Operations on Data

In Chapter 3 we showed how to store different types of data in a computer. In this chapter, we show how to operate on data stored in a computer. Operations on data can be divided into three broad categories: logic operations, shift operations, and arithmetic operations.

Objectives

After studying this chapter, the student should be able to:

- ❑ List the three categories of operations performed on data.
- ❑ Perform unary and binary logic operations on bit patterns.
- ❑ Distinguish between logic shift operations and arithmetic shift operations.
- ❑ Perform logic shift operations on bit patterns.
- ❑ Perform arithmetic shift operations on integers stored in two's complement format.
- ❑ Perform addition and subtraction on integers when they are stored in two's complement format.
- ❑ Perform addition and subtraction on integers when they are stored in sign-and-magnitude format.
- ❑ Perform addition and subtraction operations on reals when they are stored in floating-point format.
- ❑ Understand some applications of logical and shift operations such as setting, unsetting, and flipping specific bits.

4.1 LOGIC OPERATIONS

In Chapter 3 we discussed the fact that data inside a computer is stored as patterns of bits. Logic operations refer to those operations that apply the same basic operation on individual bits of a pattern, or on two corresponding bits in two patterns. This means that we can define logic operations at the bit level and at the pattern level. A logic operation at the pattern level is *n* logic operations, of the same type, at the bit level where *n* is the number of bits in the pattern.

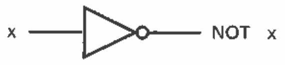
4.1.1 Logic operations at bit level

A bit can take one of the two values: 0 or 1. If we interpret 0 as the value *false* and 1 as the value *true*, we can apply the operations defined in **Boolean algebra** to manipulate bits. Boolean algebra, named in honor of George Boole, belongs to a special field of mathematics called *logic*. Boolean algebra and its application to building logic circuits in computers are briefly discussed in Appendix E. In this section, we show briefly four bit-level operations that are used to manipulate bits: NOT, AND, OR, and XOR.

Boolean algebra and logic circuits are discussed in Appendix E.

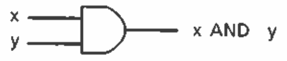
Figure 4.1 shows the symbols for these four bit-level operators and their truth tables. A **truth table** defines the values of output for each possible input or inputs. Note that the output of each operator is always one bit, but the input can be one or two bits.

Figure 4.1 *Logic operations at the bit level*



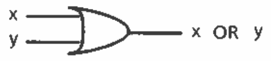
NOT

x	NOT x
0	1
1	0



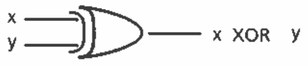
AND

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1



OR

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1



XOR

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

NOT

The NOT **operator** is a unary operator: it takes only one input. The output bit is the complement of the input. If the input is 0, the output is 1, if the input is 1, the output is 0. In other words, the NOT operator flips its input. The truth table of the NOT operator has only two rows because the single input can be either 0 or 1: two possibilities.

AND

The AND **operator** is a binary operator: it takes two inputs. The output bit is 1 if both inputs are 1s and the output is 0 in the other three cases. The truth table of the AND operator has four rows because, with two inputs, there are four possible input combinations.

A property

One interesting point about the AND operator is that if a bit in one input is 0, we do not have to check the corresponding bit in the other input: we can quickly conclude that the result is 0. We use this property when we discuss the application of this operator in relation to a bit pattern.

For $x = 0$ or 1 $\quad x$ AND $0 \rightarrow 0$ $\quad$ and $\quad$ 0 AND $x \rightarrow 0$

OR

The **OR operator** is a also a binary operator: it takes two inputs. The output bit is 0 if both inputs are 0s and the output is 1 in other three cases. The truth table of the OR operator has also four rows. The OR operator is sometimes called the *inclusive-or operator* because the output is 1 not only when one of the inputs is 1, but also when both inputs are 1s. This is in contrast to the operator we introduce next.

A property

One interesting point about the OR operator is that if a bit in one input is 1, we do not have to check the corresponding bit in the other input: we can quickly conclude that the result is 1. We use this property when we discuss the application of this operator in relation to a bit pattern.

For $x = 0$ or 1 $\quad x$ OR $1 \rightarrow 1$ $\quad$ and $\quad$ 1 OR $x \rightarrow 1$

XOR

The **XOR operator** (pronounced 'exclusive-or') is also a binary operator like the OR operator, with only one difference: the output is 0 if both inputs are 1s. We can look at this operator in another way: the output is 0 when both inputs are the same, and the output is 1 when the inputs are different.

Example 4.1

In English we use the conjunction 'or' sometimes to means an inclusive-or, and sometimes to means an exclusive-or.

a. The sentence 'I wish to have a car *or* a house' uses 'or' in the inclusive sense—I wish a car, a house, or both.

b. The sentence 'Today is either Monday or Tuesday' uses 'or' in the exclusive sense—today is either Monday or Tuesday, but it cannot be both.

