3

Digital Arithmetic

Having discussed different methods of numeric and alphanumeric data representation in the first two chapters, the next obvious step is to study the rules of data manipulation. Two types of operation that are performed on binary data include arithmetic and logic operations. Basic arithmetic operations include addition, subtraction, multiplication and division. AND, OR and NOT are the basic logic functions. While the rules of arithmetic operations are covered in the present chapter, those related to logic operations will be discussed in the next chapter.

3.1 Basic Rules of Binary Addition and Subtraction

The basic principles of binary addition and subtraction are similar to what we all know so well in the case of the decimal number system. In the case of addition, adding '0' to a certain digit produces the same digit as the sum, and, when we add '1' to a certain digit or number in the decimal number system, the result is the next higher digit or number, as the case may be. For example, 6 + 1 in decimal equals '7' because '7' immediately follows '6' in the decimal number system. Also, 7 + 1 in octal equals '10' as, in the octal number system, the next adjacent higher number after '7' is '10'. Similarly, 9 + 1 in the hexadecimal number system is 'A'. With this background, we can write the basic rules of binary addition as follows:

1. $0 + 0 = 0$.
2. $0 + 1 = 1$.
3. $1 + 0 = 1$.
4. $1 + 1 = 0$ with a carry of '1' to the next more significant bit.
5. $1 + 1 + 1 = 1$ with a carry of '1' to the next more significant bit.

Table 3.1 summarizes the sum and carry outputs of all possible three-bit combinations. We have taken three-bit combinations as, in all practical situations involving the addition of two larger bit

Table 3.1 Binary addition of three bits.

A	B	Carry-in (C_{in})	Sum	Carry-out (C_o)	A	B	Carry-in (C_{in})	Sum	Carry-out (C_o)
0	0	0	0	0	1	0	0	1	0
0	0	1	1	0	1	0	1	0	1
0	1	0	1	0	1	1	0	0	1
0	1	1	0	1	1	1	1	1	1

numbers, we need to add three bits at a time. Two of the three bits are the bits that are part of the two binary numbers to be added, and the third bit is the carry-in from the next less significant bit column.

The basic principles of binary subtraction include the following:

1. $0 - 0 = 0$.
2. $1 - 0 = 1$.
3. $1 - 1 = 0$.
4. $0 - 1 = 1$ with a borrow of 1 from the next more significant bit.

The above-mentioned rules can also be explained by recalling rules for subtracting decimal numbers. Subtracting ‘0’ from any digit or number leaves the digit or number unchanged. This explains the first two rules. Subtracting ‘1’ from any digit or number in decimal produces the immediately preceding digit or number as the answer. In general, the subtraction operation of larger-bit binary numbers also involves three bits, including the two bits involved in the subtraction, called the minuend (the upper bit) and the subtrahend (the lower bit), and the borrow-in. The subtraction operation produces the difference output and borrow-out, if any. Table 3.2 summarizes the binary subtraction operation. The entries in Table 3.2 can be explained by recalling the basic rules of binary subtraction mentioned above, and that the subtraction operation involving three bits, that is, the minuend (A), the subtrahend (B) and the borrow-in (B_{in}), produces a difference output equal to ($A - B - B_{in}$). It may be mentioned here that, in the case of subtraction of larger-bit binary numbers, the least significant bit column always involves two bits to produce a difference output bit and the borrow-out

Table 3.2 Binary subtraction.

Inputs			Outputs	
Minuend (A)	Subtrahend (B)	Borrow-in (B_{in})	Difference (D)	Borrow-out (B_o)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

bit. The borrow-out bit produced here becomes the borrow-in bit for the next more significant bit column, and the process continues until we reach the most significant bit column. The addition and subtraction of larger-bit binary numbers is illustrated with the help of examples in sections 3.2 and 3.3 respectively.

3.2 Addition of Larger-Bit Binary Numbers

The addition of larger binary integers, fractions or mixed binary numbers is performed columnwise in just the same way as in the case of decimal numbers. In the case of binary numbers, however, we follow the basic rules of addition of two or three binary digits, as outlined earlier. The process of adding two larger-bit binary numbers can be best illustrated with the help of an example.

Consider two generalized four-bit binary numbers $(A_3 A_2 A_1 A_0)$ and $(B_3 B_2 B_1 B_0)$, with A_0 and B_0 representing the LSB and A_3 and B_3 representing the MSB of the two numbers. The addition of these two numbers is performed as follows. We begin with the LSB position. We add the LSB bits and record the sum S_0 below these bits in the same column and take the carry C_0, if any, to the next column of bits. For instance, if $A_0 = 1$ and $B_0 = 0$, then $S_0 = 1$ and $C_0 = 0$. Next we add the bits A_1 and B_1 and the carry C_0 from the previous addition. The process continues until we reach the MSB bits. The four steps are shown ahead. C_0, C_1, C_2 and C_3 are carrys, if any, produced as a result of adding first, second, third and fourth column bits respectively, starting from LSB and proceeding towards MSB. A similar procedure is followed when the given numbers have both integer as well as fractional parts:

$$
1.\quad
\begin{array}{cccc}
 & & (C_0) & \\
A_3 & A_2 & A_1 & A_0 \\
B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & S_0 \\
\hline
\end{array}
\qquad
2.\quad
\begin{array}{ccccc}
 & & (C_1) & (C_0) & \\
 & A_3 & A_2 & A_1 & A_0 \\
 & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & S_1 & S_0 \\
\hline
\end{array}
$$

$$
3.\quad
\begin{array}{cccc}
(C_2) & (C_1) & (C_0) & \\
A_3 & A_2 & A_1 & A_0 \\
B_3 & B_2 & B_1 & B_0 \\
\hline
 & S_2 & S_1 & S_0 \\
\hline
\end{array}
\qquad
4.\quad
\begin{array}{ccccc}
 & (C_2) & (C_1) & (C_0) & \\
 & A_3 & A_2 & A_1 & A_0 \\
 & B_3 & B_2 & B_1 & B_0 \\
\hline
C_3 & S_3 & S_2 & S_1 & S_0 \\
\hline
\end{array}
$$

3.2.1 Addition Using the 2's Complement Method

The 2's complement is the most commonly used code for processing positive and negative binary numbers. It forms the basis of arithmetic circuits in modern computers. When the decimal numbers to be added are expressed in 2's complement form, the addition of these numbers, following the basic laws of binary addition, gives correct results. Final carry obtained, if any, while adding MSBs should be disregarded. To illustrate this, we will consider the following four different cases:

1. Both the numbers are positive.
2. Larger of the two numbers is positive.
3. The larger of the two numbers is negative.
4. Both the numbers are negative.

