1.38. At the receiving end, the parity of the word is checked (PC) and the parity bit is removed. The parity generator and parity checker are both implemented using modulo-2 addition.

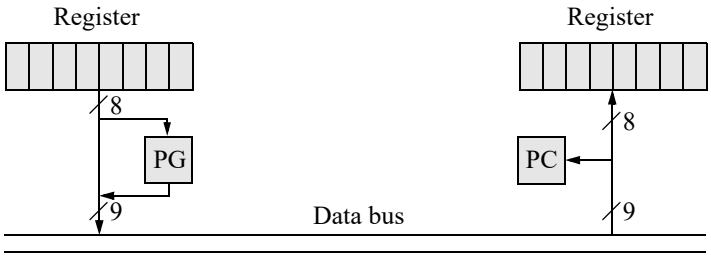


Figure 1.38 General architecture showing parity generation (PG) and parity checking (PC).

1.4.2 Hamming Code

This section provides a brief introduction to the Hamming code error detection and correction technique. An in-depth coverage is presented in a later chapter and includes matrix algebra from which the Hamming code is derived. A common error detection technique is to add a parity bit to the message being transmitted, as shown in the previous section. Thus, single-bit errors — and an odd number of errors — can be detected in the received message. The error bits, however, cannot be corrected because their location in the message is unknown. For example, assume that the message shown below was transmitted using odd parity.

Bit position	7	6	5	4	3	2	1	0	Parity bit
Message	0	1	1	0	0	1	0	0	0

Assume that the message shown below is the received message. The parity of the message is even; therefore, the received message has an error. However, the location of the bit (or bits) in error is unknown. Therefore, in this example, bit 5 cannot be corrected.

Bit position	7	6	5	4	3	2	1	0	Parity bit
Message	0	1	0	0	0	1	0	0	0

Richard W. Hamming developed a code in 1950 that resolves this problem. The *Hamming code* can be considered as an extension of the parity code presented in the previous section, because multiple parity bits provide parity for subsets of the message bits. The subsets overlap such that each message bit is contained in at least two

subsets. The basic Hamming code can detect single or double errors and can correct a single error.

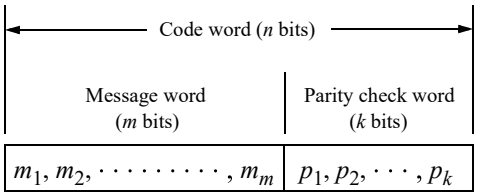
Assume a 2-element code as shown below; that is, there are only two code words in the code. If a single error occurred in code word X, then the assumed message is $X = 000$. Similarly, if a single error occurred in code word Y, then the assumed message is $Y = 111$. Therefore, detection and correction is possible, because the code words *differ in three bit positions*.

Code word $X =$	0 0 0
	0 0 1
	0 1 0
	1 0 0

Code word $Y =$	1 1 1
	0 1 1
	1 0 1
	1 1 0

A *code word* contains n bits consisting of m message bits plus k parity check bits as shown in Figure 1.39. The m bits represent the information or message part of the code word; the k bits are used for detecting and correcting errors, where $k = n - m$. The *Hamming distance* of two code words X and Y is the number of bits in which the two words differ in their corresponding columns.

For example, the Hamming distance is three for code words X and Y as shown in Figure 1.40. Since the minimum distance is three, single error detection and correction is possible. A later chapter will provide a technique to detect single and double errors and to correct single errors.



Code word $X = x_1, x_2, \cdots\cdots\cdots, x_m, \quad x_{m+1}, \cdots, x_n$

Figure 1.39 Code word of n bits containing m message bits and k parity check bits.

$X =$	1	0	1	1	1	1	0	1
$Y =$	1	0	1	0	1	1	1	0

Figure 1.40 Two code words to illustrate a Hamming distance of three.

1.4.3 Cyclic Redundancy Check Code

A class of codes has been developed specifically for serial data transfer called cyclic redundancy check (CRC) codes. This section will provide a brief introduction to the CRC codes. A more detailed description of CRC codes will be presented in a later chapter. Cyclic redundancy check codes can detect both single-bit errors and multiple-bit errors and are especially useful for large strings of serial binary data found on single-track storage devices, such as disk drives. They are also used in serial data transmission networks and in 9-track magnetic tape systems, where each track is treated as a serial bit stream.

The generation of a CRC character uses modulo-2 addition, which is a linear operation; therefore, a linear feedback shift register is used in its implementation. The CRC character that is generated is placed at or near the end of the message.