Example 4.2

The XOR operator is not actually a new operator. We can always simulate it using the other three operators. The following two expressions are equivalent

$$x \text{ XOR } y \leftrightarrow [x \text{ AND } (\text{NOT } y)] \text{ OR } [(\text{NOT } x) \text{ AND } y]$$

The equivalence can be proved if we make the truth table for both.

A property

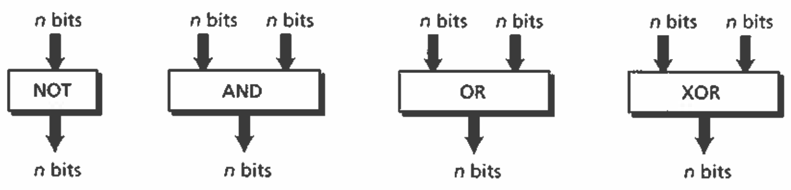
A property of XOR is that if a bit in one input is 1, the result is the complement of the corresponding bit in the other input. We use this property when we discuss the application of this operator in relation to a bit pattern.

$$\text{For } x = 0 \text{ or } 1 \qquad 1 \text{ XOR } x \rightarrow \text{NOT } x \qquad \text{and} \qquad x \text{ XOR } 1 \rightarrow \text{NOT } x$$

4.1.2 Logic operations at pattern level

The same four operators (NOT, AND, OR, and XOR) can be applied to an *n*-bit pattern. The effect is the same as applying each operator to each individual bit for NOT and to each corresponding pair of bits for other three operators. Figure 4.2 shows these four operators with input and output patterns.

Figure 4.2 *Logic operators applied to bit patterns*



Example 4.3

Use the NOT operator on the bit pattern 10011000.

Solution

The solution is shown below. Note that the NOT operator changes every 0 to 1 and every 1 to 0.

NOT	1	0	0	1	1	0	0	0	Input
	0	1	1	0	0	1	1	1	Output

Example 4.4

Use the AND operator on the bit patterns 10011000 and 00101010.

Solution

The solution is shown below. Note that only one bit in the output is 1, where both corresponding inputs are 1s.

	1	0	0	1	1	0	0	0	Input 1
AND	0	0	1	0	1	0	1	0	Input 2
	0	**0**	**0**	**0**	**1**	**0**	**0**	**0**	Output

Example 4.5

Use the OR operator on the bit patterns 10011001 and 00101110.

Solution

The solution is shown below. Note that only one bit in the output is 0, where both corresponding inputs are 0s.

	1	0	0	1	1	0	0	1	Input 1
OR	0	0	1	0	1	1	1	0	Input 2
	1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	Output

Example 4.6

Use the XOR operator on the bit patterns 10011001 and 00101110.

Solution

The solution is shown below. Compare the output in this example with the one in Example 4.5. The only difference is that when the two inputs are 1s, the result is 0 (the effect of exclusion).

	1	0	0	1	1	0	0	1	Input 1
XOR	0	0	1	0	1	1	1	0	Input 2
	1	**0**	**1**	**1**	**0**	**1**	**1**	**1**	Output

Applications

Four logic operations can be used to modify a bit pattern.

Complementing

The only application of the NOT operator is to complement the whole pattern. Applying this operator to a pattern changes every 0 to 1 and every 1 to 0. This is sometimes referred to as a one's complement operation. Example 4.3 shows the effect of complementing.

Unsetting specific bits

One of the applications of the AND operator is to unset (force to 0) specific bits in a bit pattern. The second input in this case is called a **mask**. The 0-bits in the mask unset the corresponding bits in the first input: the 1-bits in the mask leave the corresponding bits in the first input unchanged. This is due to the property we mentioned for the AND operator: if one of the inputs is 0, the output is 0 no matter what the other input is. Unsetting the bits in a pattern have many applications. For example, if an image is using only one bit per pixel (a black and white image), then we can make a specific pixel black using a mask and the AND operator.

Example 4.7

Use a mask to unset (clear) the five leftmost bits of a pattern. Test the mask with the pattern 10100110.

Solution

The mask is 00000111. The result of applying the mask is:

	1	0	1	0	0	1	1	0	Input
AND	0	0	0	0	0	1	1	1	Mask
	0	0	0	0	0	1	1	0	Output

Note that the three rightmost bits remain unchanged, while the five leftmost bits are unset (changed to 0) no matter what their previous values.

Setting specific bits

One of the applications of the OR operator is to set (force to 1) specific bits in a bit pattern. Again we can use a mask, but a different one. The 1-bits in the mask set the corresponding bits in the first input, and the 0-bits in the mask leave the corresponding bits in the first input unchanged. This is due to the property we mentioned for the OR operator: if one of the inputs is 1, the output is 1 no matter what the other input is. Setting the bits in a pattern has many applications. For example, if an image is using only one bit per pixel (a black and white image), then we can make a specific pixel white using a mask and the OR operator.