Case 1

- Consider the decimal numbers +37 and +18.
- The 2's complement of +37 in eight-bit representation = 00100101.
- The 2's complement of +18 in eight-bit representation = 00010010.
- The addition of the two numbers, that is, +37 and +18, is performed as follows

$$\begin{array}{r} 00100101 \\ +\,\underline{00010010} \\ \underline{00110111} \end{array}$$

- The decimal equivalent of $(00110111)_2$ is (+55), which is the correct answer.

Case 2

- Consider the two decimal numbers +37 and -18.
- The 2's complement representation of +37 in eight-bit representation = 00100101.
- The 2's complement representation of −18 in eight-bit representation = 11101110.
- The addition of the two numbers, that is, +37 and −18, is performed as follows:

$$\begin{array}{r} 00100101 \\ +\,\underline{11101110} \\ \underline{00010011} \end{array}$$

- The final carry has been disregarded.
- The decimal equivalent of $(00010011)_2$ is +19, which is the correct answer.

Case 3

- Consider the two decimal numbers +18 and −37.
- −37 in 2's complement form in eight−bit representation = 11011011.
- +18 in 2's complement form in eight−bit representation = 00010010.
- The addition of the two numbers, that is, −37 and +18, is performed as follows:

$$\begin{array}{r} 11011011 \\ +\,\underline{00010010} \\ \underline{11101101} \end{array}$$

- The decimal equivalent of $(11101101)_2$, which is in 2's complement form, is −19, which is the correct answer. 2's complement representation was discussed in detail in Chapter 1 on number systems.

Case 4

- Consider the two decimal numbers −18 and −37.
- −18 in 2's complement form is 11101110.
- −37 in 2's complement form is 11011011.
- The addition of the two numbers, that is, −37 and −18, is performed as follows:

$$\begin{array}{r} 11011011 \\ +\,\underline{11101110} \\ \underline{11001001} \end{array}$$

- The final carry in the ninth bit position is disregarded.
- The decimal equivalent of $(11001001)_2$, which is in 2's complement form, is -55, which is the correct answer.

It may also be mentioned here that, in general, 2's complement notation can be used to perform addition when the expected result of addition lies in the range from -2^{n-1} to $+(2^{n-1}-1)$, n being the number of bits used to represent the numbers. As an example, eight-bit 2's complement arithmetic cannot be used to perform addition if the result of addition lies outside the range from -128 to $+127$. Different steps to be followed to do addition in 2's complement arithmetic are summarized as follows:

1. Represent the two numbers to be added in 2's complement form.
2. Do the addition using basic rules of binary addition.
3. Disregard the final carry, if any.
4. The result of addition is in 2's complement form.

Example 3.1

Perform the following addition operations:

1. *$(275.75)_{10} + (37.875)_{10}$.*
2. *$(AF1.B3)_{16} + (FFF.E)_{16}$.*

Solution

1. As a first step, the two given decimal numbers will be converted into their equivalent binary numbers (decimal-to-binary conversion has been covered at length in Chapter 1, and therefore the decimal-to-binary conversion details will not be given here):

   $$(275.75)_{10} = (100010011.11)_2 \text{ and } (37.875)_{10} = (100101.111)_2$$

   The two binary numbers can be rewritten as $(100010011.110)_2$ and $(000100101.111)_2$ to have the same number of bits in their integer and fractional parts. The addition of two numbers is performed as follows:

   $$\begin{array}{r} 100010011.110 \\ \underline{000100101.111} \\ \underline{100111001.101} \end{array}$$

   The decimal equivalent of $(100111001.101)_2$ is $(313.625)_{10}$.

2. $(\mathrm{AF1.B3})_{16} = (101011110001.10110011)_2$ and $(\mathrm{FFF.E})_{16} = (111111111111.1110)_2$. $(111111111111.1110)_2$ can also be written as $(111111111111.11100000)_2$ to have the same number of bits in the integer and fractional parts. The two numbers can now be added as follows:

$$\begin{array}{r} 0101011110001.10110011 \\ \underline{0111111111111.11100000} \\ \underline{1101011110001.10010011} \end{array}$$

The hexadecimal equivalent of $(1101011110001.10010011)_2$ is $(\mathrm{1AF1.93})_{16}$, which is equal to the hex addition of $(\mathrm{AF1.B3})_{16}$ and $(\mathrm{FFF.E})_{16}$.

Example 3.2

Find out whether 16-bit 2's complement arithmetic can be used to add 14 276 and 18 490.

Solution

The addition of decimal numbers 14 276 and 18 490 would yield 32 766. 16-bit 2's complement arithmetic has a range of -2^{15} to $+(2^{15}-1)$, i.e. $-32\,768$ to $+32\,767$. The expected result is inside the allowable range. Therefore, 16-bit arithmetic can be used to add the given numbers.

Example 3.3

Add −118 and −32 firstly using eight-bit 2's complement arithmetic and then using 16-bit 2's complement arithmetic. Comment on the results.

Solution

- −118 in eight-bit 2's complement representation = 10001010.
- −32 in eight-bit 2's complement representation = 11100000.
- The addition of the two numbers, after disregarding the final carry in the ninth bit position, is 01101010. Now, the decimal equivalent of $(01101010)_2$, which is in 2's complement form, is +106. The reason for the wrong result is that the expected result, i.e. −150, lies outside the range of eight-bit 2's complement arithmetic. Eight-bit 2's complement arithmetic can be used when the expected result lies in the range from -2^7 to $+(2^7-1)$, i.e. −128 to +127. −118 in 16-bit 2's complement representation = 1111111110001010.
- −32 in 16-bit 2's complement representation = 1111111111100000.
- The addition of the two numbers, after disregarding the final carry in the 17th position, produces 1111111101101010. The decimal equivalent of $(1111111101101010)_2$, which is in 2's complement form, is −150, which is the correct answer. 16-bit 2's complement arithmetic has produced the correct result, as the expected result lies within the range of 16-bit 2's complement notation.