A possible track format for a disk drive is shown in Figure 1.41, which has separate address and data fields. There is a CRC character for each of the two fields; the CRC character in the address field checks the cylinder, head, and sector addresses; the CRC character in the data field checks the data stream.

The address field and data field both have a *preamble* and a *postamble*. The preamble consists of fifteen 0s followed by a single 1 and is used to synchronize the clock to the data and to differentiate between 0s and 1s. The postamble consists of sixteen 0s and is used to separate the address and data fields.

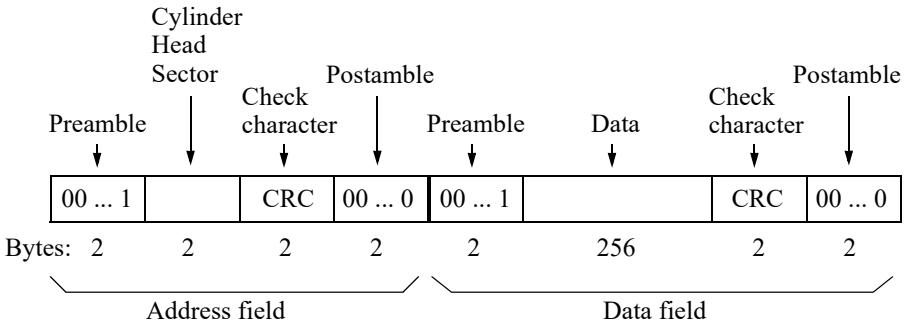


Figure 1.41 Possible track format for a disk drive.

The synchronous data link control (SDLC) — a subset of the high-level data link control (HDLC) — was developed by IBM for their System Network Architecture. It manages synchronous transmission of serial data over a communication channel. The format for the SDLC protocol is illustrated in Figure 1.42, which shows the various fields of a frame.

The first field is the flag field, which is a delimiter indicating the beginning of block. This is followed by an address field, which is decoded by all receivers, then a control field, indicating frame number, last frame, and other control information. The

text field contains *n* bits of serial data followed by a 16-bit CRC character. The final field is a delimiter indicating end of block.

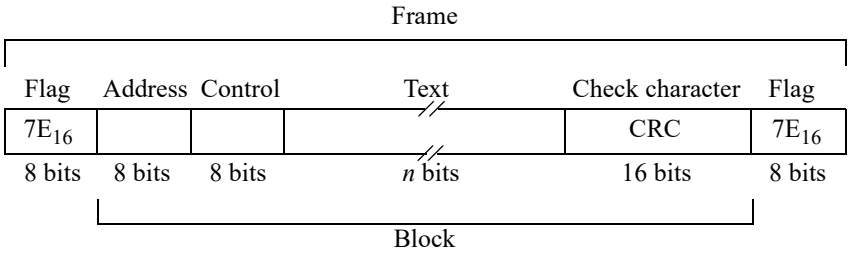


Figure 1.42 One frame of the synchronous data link control format.

Since there may be flags embedded in the text as data, the transmitter inserts a 0 bit after every five consecutive 1s. The receiver discards the 0 bit after every five consecutive 1s; thus, no flag is transmitted randomly. Figure 1.43 shows two examples of 0s inserted after five consecutive 1s. Note that there is no ambiguity between a flag as data with an inserted 0 or a data bit stream of five 1s followed by a 0.

Text	0	1	1	1	1	1	1	0
Sent as	0	1	1	1	1	1	0	1	0
Text	0	1	1	1	1	1	0	
Sent as	0	1	1	1	1	1	0	0	

Figure 1.43 Examples of 0s inserted after every five consecutive 1s so that a flag may not be transmitted as data for the SDLC protocol.

1.4.4 Checksum

The *checksum* is the sum derived from the application of an algorithm that is calculated before and after transmission to ensure that the data is free from errors. The checksum character is a numerical value that is based on the number of asserted bits in the message and is appended to the end of the message. The receiving unit then applies the same algorithm to the message and compares the results with the appended checksum character.

There are many versions of checksum algorithms. The parity checking method presented in Section 1.41 is a modulo-2 addition of all bits in the string of data. The modulo-2 sum is 1 if the number of 1s in the 8-bit byte is odd, otherwise, the modulo-2 sum is 0.