Example 4.8

Use a mask to set the five leftmost bits of a pattern. Test the mask with the pattern 10100110.

Solution

The mask is 11111000. The result of applying the mask is:

	1	0	1	0	0	1	1	0	Input
OR	1	1	1	1	1	0	0	0	Mask
	1	1	1	1	1	1	1	0	Output

Flipping specific bits

One of the applications of the XOR operator is to flip (complement) specific bits in a bit pattern. Again we can use a mask, but a different one. The 1-bits in the mask flip the corresponding bits in the first input, and the 0-bits in the mask leave the corresponding bits in the first input unchanged. This is due to the property we mentioned for the XOR operator: if one of the inputs is 1, the output is the complement of the corresponding bit. Note the difference between the NOT operator and the XOR operator. The NOT operator complements all the bits in the input, while the XOR operator complements only the specific bits in the first input as defined by the mask.

Example 4.9

Use a mask to flip the five leftmost bits of a pattern. Test the mask with the pattern 10100110.

Solution

The mask is 11111000. The result of applying the mask is:

	1	0	1	0	0	1	1	0	Input 1
XOR	1	1	1	1	1	0	0	0	Mask
	0	1	0	1	1	1	1	0	Output

4.2 SHIFT OPERATIONS

Shift operations move the bits in a pattern, changing the positions of the bits. They can move bits to the left or to the right. We can divide shift operations into two categories: logical shift operations and arithmetic shift operations.

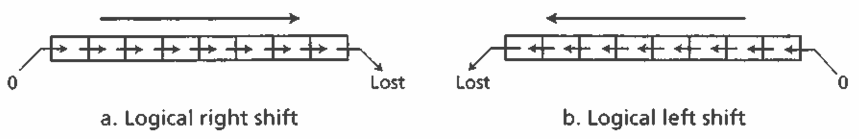
4.2.1 Logical shift operations

A **logical shift operation** is applied to a pattern that does not represent a signed number. The reason is that these shift operation may change the sign of the number that is defined by the leftmost bit in the pattern. We distinguish two types of logical shift operations, as described below.

Logical shift

A logical right shift operation shifts each bit one position to the right. In an n-bit pattern, the rightmost bit is lost and a 0 fills the leftmost bit. A logical left shift operation shifts each bit one position to the left. In an n-bit pattern, the leftmost bit is lost and a 0 fills the rightmost bit. Figure 4.3 shows the logical right shift and logical left shift operations for an 8-bit pattern.

Figure 4.3 *Logical shift operations*



Example 4.10

Use a logical left shift operation on the bit pattern 10011000.

Solution

The solution is shown below. The leftmost bit (white in the black background) is lost and a 0 is inserted as the rightmost bit (the bit in color).

←	1	0	0	1	1	0	0	0	Original
	0	0	1	1	0	0	0	0	After shift

Circular shift

A **circular shift operation** (or rotate operation) shift bits, but no bit is lost or added. A circular right shift (or *right rotate*) shifts each bit one position to the right. The rightmost bit is circulated and becomes the leftmost bit. A circular left shift (or *left rotate*) shifts each bit one position to the left. The leftmost bit circulates and become the rightmost bit. Figure 4.4 shows the circular shift left and circular shift right operation.

Figure 4.4 *Circular shift operations*

a. Circular right shift

b. Circular left shift

Example 4.11

Use a circular left shift operation on the bit pattern 10011000.

Solution

The solution is shown below. The leftmost bit (the white bit in the black background) is circulated and becomes the rightmost bit.

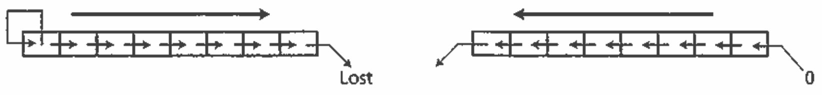
Original	1	0	0	1	1	0	0	0
After shift	0	0	1	1	0	0	0	1

Circular

Arithmetic shift operations

Arithmetic shift operations assume that the bit pattern is a signed integer in two's complement format. Arithmetic right shift is used to divide an integer by two, while arithmetic left shift is used to multiply an integer by two (discussed later). These operations should not change the sign (leftmost) bit. An arithmetic right shift retains the sign bit, but also copies it into the next right bit, so that the sign is preserved. An arithmetic left shift discards the sign bit and accept the bit to the left of the sign bit as the sign. If the new sign bit is the same as the previous one, the operation is successful, otherwise an overflow or underflow has occurred and the result is not valid. Figure 4.5 shows these two operations.