3.3 Subtraction of Larger-Bit Binary Numbers

Subtraction is also done columnwise in the same way as in the case of the decimal number system. In the first step, we subtract the LSBs and subsequently proceed towards the MSB. Wherever the subtrahend (the bit to be subtracted) is larger than the minuend, we borrow from the next adjacent higher bit position having a ‘1’. As an example, let us go through different steps of subtracting $(1001)_2$ from $(1100)_2$.

In this case, ‘1’ is borrowed from the second MSB position, leaving a ‘0’ in that position. The borrow is first brought to the third MSB position to make it ‘10’. Out of ‘10’ in this position, ‘1’ is taken to the LSB position to make ‘10’ there, leaving a ‘1’ in the third MSB position. $10 - 1$ in the LSB column gives ‘1’, $1 - 0$ in the third MSB column gives ‘1’, $0 - 0$ in the second MSB column gives ‘0’ and $1 - 1$ in the MSB also gives ‘0’ to complete subtraction. Subtraction of mixed numbers is also done in the same manner. The above-mentioned steps are summarized as follows:

$$\begin{array}{lcccc}
\mathbf{1.} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\
 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\
\hline
 & & & & \mathbf{1} \\
\hline
\end{array}
\qquad
\begin{array}{lcccc}
\mathbf{2.} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\
 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\
\hline
 & & & \mathbf{1} & \mathbf{1} \\
\hline
\end{array}$$

$$\begin{array}{lcccc}
\mathbf{3.} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\
 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\
\hline
 & & \mathbf{0} & \mathbf{1} & \mathbf{1} \\
\hline
\end{array}
\qquad
\begin{array}{lcccc}
\mathbf{4.} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\
 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\
\hline
 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\
\hline
\end{array}$$

3.3.1 Subtraction Using 2’s Complement Arithmetic

Subtraction is similar to addition. Adding 2’s complement of the subtrahend to the minuend and disregarding the carry, if any, achieves subtraction. The process is illustrated by considering six different cases:

1. Both minuend and subtrahend are positive. The subtrahend is the smaller of the two.
2. Both minuend and subtrahend are positive. The subtrahend is the larger of the two.
3. The minuend is positive. The subtrahend is negative and smaller in magnitude.
4. The minuend is positive. The subtrahend is negative and greater in magnitude.
5. Both minuend and subtrahend are negative. The minuend is the smaller of the two.
6. Both minuend and subtrahend are negative. The minuend is the larger of the two.

Case 1

- Let us subtract +14 from +24.
- The 2’s complement representation of $+24 = 00011000$.
- The 2’s complement representation of $+14 = 00001110$.
- Now, the 2’s complement of the subtrahend (i.e. +14) is 11110010.
- Therefore, $+24 - (+14)$ is given by

$$\begin{array}{r}
00011000 \\
+\ \underline{11110010} \\
\underline{00001010}
\end{array}$$

  with the final carry disregarded.
- The decimal equivalent of $(00001010)_2$ is +10, which is the correct answer.

Case 2

- Let us subtract +24 from +14.
- The 2's complement representation of +14 = 00001110.
- The 2's complement representation of +24 = 00011000.
- The 2's complement of the subtrahend (i.e. +24) = 11101000.
- Therefore, +14 − (+24) is given by

$$\begin{array}{r} 00001110 \\ +\,11101000 \\ \hline \underline{11110110} \end{array}$$

- The decimal equivalent of $(11110110)_2$, which is of course in 2's complement form, is −10 which is the correct answer.

Case 3

- Let us subtract −14 from +24.
- The 2's complement representation of +24 = 00011000 = minuend.
- The 2's complement representation of −14 = 11110010 = subtrahend.
- The 2's complement of the subtrahend (i.e. −14) = 00001110.
- Therefore, +24 − (−14) is performed as follows:

$$\begin{array}{r} 00011000 \\ +\,00001110 \\ \hline \underline{00100110} \end{array}$$

- The decimal equivalent of $(00100110)_2$ is +38, which is the correct answer.

Case 4

- Let us subtract −24 from +14.
- The 2's complement representation of +14 = 00001110 = minuend.
- The 2's complement representation of −24 = 11101000 = subtrahend.
- The 2's complement of the subtrahend (i.e. −24) = 00011000.
- Therefore, +14 − (−24) is performed as follows:

$$\begin{array}{r} 00001110 \\ +\,00011000 \\ \hline \underline{00100110} \end{array}$$

- The decimal equivalent of $(00100110)_2$ is +38, which is the correct answer.

Case 5

- Let us subtract −14 from −24.
- The 2's complement representation of −24 = 11101000 = minuend.

- The 2's complement representation of −14=11110010 = subtrahend.
- The 2's complement of the subtrahend = 00001110.
- Therefore, −24 − (−14) is given as follows:

$$\begin{array}{r} 11101000 \\ +\,00001110 \\ \hline \underline{11110110} \end{array}$$

- The decimal equivalent of $(11110110)_2$, which is in 2's complement form, is −10, which is the correct answer.

Case 6

- Let us subtract −24 from −14.
- The 2's complement representation of −14 = 11110010 = minuend.
- The 2's complement representation of −24=11101000 = subtrahend.
- The 2's complement of the subtrahend = 00011000.
- Therefore, −14 − (−24) is given as follows:

$$\begin{array}{r} 11110010 \\ +\,00011000 \\ \hline \underline{00001010} \end{array}$$

  with the final carry disregarded.
- The decimal equivalent of $(00001010)_2$, which is in 2's complement form, is +10, which is the correct answer.