If the information consists of n bytes of data, then a simple checksum algorithm is to perform modulo-256 addition on the bytes in the message. The sum thus obtained is the checksum byte and is appended to the last byte creating a message of $n + 1$ bytes. The receiving unit then regenerates the checksum by obtaining the sum of the first n bytes and compares that sum to byte $n + 1$. This method can detect a single byte error.

Alternatively, the sum that is obtained by modulo-256 addition in the transmitting unit is 2s complemented and becomes the checksum character which is appended to the end of the transmitted message. The receiving unit uses the same algorithm and adds the recalculated uncomplemented checksum character to the transmitted checksum character, resulting in a sum of zero if the message had no errors. An example of this algorithm is shown in Figure 1.44 for a 4-byte message.

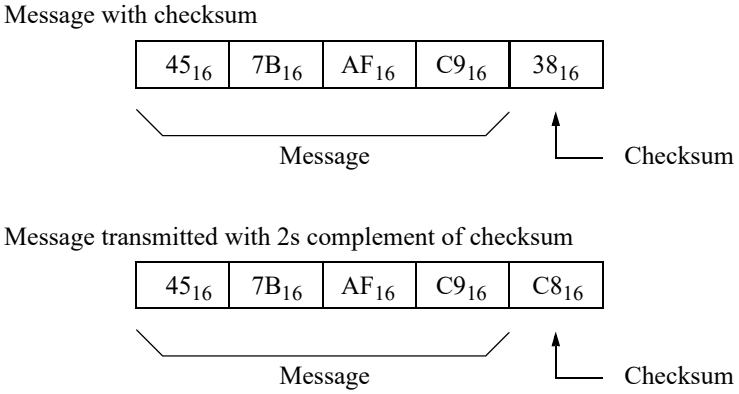


Figure 1.44 Checksum generated for a 4-byte message.

Memories can also include a checksum in their implementation. Consider a small memory as shown in Figure 1.45, containing only eight bytes of data. The checksum is generated by using modulo-2 addition on the columns of each byte.

1	1	0	0	1	0	1	0	Data
0	0	1	1	0	1	1	0	
0	1	1	1	1	0	0	0	
0	1	0	0	1	1	1	1	
0	0	1	1	0	1	1	1	
0	0	0	0	0	0	1	1	
0	1	1	0	1	1	0	0	
0	0	0	0	0	0	0	0	
1	0	0	1	0	0	1	1	Checksum

Figure 1.45 Checksum generation for memories.

1.4.5 Two-Out-Of-Five Code

The two-out-of-five code is 5-bit nonweighted code that is characterized by having exactly two 1s and three 0s in any code word. This code has a minimum distance of two and is relatively easy to provide error detection by counting the number of 1s in a code word.

An error is detected whenever the number of 1s in a code word is not equal to two. This can result from a change of one or more bits which cause the total number of 1s to differ from two. However, an error will be undetected if there are two simultaneous bit changes which result in a valid code word with two 1s. For example, if code word 01100 (7) were changed during transmission to 01010 (5), then the error would be undetected. The two-out-of-five code is representative of *m*-out-of-*n* codes. Table 1.21 shows a typical two-out-of-five code for the decimal digits 0 through 9.

Table 1.21 Two-out-of-Five Code for the Decimal Digits

Decimal	Two-Out-of-Five Code
0	00011
1	00101
2	01001
3	10001
4	00110
5	01010
6	10010
7	01100
8	10100
9	11000

1.4.6 Horizontal and Vertical Parity Check

For single-error correction and double-error detection, Hamming code is the ideal method for detecting and correcting errors in memory operations. An alternative method to Hamming code for smaller memories — which has less redundancy — is a technique using horizontal and vertical parity. Horizontal parity utilizes an odd parity bit for each word in memory. Vertical parity is the modulo-2 addition of identical bit positions of each word in a block of memory.

Errors are detected by the horizontal parity, which indicates the word in error. This error is used in conjunction with the vertical parity to detect and correct single-bit errors. Figure 1.46 shows a 10-word block of memory, which will be used to illustrate the technique of single-bit error detection and correction using horizontal and vertical parity.

Word	7	6	5	4	3	2	1	0	Horizontal parity (odd)
0	1	1	1	0	1	0	0	0	1
1	0	1	0	1	0	1	0	1	1
2	0	1	1	0	1	1	1	0	0
3	1	1	0	0	1	1	0	1	0
4	0	1	1	1	1	0	1	0	0
5	1	0	0	1	1	0	0	1	1
6	1	0	1	0	0	0	1	0	0
7	0	0	1	1	0	1	1	0	1
8	0	0	0	1	0	0	1	1	0
9	1	0	1	1	0	1	1	1	1
Vertical parity (odd)	0	0	1	1	0	0	1	0	

Figure 1.46 Ten-word memory to illustrate single-error detection and correction using horizontal and vertical parity.