Figure 4.5 *Arithmetic shift operations*



a. Arithmetic right shift b. Arithmetic left shift

Example 4.12

Use an arithmetic right shift operation on the bit pattern 10011001. The pattern is an integer in two's complement format.

Solution

The solution is shown below. The leftmost bit is retained and also copied to its right neighbor bit (the white bit over the black background). The bit in color is lost.

Arithmetic Right	1	0	0	1	1	0	0	1	Original
	1	1	0	0	1	1	0	0	After shift

The original number was –103 and the new number is –52, which is the result of dividing –103 by 2 truncated to the smaller integer.

Example 4.13

Use an arithmetic left shift operation on the bit pattern 11011001. The pattern is an integer in two's complement format.

Solution

The solution is shown below. The leftmost bit (shown in color) is lost and a 0 (shown as white in the black background) is inserted as the rightmost bit.

Arithmetic Right	1	1	0	1	1	0	0	1	Original
	1	0	1	1	0	0	1	0	After shift

The original number was –39 and the new number is –78. The original number is multiplied by two. The operation is valid because no underflow occurred.

Example 4.14

Use an arithmetic left shift operation on the bit pattern 01111111. The pattern is an integer in two's complement format.

Solution

The solution is shown below. The leftmost bit (in color) is lost and a 0 (white in the black background) is inserted as the rightmost bit.

Arithmetic Right	0	1	1	1	1	1	1	1	Original
	1	1	1	1	1	1	1	0	After shift

The original number was 127 and the new number is −2. Here the result is not valid because an overflow has occurred. The expected answer 127 × 2 = 254 cannot be represented by an 8-bit pattern.

Example 4.15

Combining logic operations and logical shift operations give us some tools for manipulating bit patterns. Assume that we have a pattern and we need to use the third bit (from the right) of this pattern in a decision-making process. We want to know if this particular bit is 0 or 1. The following shows how we can find out.

	h	g	f	e	d	c	b	a	Original
	0	h	g	f	e	d	c	b	One right shift
	0	0	h	g	f	e	d	c	Two right shifts
AND	0	0	0	0	0	0	0	1	Mask
	0	0	0	0	0	0	0	c	Result

We shift the pattern two bits to the right so that the target bit moves to the rightmost position. The result is then ANDed with a mask which has one 1 at the leftmost position. The result is a pattern with seven 0s and the target bit at the rightmost position. We can then test the result: if it is an unsigned integer 1, the target bit was 1, whereas if the result is an unsigned integer 0, the target bit was 0.

4.3 ARITHMETIC OPERATIONS

Arithmetic operations involve adding, subtracting, multiplying, and dividing. We can apply these operations to integers and floating-point numbers.

4.3.1 Arithmetic operations on integers

All arithmetic operations such as addition, subtraction, multiplication, and division can be applied to integers. Although multiplication (division) of integers can be implemented using repeated addition (subtraction), the procedure is not efficient. There are more efficient procedures for multiplication and division, such as Booth procedures, but these are beyond the scope of this book. For this reason, we only discuss addition and subtraction of integers here.

Addition and subtraction for two's complement integers

We first discuss addition and subtraction for integers in two's complement representation, because it is easier. As we discussed in Chapter 3, integers are normally stored in two's complement format. One of the advantages of two's complement representation is that there is no difference between addition and subtraction. When the subtraction operation is encountered, the computer simply changes it to an addition operation, but makes two's complement of the second number. In other words:

$$A - B \leftrightarrow A + (\overline{B} + 1) \quad \text{where } ((\overline{B} + 1)) \text{ means the two's complement of B}$$

This means that we only need to discuss addition. Adding numbers in two's complement is like adding the numbers in decimal: we add column by column, and if there is a carry, it is added to the next column. However, the carry produced from the last column is discarded.

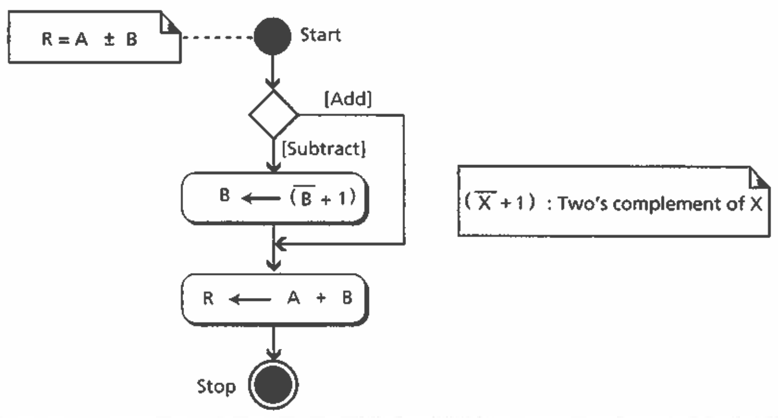
We should remember that we add integers column by column. In each column, we have either two bits to add if there is no carry from the previous column, or three bits to add if there is a carry from the previous column. The number of 1s in each column can be zero, one, two, or three. Table 4.1 shows the sum and carry (C).