It may be mentioned that, in 2's complement arithmetic, the answer is also in 2's complement notation, only with the MSB indicating the sign and the remaining bits indicating the magnitude. In 2's complement notation, positive magnitudes are represented in the same way as the straight binary numbers, while the negative magnitudes are represented as the 2's complement of their straight binary counterparts. A '0' in the MSB position indicates a positive sign, while a '1' in the MSB position indicates a negative sign.

The different steps to be followed to do subtraction in 2's complement arithmetic are summarized as follows:

1. Represent the minuend and subtrahend in 2's complement form.
2. Find the 2's complement of the subtrahend.
3. Add the 2's complement of the subtrahend to the minuend.
4. Disregard the final carry, if any.
5. The result is in 2's complement form.
6. 2's complement notation can be used to perform subtraction when the expected result of subtraction lies in the range from -2^{n-1} to $+(2^{n-1}-1)$, n being the number of bits used to represent the numbers.

Example 3.4

Subtract $(1110.011)_2$ from $(11011.11)_2$ using basic rules of binary subtraction and verify the result by showing equivalent decimal subtraction.

Solution

The minuend and subtrahend are first modified to have the same number of bits in the integer and fractional parts. The modified minuend and subtrahend are $(11011.110)_2$ and $(01110.011)_2$ respectively:

$$\begin{array}{r} 11011.110 \\ -\,01110.011 \\ \hline 01101.011 \\ \hline \end{array}$$

The decimal equivalents of $(11011.110)_2$ and $(01110.011)_2$ are 27.75 and 14.375 respectively. Their difference is 13.375, which is the decimal equivalent of $(01101.011)_2$.

Example 3.5

Subtract (a) $(-64)_{10}$ from $(+32)_{10}$ and (b) $(29.A)_{16}$ from $(4F.B)_{16}$. Use 2's complement arithmetic.

Solution:

(a) $(+32)_{10}$in 2's complement notation = $(00100000)_2$.
$(-64)_{10}$ in 2's complement notation = $(11000000)_2$.
The 2's complement of $(-64)_{10}$ = $(01000000)_2$.
$(+32)_{10} - (-64)_{10}$ is determined by adding the 2's complement of $(-64)_{10}$ to $(+32)_{10}$.
Therefore, the addition of $(00100000)_2$ to $(01000000)_2$ should give the result. The operation is shown as follows:

$$\begin{array}{r} 00100000 \\ +\,01000000 \\ \hline 01100000 \\ \hline \end{array}$$

The decimal equivalent of $(01100000)_2$ is +96, which is the correct answer as $+32 - (-64) = +96$.

(b) The minuend = $(4F.B)_{16}$ = $(01001111.1011)_2$.
The minuend in 2's complement notation = $(01001111.1011)_2$.
The subtrahend = $(29.A)_{16}$ = $(00101001.1010)_2$.
The subtrahend in 2's complement notation = $(00101001.1010)_2$.
The 2's complement of the subtrahend = $(11010110.0110)_2$.
$(4F.B)_{16} - (29.A)_{16}$ is given by the addition of the 2's complement of the subtrahend to the minuend.

$$\begin{array}{r} 01001111.1011 \\ +\,11010110.0110 \\ \hline 00100110.0001 \\ \hline \end{array}$$

with the final carry disregarded. The result is also in 2's complement form. Since the result is a positive number, 2's complement notation is the same as it would be in the case of the straight binary code.
The hex equivalent of the resulting binary number = $(26.1)_{16}$, which is the correct answer.

3.4 BCD Addition and Subtraction in Excess-3 Code

Below, we will see how the excess-3 code can be used to perform addition and subtraction operations on BCD numbers.

3.4.1 Addition

The excess-3 code can be very effectively used to perform the addition of BCD numbers. The steps to be followed for excess-3 addition of BCD numbers are as follows:

1. The given BCD numbers are written in excess-3 form by adding '0011' to each of the four-bit groups.
2. The two numbers are then added using the basic laws of binary addition.
3. Add '0011' to all those four-bit groups that produce a carry, and subtract '0011' from all those four-bit groups that do not produce a carry during addition.
4. The result thus obtained is in excess-3 form.

3.4.2 Subtraction

Subtraction of BCD numbers using the excess-3 code is similar to the addition process discussed above. The steps to be followed for excess-3 substraction of BCD numbers are as follows:

1. Express both minuend and subtrahend in excess-3 code.
2. Perform subtraction following the basic laws of binary subtraction.
3. Subtract '0011' from each invalid BCD four-bit group in the answer.
4. Subtract '0011' from each BCD four-bit group in the answer if the subtraction operation of the relevant four-bit groups required a borrow from the next higher adjacent four-bit group.
5. Add '0011' to the remaining four-bit groups, if any, in the result.
6. This gives the result in excess-3 code.

The process of addition and subtraction can be best illustrated with the help of following examples.

Example 3.6

Add $(0011\ 0101\ 0110)_{BCD}$ and $(0101\ 0111\ 1001)_{BCD}$ using the excess-3 addition method and verify the result using equivalent decimal addition.

Solution

The excess-3 equivalents of 0011 0101 0110 and 0101 0111 1001 are 0110 1000 1001 and 1000 1010 1100 respectively. The addition of the two excess-3 numbers is given as follows:

```
0110 1000 1001
1000 1010 1100
--------------
1111 0011 0101
--------------
```

After adding 0011 to the groups that produced a carry and subtracting 0011 from the groups that did not produce a carry, we obtain the result of the above addition as 1100 0110 1000. Therefore, 1100

0110 1000 represents the excess-3 code for the true result. The result in BCD code is 1001 0011 0101, which is the BCD equivalent of 935. This is the correct answer as the addition of the given BCD numbers 0011 0101 0110 = $(356)_{10}$ and 0101 0111 1001 = $(579)_{10}$ yields $(935)_{10}$ only.

Example 3.7

Perform $(185)_{10} - (8)_{10}$ using the excess-3 code.