Assume that word four has an error in bit position four such that word four changes from 0111 1010 0 to 0110 1010 0, which is incorrect for odd parity. To correct the bit in error, the current vertical parity word 0011 0010 is exclusive-ORed with the new vertical parity word obtained by the modulo-2 addition of identical bit positions of each word in a block of memory with the exception of word four. This will generate the original word four, as shown below.

	Current vertical parity word	0	0	1	1	0	0	1	0
$\oplus$	New vertical parity word without word four	0	1	0	0	1	0	0	0
	Original word four	0	1	1	1	1	0	1	0

As another example, assume that word seven has an error in bit position two such that word seven changes from 0011 0110 1 to 0011 0010 1, which is incorrect for odd parity. Using the procedure outlined above, the correct word seven is obtained, as shown below.

	Current vertical parity word	0	0	1	1	0	0	1	0
$\oplus$	New vertical parity word without word seven	0	0	0	0	0	1	0	0
	Original word seven	0	0	1	1	0	1	1	0

1.5 Serial Data Transmission

The standard binary code for alphanumeric characters is the American Standard Code for Information Interchange (ASCII). The code uses seven bits to encode 128 characters, which include 26 uppercase letters, 26 lowercase letters, the digits 0 through 9, and 32 special characters, such as &, #, and $. The bits are labeled b_6 through b_0, where b_0 is the low-order bit and is transmitted first. There is also a parity bit which maintains either odd or even parity for the eight bits.

There are two basic types of serial communication: synchronous and asynchronous. In *synchronous* communication, information is transferred using a *self-clocking* scheme; that is, the clock is synchronized to the data, which determines the rate of transfer. Self-clocking is presented in detail in a later chapter. Alternatively, there may be a separate clock signal to determine the bit cell boundaries. Information is normally sent as blocks of data, not as individual bytes, and may contain error checking.

In *asynchronous* communication, the timing occurs for each character. The communications line for asynchronous serial data transfer is in an idle state (a logic 1 level) when characters are not being transmitted. Figure 1.47 shows the format for asynchronous serial data transmission for the ASCII character *W*. The receiver and transmitter must operate at the same clock frequency. The *start bit* causes a transition from a logic 1 level to a logic 0 level and synchronizes the receiver with the transmitter. This synchronization is for one character only — the next character will contain a new start bit. The start bit signals the receiver to start assembling a character.

The start bit is followed by the seven data bits, which in turn are followed by the parity bit (odd parity is shown). The parity bit is followed by two *stop bits* that are asserted high, which ensure that the start bit of the next character will cause a transition from high to low. The stop bits are also used to isolate consecutive characters.

There is a wide range of standard transmission rates, referred to as bauds. A *baud* is a unit of signaling speed, which refers to the number of times the state of a signal changes per second; that is, bits per second.

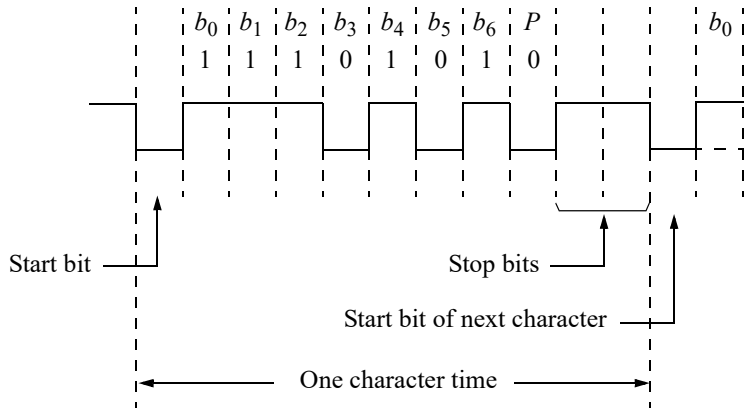


Figure 1.47 Asynchronous serial data transfer for the ASCII character *W*.