Table 4.1 Carry and sum resulting from adding two bits

Column	*Carry*	*Sum*	*Column*	*Carry*	*Sum*
Zero 1s	0	0	Two 1s	1	0
One 1	0	1	Three 1s	1	1

Now we can show the procedure for addition or subtraction of two integers in two's complement format (Figure 4.6). Note that we use the notation (X + 1) to mean two's complement of X. This notation is very common in literature because X denotes the one's complement of X. If we add 1 to the one's complement of an integer, we get its two's complement.

Figure 4.6 *Addition and subtraction of integers in two's complement format*


R = A ± B
Start
[Add]
[Subtract]
$B \leftarrow (\overline{B} + 1)$
$R \leftarrow A + B$
Stop
$(\overline{X} + 1)$: Two's complement of X

The procedure is as follows:

1. If the operation is subtraction, we take the two's complement of the second integer. Otherwise, we move to the next step.
2. We add the two integers.

Example 4.16

Two integers A and B are stored in two's complement format. Show how B is added to A.

$$A = (00010001)_2 \quad B = (00010110)_2$$

Solution

The operation is adding. A is added to B and the result is stored in R.

$$\begin{array}{cccccccccl}
 & & & 1 & & & & & & \text{Carry} \\
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \text{A} \\
+ & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & \text{B} \\
\hline
 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & \text{R}
\end{array}$$

We check the result in decimal: (+17) + (+22) = (+39).

Example 4.17

Two integers A and B are stored in two's complement format. Show how B is added to A.

$$A = (00011000)_2 \quad B = (11101111)_2$$

Solution

The operation is adding. A is added to B and the result is stored in R. Note that the last carry is discarded because the size of the memory is only 8 bits.

$$\begin{array}{cccccccccl}
1 & 1 & 1 & 1 & 1 & & & & & \text{Carry} \\
 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & \text{A} \\
+ & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & \text{B} \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \text{R}
\end{array}$$

Checking the result in decimal, (+24) + (−17) = (+7).

Example 4.18

Two integers A and B are stored in two's complement format. Show how B is subtracted from A.

$$A = (00011000)_2 \quad B = (11101111)_2$$

Solution

The operation is subtracting. A is added to $(\overline{B} + 1)$ and the result is stored in R.

$$\begin{array}{cccccccccl}
 & & & 1 & & & & & & \text{Carry} \\
 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & \text{A} \\
+ & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & (\overline{B} + 1) \\
\hline
 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & \text{R}
\end{array}$$

Checking the result in decimal, (+24) − (−17) = (+41).

Example 4.19

Two integers A and B are stored in two's complement format. Show how B is subtracted from A.

$$A = (11011101)_2 \quad B = (00010100)_2$$

Solution

The operation is subtracting. A is added to $(\overline{B} + 1)$ and the result is stored in R.

1	1	1	1	1	1				Carry
	1	1	0	1	1	1	0	1	A
+	1	1	1	0	1	1	0	0	$(\overline{B} + 1)$
	1	1	0	0	1	0	0	1	R

Checking the result in decimal, (−35) − (+20) = (−55). Note that the last carry is discarded.

Example 4.20

Two integers A and B are stored in two's complement format. Show how B is added to A.

$$A = (01111111)_2 \quad B = (00000011)_2$$

Solution

The operation is adding. A is added to B and the result is stored in R.