Solution

- $(185)_{10} = (0001\ 1000\ 0101)_{BCD}$.The excess-3 equivalent of $(0001\ 1000\ 0101)_{BCD}$ = 0100 1011 1000.
- $(8)_{10} = (008)_{10} = (0000\ 0000\ 1000)_{BCD}$. The excess-3 equivalent of $(0000\ 0000\ 1000)_{BCD}$ = 0011 0011 1011.
- Subtraction is performed as follows:

$$\begin{array}{r} 0100\ 1011\ 1000 \\ -\ 0011\ 0011\ 1011 \\ \hline 0001\ 0111\ 1101 \\ \hline \end{array}$$

- In the subtraction operation, the least significant column of four-bit groups needed a borrow, while the other two columns did not need any borrow. Also, the least significant column has produced an invalid BCD code group. Subtracting 0011 from the result of this column and adding 0011 to the results of other two columns, we get 0100 1010 1010. This now constitutes the result of subtraction expressed in excess-3 code.
- The result in BCD code is therefore 0001 0111 0111.
- The decimal equivalent of 0001 0111 0111 is 177, which is the correct result.

3.5 Binary Multiplication

The basic rules of binary multiplication are governed by the way an AND gate functions when the two bits to be multiplied are fed as inputs to the gate. Logic gates are discussed in detail in the next chapter. As of now, it would suffice to say that the result of multiplying two bits is the same as the output of the AND gate with the two bits applied as inputs to the gate. The basic rules of multiplication are listed as follows:

1. $0 \times 0 = 0$.
2. $0 \times 1 = 0$.
3. $1 \times 0 = 0$.
4. $1 \times 1 = 1$.

One of the methods for multiplication of larger-bit binary numbers is similar to what we are familiar with in the case of decimal numbers. This is called the 'repeated left-shift and add' algorithm. Microprocessors and microcomputers, however, use what is known as the 'repeated add and right-shift' algorithm to do binary multiplication as it is comparatively much more convenient to implement than the 'repeated left-shift and add' algorithm. The two algorithms are briefly described below. Also, binary multiplication of mixed binary numbers is done by performing multiplication without considering the binary point. Starting from the LSB, the binary point is then placed after *n* bits, where *n* is equal to the sum of the number of bits in the fractional parts of the multiplicand and multiplier.

3.5.1 Repeated Left-Shift and Add Algorithm

In the ‘repeated left-shift and add’ method of binary multiplication, the end-product is the sum of several partial products, with the number of partial products being equal to the number of bits in the multiplier binary number. This is similar to the case of decimal multiplication. Each successive partial product after the first is shifted one digit to the left with respect to the immediately preceding partial product. In the case of binary multiplication too, the first partial product is obtained by multiplying the multiplicand binary number by the LSB of the multiplier binary number. The second partial product is obtained by multiplying the multiplicand binary number by the next adjacent higher bit in the multiplier binary number and so on. We begin with the LSB of the multiplier to obtain the first partial product. If the LSB is a ‘1’, a copy of the multiplicand forms the partial product, and it is an all ‘0’ sequence if the LSB is a ‘0’. We proceed towards the MSB of the multiplier and obtain various partial products. The second partial product is shifted one bit position to the left relative to the first partial product; the third partial product is shifted one bit position to the left relative to the second partial product and so on. The addition of all partial products gives the final answer. If the multiplicand and multiplier have different signs, the end result has a negative sign, otherwise it is positive. The procedure is further illustrated by showing $(23)_{10} \times (6)_{10}$ multiplication.

$$\begin{array}{rrl}
 & 1\,0\,1\,1\,1 & \\
\textit{Multiplicand}: & \times\,1\,1\,0 & \ldots\ldots\ldots\ldots\, (23)_{10} \\
\textit{Multiplier}: & \overline{} & \ldots\ldots\ldots\ldots\, (6)_{10} \\
 & 0\,0\,0\,0\,0 & \\
 & 1\,0\,1\,1\,1 & \\
 & 1\,0\,1\,1\,1 & \\
 & \overline{1\,0\,0\,0\,1\,0\,1\,0} &
\end{array}$$

The decimal equivalent of $(10001010)_2$ is $(138)_{10}$, which is the correct result.

3.5.2 Repeated Add and Right-Shift Algorithm

The multiplication process starts with writing an all ‘0’ bit sequence, with the number of bits equal to the number of bits in the multiplicand. This bit sequence (all ‘0’ sequence) is added to another same-sized bit sequence, which is the same as the multiplicand if the LSB of the multiplier is a ‘1’, and an all ‘0’ sequence if it is a ‘0’. The result of the first addition is shifted one bit position to the right, and the bit shifted out is recorded. The vacant MSB position is replaced by a ‘0’. This new sequence is added to another sequence, which is an all ‘0’ sequence if the next adjacent higher bit in the multiplier is a ‘0’, and the same as the multiplicand if it is a ‘1’. The result of the second addition is also shifted one bit position to the right, and a new sequence is obtained. The process continues until all multiplier bits are exhausted. The result of the last addition together with the recorded bits constitutes the result of multiplication. We will illustrate the procedure by doing $(23)_{10} \times (6)_{10}$ multiplication again, this time by using the ‘repeated add and right-shift’ algorithm:

- The multiplicand $= (23)_{10} = (10111)_2$ and the multiplier $= (6)_{10} = (110)_2$. The multiplication process is shown in Table 3.3.
- Therefore, $(10111)_2 \times (110)_2 = (10001010)_2$.

Table 3.3 Multiplication using the repeated add and right-shift algorithm.