1.6 Problems

1.1 Convert the unsigned binary number 0111 1101_2 to radix 10.

1.2 Convert the octal number 5476_8 to radix 10.

1.3 Convert the hexadecimal number $4AF9_{16}$ to radix 10.

1.4 Convert the unsigned binary number 1100.110_2 to radix 10.

1.5 Convert the octal number 173.25_8 to radix 10.

1.6 Convert the hexadecimal number $6BC.5_{16}$ to radix 10.

1.7 Convert the hexadecimal number $4A3CB_{16}$ to binary and octal.

1.8 Represent the following numbers in sign magnitude, diminished-radix complement, and radix complement for radix 2 and radix 10: +136 and –136.

1.9 Represent the decimal numbers +54, –28, +127, and –13 in sign magnitude, diminished-radix complement, and radix complement for radix 2 using 8 bits.

1.10 Convert the following octal numbers to hexadecimal: 6536_8 and 63457_8.

1.11 Convert the binary number 0100 1101.1011_2 to decimal and hexadecimal notation.

1.12 Convert 7654_8 to radix 3.

1.13 Determine the range of positive and negative numbers in radix 2 for sign magnitude, diminished-radix complement, and radix complement.

1.14 Add the unsigned binary numbers 1101 0110_2 and 0111 1100_2.

1.15 Multiply the unsigned binary numbers 1111_2 and 0011_2.

1.16 Divide the following unsigned binary numbers: 101 $1110_2 \div 1001_2$.

1.17 Convert the following radix –4 number to radix 3 with three fraction digits: 123.13_{-4}.

1.18 Convert 263.2_7 to radix 4 with three fraction digits.

1.19 Obtain the radix complement of $F8B6_{16}$.

1.20 Obtain the radix complement of 54320_6.

1.21 The numbers shown below are in sign-magnitude representation. Convert the numbers to 2s complement representation for radix 2 with the same numerical value using 8 bits.

1001 1001
0001 0000
1011 1111

1.22 The numbers shown below are in 2s complement representation for radix 2. Determine whether the indicated arithmetic operations produce an overflow.

0011 0110 + 1110 0011
1001 1000 – 0010 0010
0011 0110 – 1110 0011

1.23 Convert the following unsigned binary numbers to radix 10:

(a) 10111_2
(b) 11111_2
(c) 1111000.101_2
(d) 1000001.111_2

1.24 Use direct subtraction to obtain the difference of the following unsigned binary numbers:

$11010_2 - 10111_2$

1.25 Perform the following binary subtraction using the diminished-radix complement method:

$101111_2 - 000011_2$

1.26 Convert the following decimal numbers to signed binary numbers in 2s complement representation using 8 bits.

(a) $+56_{10}$ (b) -27_{10}

1.27 Add the following binary numbers:

$111111_2 + 111111_2 + 111111_2 + 111111_2$

1.28 Add the following BCD numbers:

$1001\ 1000_{BCD} + 1001\ 0111_{BCD}$

1.29 Obtain the sum of the following binary numbers:

$1111\ 1111_2$
$1111\ 1111_2$
$1111\ 1111_2$
$1111\ 1111_2$
$1111\ 1111_2$
$1111\ 1111_2$

1.30 Obtain the sum of the following radix 3 numbers:

$1111_3 + 1111_3 + 1111_3 + 1111_3$

1.31 Obtain the following hexadecimal numbers:

$FFFF_{16} + FFFF_{16} + FFFF_{16} + FFFF_{16}$

1.32 Add the following radix 4 numbers:

3213_4
1010_4
0213_4
3010_4
2101_4

1.33 Perform a subtraction on the operands shown below, which are in radix complementation for radix 3.

$02021_3 - 22100_3$

1.34 Convert the radix 5 number 2434.1_5 to radix 9 with a precision of one fraction digit.

1.35 Multiply the two binary numbers shown below, which are in radix complementation, using the paper-and-pencil method.

$11111_2 \times 01011_2$

1.36 Let A and B be two binary numbers in 2s complement representation as shown below, where A' and B' are the 1s complement of A and B, respectively. Perform the operation listed below.

$A = 1011\ 0001_2$
$B = 1110\ 0100_2$

$(A' + 1) - (B' + 1)$

1.37 The decimal operands shown below are to be added using decimal BCD addition. Obtain the intermediate sum; that is, the sum before adjustment is applied.

$725_{10} + 536_{10}$

1.38 Use the paper-and-pencil method for binary restoring division to perform the following divide operation using an 8-bit dividend and a 4-bit divisor:

$73_{10} \div 5_{10}$

1.39 Convert the following binary number to BCD using the shift left method:

10101110_2

1.40 Convert the following decimal number from BCD to binary using the shift right method:

363_{10}