	1	1	1	1	1	1	1		Carry
	0	1	1	1	1	1	1	1	A
+	0	0	0	0	0	0	1	1	B
	1	0	0	0	0	0	1	0	R

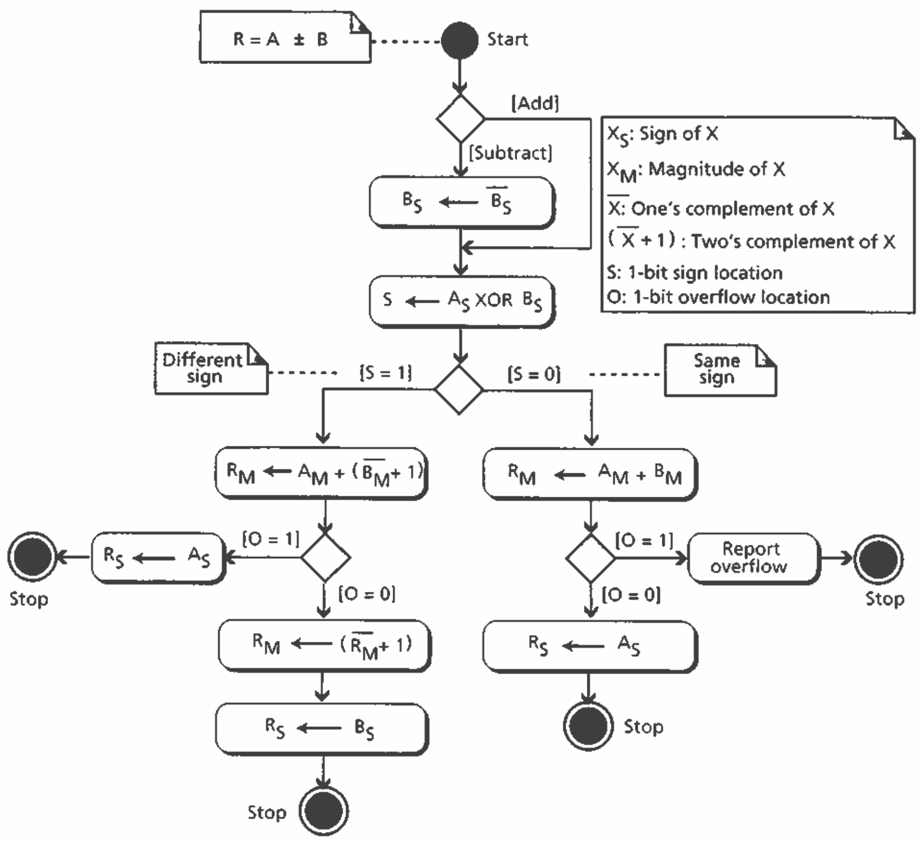
We expect the result to be 127 + 3 = 130, but the answer is −126. The error is due to overflow, because the expected answer (+130) is not in the range −128 to +127.

When we do arithmetic operations on numbers in a computer, we should remember that each number and the result should be in the range defined by the bit allocation.

Addition or subtraction for sign-and-magnitude integers

Addition and subtraction for integers in sign-and-magnitude representation looks very complex. We have four different combinations of signs (two signs, each of two values) for addition, and four different conditions for subtraction. This means that we need to consider eight different situations. However, if we first check the signs, we can reduce these cases, as shown in Figure 4.7.

Figure 4.7 *Addition and subtraction of integers in sign-and-magnitude format*


R = A ± B
Start
[Add]
[Subtract]
$B_S \leftarrow \overline{B_S}$
X_S: Sign of X
X_M: Magnitude of X
$\overline{X}$: One's complement of X
$(\overline{X}+1)$: Two's complement of X
S: 1-bit sign location
O: 1-bit overflow location
$S \leftarrow A_S$ XOR B_S
Different sign
[S = 1]
[S = 0]
Same sign
$R_M \leftarrow A_M + (\overline{B_M}+1)$
$R_M \leftarrow A_M + B_M$
[O = 1]
$R_S \leftarrow A_S$
Stop
[O = 0]
$R_M \leftarrow (\overline{R_M}+1)$
$R_S \leftarrow B_S$
Stop
[O = 1]
Report overflow
Stop
[O = 0]
$R_S \leftarrow A_S$
Stop

Let us first explain the diagram:

1. We check the operation. If the operation is subtraction, we change the sign of the second integer (B). This means we now only have to worry about addition of two signed integers.
2. We apply the XOR operation to the two signs. If the result (stored in temporary location S) is 0, it means that the signs are the same (either both signs are positive or both are negative).
3. If the signs are the same, $R = \pm (A_M + B_M)$. We need to add the magnitude and the sign of the result is the common sign. So, we have:

$$R_M = (A_M) + (B_M) \quad \text{and} \quad R_S = A_S$$

   Where the subscript M means magnitude and subscript S means sign. In this case, however, we should be careful about the overflow. When we add the two magnitudes, an overflow may occur that must be reported and the process aborted.
4. If the sign are different, $R = \pm (A_M - B_M)$. So we need to subtract B_M from A_M and then make a decision about the sign. Instead of subtracting bit by bit, we take the two's

complement of the second magnitude (B_M) and add them. The sign of the result is the sign of the integer with larger magnitude.

a. It can be shown that if $A_M \geq B_M$, there is an overflow and the result is a positive number. Therefore, if there is an overflow, we discard the overflow and the let the sign of the result to be the sign of A.

b. It can be shown that if $A_M < B_M$, there is no overflow, but the result is a negative number. So if there is no overflow, we make the two's complement of the result and let the sign of the result be the sign of B.

Example 4.21

Two integers A and B are stored in sign-and-magnitude format (we have separated the sign from the magnitude for clarity). Show how B is added to A.

$$A = (0\ 0010001)_2 \quad B = (0\ 0010110)_2$$

Solution

The operation is adding: the sign of B is not changed. Since $S = A_S$ XOR $B_S = 0$, $R_M = A_M + B_M$ and $R_S = A_S$. There is no overflow.