1 0 1 1 1 1 1 0	Multiplicand Multiplier
0 0 0 0 0 + 0 0 0 0 0	Start
0 0 0 0 0	Result of first addition
0 0 0 0 0 + 1 0 1 1 1	0 (Result of addition shifted one bit to right)
1 0 1 1 1	Result of second addition
0 1 0 1 1 + 1 0 1 1 1	10 (Result of addition shifted one bit to right)
1 0 0 0 1 0	Result of third addition
0 1 0 0 0 1	010 (Result of addition shifted one bit to right)

Example 3.8

Multiply (a) $(100.01)_2 \times (10.1)_2$ *by using the 'repeated add and left-shift' algorithm and (b)* $(2B)_{16} \times (3)_{16}$ *by using the 'add and right-shift' algorithm. Verify the results by showing equivalent decimal multiplication.*

Solution

(a) As a first step, we will multiply $(10001)_2$ by $(101)_2$. The process is shown as follows:

```
    1 0 0 0 1
      × 1 0 1
  -----------
    1 0 0 0 1
  0 0 0 0 0
1 0 0 0 1
  -----------
1 0 1 0 1 0 1
```

The multiplication result is then given by placing the binary point three bits after the LSB, which gives $(1010.101)_2$ as the final result. Also, $(100.01)_2 = (4.25)_{10}$ and $(10.1)_2 = (2.5)_{10}$. Moreover, $(4.25)_{10} \times (2.5)_{10} = (10.625)_{10}$ and $(1010.101)_2$ equals $(10.625)_{10}$, which verifies the result.

(b) $(2B)_{16} = 00101011 = 101011$ and $(3)_{16} = 0011 = 11$.
Different steps involved in the multiplication process are shown in Table 3.4.
The result of multiplication is therefore $(10000001)_2$. Also, $(2B)_{16} = (43)_{10}$ and $(3)_{16} = (3)_{10}$. Therefore, $(2B)_{16} \times (3)_{16} = (129)_{10}$. Moreover, $(10000001)_2 = (129)_{10}$, which verifies the result.

3.6 Binary Division

While binary multiplication is the process of repeated addition, binary division is the process of repeated subtraction. Binary division can be performed by using either the 'repeated right-shift and

Table 3.4 Example 3.8.

1 0 1 0 1 1	Multiplicand
1 1	Multiplier
0 0 0 0 0 0	Start
+ 1 0 1 0 1 1	
1 0 1 0 1 1	Result of first addition
0 1 0 1 0 1	1 (Result of addition shifted one bit to right)
+ 1 0 1 0 1 1	
1 0 0 0 0 0 0	Result of second addition
0 1 0 0 0 0 0	01 (Result of addition shifted one bit to right)

subtract' or the 'repeated subtract and left-shift' algorithm. These are briefly described and suitably illustrated in the following sections.

3.6.1 Repeated Right-Shift and Subtract Algorithm

The algorithm is similar to the case of conventional division with decimal numbers. At the outset, starting from MSB, we begin with the number of bits in the dividend equal to the number of bits in the divisor and check whether the divisor is smaller or greater than the selected number of bits in the dividend. If it happens to be greater, we record a '0' in the quotient column. If it is smaller, we subtract the divisor from the dividend bits and record a '1' in the quotient column. If it is greater and we have already recorded a '0', then, as a second step, we include the next adjacent bit in the dividend bits, shift the divisor to the right by one bit position and again make a similar check like the one made in the first step. If it is smaller and we have made the subtraction, then in the second step we append the next MSB of the dividend to the remainder, shift the divisor one bit to the right and again make a similar check. The options are again the same. The process continues until we have exhausted all the bits in the dividend. We will illustrate the algorithm with the help of an example. Let us consider the division of $(100110)_2$ by $(1100)_2$. The sequence of operations needed to carry out the above division is shown in Table 3.5. The quotient = 011 and the remainder = 10.

Table 3.5 Binary division using the repeated right-shift and subtract algorithm.

	Quotient		
First step	0	1 0 0 1 1 0	Dividend
		−1 1 0 0	Divisor
Second step	1	1 0 0 1 1	First five MSBs of dividend
		−1 1 0 0	Divisor shifted to right
		0 1 1 1	First subtraction remainder
Third step	1	0 1 1 1 0	Next MSB appended
		−1 1 0 0	Divisor right shifted
		0 0 1 0	Second subtraction remainder

Table 3.6 Binary division using the repeate subtract and left-shift algorithm.

Quotient	1 0 0 1 −1 1 0 0	1 0
0	1 1 0 1 +1 1 0 0	Borrow exists
	1 0 0 1	Final carry ignored
	1 0 0 1 1 −1 1 0 0	Next MSB appended
1	0 1 1 1	No borrow
	0 1 1 1 0 −1 1 0 0	Next MSB appended
1	0 0 0 1 0	No borrow

3.6.2 Repeated Subtract and Left-Shift Algorithm

The procedure can again be best illustrated with the help of an example. Let us consider solving the above problem using this algorithm. The steps needed to perform the division are as follows. We begin with the first four MSBs of the dividend, four because the divisor is four bits long. In the first step, we subtract the divisor from the dividend. If the subtraction requires borrow in the MSB position, enter a '0' in the quotient column; otherwise, enter a '1'. In the present case there exists a borrow in the MSB position, and so there is a '0' in the quotient column. If there is a borrow, the divisor is added to the result of subtraction. In doing so, the final carry, if any, is ignored. The next MSB is appended to the result of the first subtraction if there is no borrow, or to the result of subtraction, restored by adding the divisor, if there is a borrow. By appending the next MSB, the remaining bits of the dividend are one bit position shifted to the left. It is again compared with the divisor, and the process is repeated. It goes on until we have exhausted all the bits of the dividend. The final remainder can be further processed by successively appending 0s and trying subtraction to get fractional part bits of the quotient. The different steps are summarized in Table 3.6. The quotient = 011 and the remainder = 10.

Example 3.9

Use the 'repeated right-shift and subtract' algorithm to divide $(110101)_2$ by $(1011)_2$. Determine both the integer and the fractional parts of the quotient. The fractional part may be determined up to three bit places.

Solution

The sequence of operations is given in Table 3.7. The operations are self-explanatory.

- The quotient = 100.110.
- Now, $(110101)_2 = (53)_{10}$ and $(1011)_2 = (11)_{10}$.
- $(53)_{10}$ divided by $(11)_{10}$ gives $(4.82)_{10}$.
- $(100.110)_2 = (4.75)_{10}$, which matches with the expected result to a good approximation.

Table 3.7 Example 3.9.

	Quotient		
First step	1	1 1 0 1 0 1 −1 0 1 1	Dividend Divisor
		0 0 1 0	First subtraction
Second step	0	0 0 1 0 0 −1 0 1 1	Next MSB appended Divisor right shifted
Third step	0	0 0 1 0 0 1 −1 0 1 1	Next MSB appended Divisor right shifted
		0 0 1 0 0 1	All bits exhausted
	1	0 0 1 0 0 1 0 −1 0 1 1	‘0’ appended Divisor right shifted
		0 1 1 1	Second subtraction
Fourth step	1	0 1 1 1 0 −1 0 1 1	‘0’ appended Divisor right shifted
		0 0 0 1 1	Third subtraction
Fifth step	0	0 0 0 1 1 0 −1 0 1 1	‘0’ appended Divisor right shifted
		0 0 1 1	Fourth subtraction

Example 3.10

Use the ‘repeated subtract and left-shift’ algorithm to divide $(100011)_2$ by $(100)_2$ to determine both the integer and fractional parts of the quotient. Verify the result by showing equivalent decimal division. Determine the fractional part to two bit places.

Solution

The sequence of operations is given in Table 3.8. The operations are self-explanatory.

- The quotient = $(1000.11)_2 = (8.75)_{10}$.
- Now, $(100011)_2 = (35)_{10}$ and $(100)_2 = (4)_{10}$.
- $(35)_{10}$ divided by $(4)_{10}$ gives $(8.75)_{10}$ and hence is verified.

Example 3.11

Divide $(AF)_{16}$ by $(09)_{16}$ using the method of ‘repeated right shift and subtract’, bearing in mind the signs of the given numbers, assuming that we are working in eight-bit 2’s complement arithmetic.

Solution

- The dividend = $(AF)_{16}$.
- As it is a negative hexadecimal number, the magnitude of this number is determined by its 2’s complement (or more precisely by its 16’s complement in hexadecimal number language).

Table 3.8 Example 3.10.

Quotient		
	1 0 0 −1 0 0	0 1 1 Dividend Divisor
1	0 0 0	No borrow
	0 0 0 0 −1 0 0	Next MSB appended
0	1 0 0 +1 0 0	Borrow exists
	0 0 0 0 0 0 1 −1 0 0	Final carry ignored Next MSB appended
0	1 0 1 + 1 0 0	Borrow exists
	0 0 1 0 0 1 1 − 1 0 0	Final carry ignored Next MSB appended
0	1 1 1 +1 0 0	Borrow exists
	0 1 1 0 1 1 0 − 1 0 0	Final carry ignored '0' appended
1	0 1 0	No borrow
	0 1 0 0 −1 0 0	'0' appended
1	0 0 0	No borrow

- The 16's complement of $(AF)_{16} = (51)_{16}$.
- The binary equivalent of $(51)_{16} = 01010001 = 1010001$.
- The divisor $= (09)_{16}$.
- It is a positive number.
- The binary equivalent of $(09)_{16} = 00001001$.
- As the dividend is a negative number and the divisor a positive number, the quotient will be a negative number. The division process using the 'repeated right-shift and subtract' algorithm is given in Table 3.9.
- The quotient $= 1001 = (09)_{16}$.
- As the quotient should be a negative number, its magnitude is given by the 16's complement of $(09)_{16}$, i.e. $(F7)_{16}$.
- Therefore, $(AF)_{16}$ divided by $(09)_{16}$ gives $(F7)_{16}$.

3.7 Floating-Point Arithmetic

Before performing arithmetic operations on floating-point numbers, it is necessary to make a few checks, such as finding the signs of the two mantissas, checking any possible misalignment of exponents, etc.

Table 3.9 Example 3.11

1	1 0 1 0 0 0 1 −1 0 0 1	Divisor less than dividend
	0 0 0 1	
0	0 0 0 1 0 −1 0 0 1	Divisor greater than dividend
0	0 0 0 1 0 0 −1 0 0 1	Divisor still greater
1	0 0 0 1 0 0 1 −1 0 0 1	Divisor less than dividend
	0 0 0 0 0 0 0	

For example, if the exponents of the two numbers are not equal, the addition and subtraction operations necessitate that they be made equal. In that case, the mantissa of the smaller of the two numbers is shifted right, and the exponent is incremented for each shift until the two exponents are equal. Once the binary points are aligned and the exponents made equal, addition and subtraction operations become straightforward. While doing subtraction, of course, a magnitude check is also required to determine the smaller of the two numbers.

3.7.1 Addition and Subtraction

If N_1 and N_2 are two floating-point numbers given by

$$N_1 = m_1 \times 2^e$$

$$N_2 = m_2 \times 2^e$$

then

$$N_1 + N_2 = m_1 \times 2^e + m_2 \times 2^e = (m_1 + m_2) \times 2^e$$

and

$$N_1 - N_2 = m_1 \times 2^e - m_2 \times 2^e = (m_1 - m_2) \times 2^e$$

The subtraction operation assumes that $N_1 > N_2$. Post-normalization of the result may be required after the addition or subtraction operation.

3.7.2 Multiplication and Division

In the case of multiplication of two floating-point numbers, the mantissas of the two numbers are multiplied and their exponents are added. In the case of a division operation, the mantissa of the quotient is given by the division of the two mantissas (i.e. dividend mantissa divided by divisor mantissa) and the exponent of the quotient is given by subtraction of the two exponents (i.e. dividend exponent minus divisor exponent).

If

$$N_1 = m_1 \times 2^{e1} and\ N_2 = m_2 \times 2^{e2}$$

then

$$N_1 \times N_2 = (m_1 \times m_2) \times 2^{(e1+e2)}$$

and

$$N_1/N_2 = (m_1/m_2) \times 2^{(e1-e2)}$$

Again, post-normalization may be required after multiplication or division, as in the case of addition and subtraction operations.

Example 3.12

Add (a) $(39)_{10}$ and $(19)_{10}$ and (b) $(1E)_{16}$ and $(F3)_{16}$ using floating-point numbers. Verify the answers by performing equivalent decimal addition.

Solution

(a) $(39)_{10} = 100111 = 0.100111 \times 2^6$.

$(19)_{10} = 10011 = 0.10011 \times 2^5 = 0.010011 \times 2^6$.

Therefore, $(39)_{10} + (19)_{10} = 0.100111 \times 2^6 + 0.010011 \times 2^6$
$= (0.100111 + 0.010011) \times 2^6 = 0.111010 \times 2^6$
$= 111010 = (58)_{10}$

and hence is verified.

(b) $(1E)_{16} = (00011110)_2 = 0.00011110 \times 2^8$.

$(F3)_{16} = (11110011)_2 = 0.11110011 \times 2^8$.

$(1E)_{16} + (F3)_{16} = (0.00011110 + 0.11110011) \times 2^8 = 100010001$
$= 000100010001$
$= (111)_{16}$.

Also, $(1E)_{16} + (F3)_{16} = (111)_{16}$ and hence is proved.

Example 3.13

Subtract $(17)_8$ from $(21)_8$ using floating-point numbers and verify the answer.

Solution

- $(21)_8 = (010001)_2 = 0.010001 \times 2^6$.
- $(17)_8 = (001111)_2 = 0.001111 \times 2^6$.
- Therefore, $(21)_8 - (17)_8 = (0.010001 - 0.001111) \times 2^6$
  $= 0.000010 \times 2^6 = 000010 = (02)_8$.
- Also, $(21)_8 - (17)_8 = (02)_8$ and hence is verified.

Example 3.14

Multiply $(37)_{10}$ by $(10)_{10}$ using floating-point numbers. Verify by showing equivalent decimal multiplication.

Solution

- The multiplicand = $(37)_{10} = (100101)_2 = 0.100101 \times 2^6$.
- The multiplier = $(10)_{10} = (1010)_2 = 0.1010 \times 2^4$.
- $(37)_{10} \times (10)_{10} = (0.100101 \times 0.1010) \times 2^{10} = 0.0101110010 \times 2^{10} = 101110010$
  $= (370)_{10}$ and hence is verified.

Example 3.15

Perform $(E3B)_{16} \div (1A)_{16}$ using binary floating-point numbers. Verify by showing equivalent decimal division.

Solution

- Dividend = $(\mathrm{E3B})_{16} = (111000111011)_2 = 0.111000111011 \times 2^{12}$.
- Divisor = $(\mathrm{1A})_{16} = (00011010)_2 = (11010)_2 = 0.11010 \times 2^5$.
- Therefore, $(\mathrm{E3B})_{16} \div (\mathrm{1A})_{16} = (0.111000111011 \div 0.11010) \times 2^7$.
- By performing division of the mantissas using either of the two division algorithms described earlier, we obtain the result of division as $(10001100.00011)_2$.
- $(10001100.00011)_2 = (140.093)_{10}$.
- Also, $(\mathrm{E3B})_{16} = (3643)_{10}$ and $(1A)_{16} = (26)_{10}$.
- $(\mathrm{E3B})_{16} \div (\mathrm{1A})_{16} = (3643)_{10} \div (26)_{10} = (140.1)_{10}$, which is the same as the result obtained with binary floating-point arithmetic to a good approximation.

Review Questions

1. Outline the different steps involved in the addition of larger-bit binary numbers for the following two cases:

   (a) The larger of the two numbers is positive and the other number is negative.
   (b) The larger of the two numbers is negative and the other number is positive.

2. Outline the different steps involved in the subtraction of larger-bit binary numbers for the following two cases:

   (a) The minuend is positive. The subtrahend is negative and smaller in magnitude.
   (b) The minuend is positive. The subtrahend is negative and larger in magnitude.

3. What decides whether a particular binary addition or subtraction operation would be possible with 2's complement arithmetic?
4. Why in microprocessors and microcomputers is the 'repeated add and right-shift' algorithm preferred over the 'repeated left-shift and add' algorithm for binary multiplication? Briefly outline the procedure for multiplication in the case of the former.

5. Prove that the largest six-digit hexadecimal number when subtracted from the largest eight-digit octal number yields zero in decimal.

Problems

1. Perform the following operations using 2's complement arithmetic. The numbers are represented using 2's or 10's or 16's complement notation as the case may be. Express the result both in 2's complement binary as well as in decimal.

   (a) $(7F)_{16} + (A1)_{16}$.
   (b) $(110)_{10} + (0111)_2$.

   (a) $(00100000)_2, (32)_{10}$; (b) $(01110101)_2, (117)_{10}$

2. Evaluate the following to two binary places:

   (a) $(100.0001)_2 \div (10.1)_2$.
   (b) $(111001)_2 \div (1001)_2$.
   (c) $(111.001)_2 \times (1.11)_2$.

   (a) 1.10; (b) 110.01; (c) 1100.01

3. Prove that 16-bit 2's complement arithmetic cannot be used to add +18 150 and +14 618, while it can be used to add −18 150 and −14 618.
4. Add the maximum positive integer to the minimum negative integer, both represented in 16-bit 2's complement binary notation. Express the answer in 2's complement binary.

   1111111111111111
5. The result of adding two BCD numbers represented in excess-3 code is 0111 1011 when the two numbers are added using simple binary addition. If one of the numbers is $(12)_{10}$, find the other.

   $(03)_{10}$
6. Perform the following operations using 2's complement arithmetic:

   (a) $(+43)_{10} - (-53)_{10}$.
   (b) $(1ABC)_{16} + (1DEF)_{16}$.
   (c) $(3E91)_{16} - (1F93)_{16}$.

   (a) 01100000; (b) $(38AB)_{16}$; (c) $(1EFE)_{16}$

Further Reading

1. Ercegovac, M. D. and Lang, T. (2003) *Digital Arithmetic*, Morgan Kaufmann Publishers, CA, USA.
2. Tocci, R. J. (2006) *Digital Systems – Principles and Applications*, Prentice-Hall Inc., NJ, USA.
3. Ashmila, E. M., Dlay, S. S. and Hinton, O. R. (2005) 'Adder methodology and design using probabilistic multiple carry estimates'. *IET Computers and Digital Techniques*, **152**(6), pp. 697–703.
4. Lu, M. (2005) *Arithmetic and Logic in Computer Systems,* John Wiley & Sons, Inc., NJ, USA.