	Sign	No overflow									
					1						Carry
A_S	0			0	0	1	0	0	0	1	A_M
B_S	0		+	0	0	1	0	1	1	0	B_M
R_S	0			0	1	0	0	1	1	1	R_M

Checking the result in decimal, (+17) + (+22) = (+39).

Example 4.22

Two integers A and B are stored in sign-and-magnitude format. Show how B is added to A.

$$A = (0\ 0010001)_2 \quad B = (1\ 0010110)_2$$

Solution

The operation is adding: the sign of B is not changed. $S = A_S$ XOR $B_S = 1$; $R_M = A_M + (\bar{B}_M + 1)$. Since there is no overflow, we need to take the two's complement of R_M. The sign of R is the sign of B.

	Sign	No overflow									Carry
A_S	0			0	0	1	0	0	0	1	A_M
B_S	1		+	1	1	0	1	0	1	0	$(\bar{B}_M+1)$
				1	1	1	1	0	1	1	R_M
R_S	1			0	0	0	0	1	0	1	$R_M=(\bar{R}_M+1)$

Checking the result in decimal, (+17) + (−22) = (−5).

Example 4.23

Two integers A and B are stored in sign-and-magnitude format. Show how B is subtracted from A.

$$A = (1\ 1010001)_2 \quad B = (1\ 0010110)_2$$

Solution

The operation is subtracting: $B_S = \overline{B}_S$. $S = A_S$ XOR $B_S = 1$, $R_M = A_M + (\overline{B}_M + 1)$. Since there is an overflow, the value of R_M is final. The sign of R is the sign of A.

	Sign	Overflow	1								Carry
A_S	1			1	0	1	0	0	0	1	A_M
B_S	1		+	1	1	0	1	0	1	0	$(\overline{B}_M+1)$
R_S	1			0	1	1	1	0	1	1	R_M

Checking the result in decimal, (−81) − (−22) = (−59).

4.3.2 Arithmetic operations on reals

All arithmetic operations such as addition, subtraction, multiplication, and division can be applied to reals stored in floating-point format. Multiplication of two reals involves multiplication of two integers in sign-and-magnitude representation. Division of two reals involves division of two integers in sign-and-magnitude representations. Since we did not discuss the multiplication or division of integers in sign-and-magnitude representation, we will not discuss the multiplication and division of reals, and only show addition and subtractions for reals.

Addition and subtraction of reals

Addition and subtraction of real numbers stored in floating-point numbers is reduced to addition and subtraction of two integers stored in sign-and-magnitude (combination of sign and mantissa) after the alignment of decimal points. Figure 4.8 shows a simplified version of the procedure (there are some special cases that we have ignored).

The simplified procedure works as follows:

1. If any of the two numbers (A or B) is zero, we let the result be 0 and stop.
2. If the operation is subtraction, we change the sign of the second number (B) to simulate addition.
3. We denormalize both numbers by including the hidden 1 in the mantissa and incrementing the exponents. The mantissa is now is treated as an integer.
4. We then align the exponents, which means that we increment the lower exponent and shift the corresponding mantissa until both have the same exponent. For example, if we have:

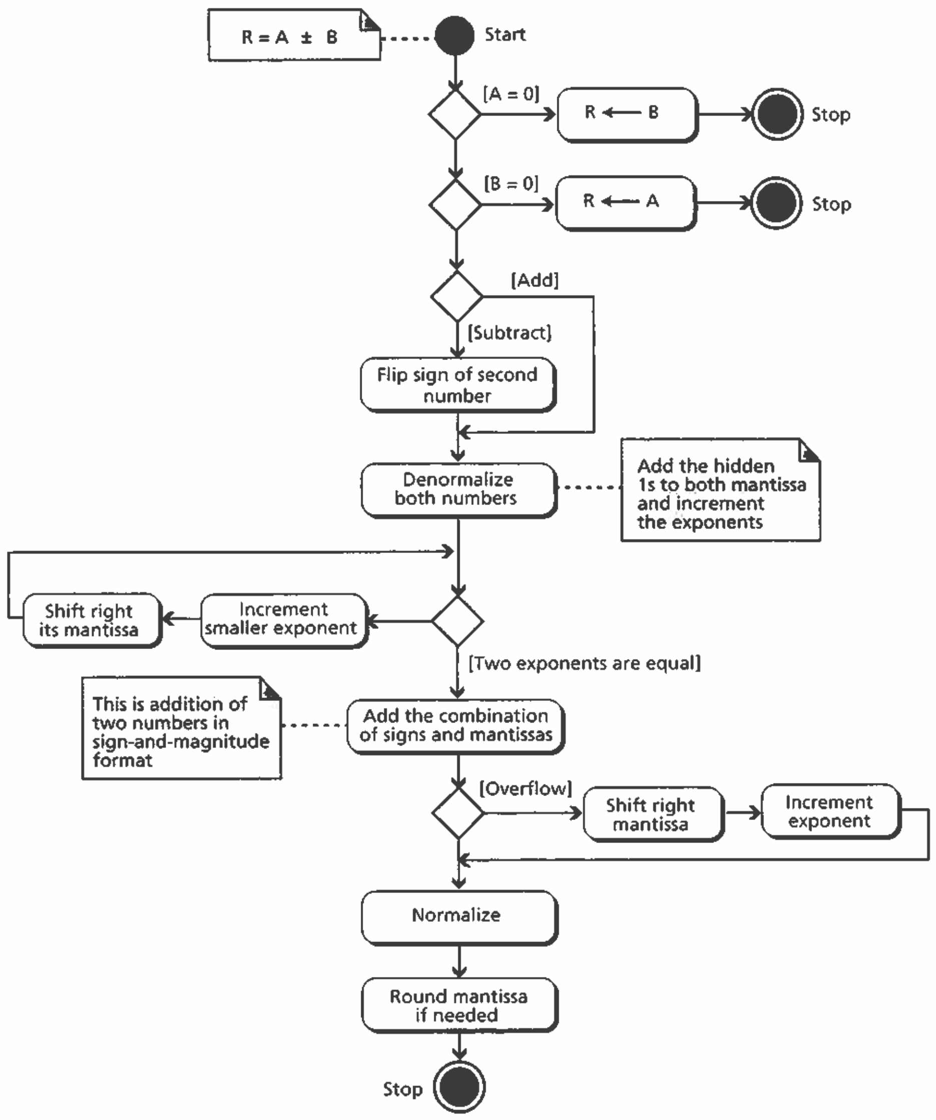
$$1.11101 \times 2^4 + 1.01 \times 2^2$$

We need to make both exponents 4:

$$1.11101 \times 2^4 + 0.0101 \times 2^4$$

5. Now, we treat the combination of the sign and mantissa of each number as an integer in sign-and-magnitude format. We add these two integers, as explained earlier in this chapter.
6. Finally, we normalized the number again to 1.000111×2^5.

Figure 4.8 *Addition and subtraction of reals in floating-point format*


R = A ± B
Start
[A = 0]
R ← B
Stop
[B = 0]
R ← A
Stop
[Add]
[Subtract]
Flip sign of second number
Denormalize both numbers
Add the hidden 1s to both mantissa and increment the exponents
Shift right its mantissa
Increment smaller exponent
[Two exponents are equal]
This is addition of two numbers in sign-and-magnitude format
Add the combination of signs and mantissas
[Overflow]
Shift right mantissa
Increment exponent
Normalize
Round mantissa if needed
Stop

Example 4.24

Show how the computer finds the result of (+5.75) + (+161.875) = (+167.625).

Solution

As we saw in Chapter 3, these two numbers are stored in floating-point format, as shown below, but we need to remember that each number has a hidden 1 (which is not stored, but assumed). Note that S is the sign, E is exponent, and M is mantissa.

	S	*E*	*M*
A	0	10000001	01110000000000000000000
B	0	10000110	01000011110000000000000

The first few steps in the UML diagram (Figure 4.8) are not needed. We move to denormalization and denormalize the numbers by adding the hidden 1s to the mantissa and incrementing the exponent. Now both denormalized mantissas are 24 bits and include the hidden 1s. They should be stored in a location that can hold all 24 bits. Each exponent is incremented.

	S	*E*	*Denormalized M*
A	0	10000010	101110000000000000000000
B	0	10000111	101000011110000000000000

Now we need to align the mantissas. We need to increment the first exponent and shift its mantissa to the right. We change the first exponent to $(10000111)_2$, so we need to shift the first mantissa right by five positions.

	S	*E*	*Denormalized M*
A	0	10000111	000001011100000000000000
B	0	10000111	101000011110000000000000

Now we do sign-and-magnitude addition, treating the sign and the mantissa of each number as one integer stored in sign-and-magnitude representation.

	S	*E*	*Denormalized M*
R	0	10000111	101001111010000000000000

There is no overflow in the mantissa, so we normalize.

	S	*E*	*M*
R	0	10000110	01001111010000000000000

The mantissa is only 23 bits, no rounding is needed. E = $(10000110)_2$ = 134 M = 0100111101. In other words, the result is $(1.0100111101)_2 \times 2^{134-127} = (10100111.101)_2 =$ 167.625.

Example 4.25

Show how the computer finds the result of (+5.75) + (−7.0234375) = −1.2734375.

Solution

These two numbers can be stored in floating-point format, as shown below:

	S	*E*	*M*
A	0	10000001	01110000000000000000000
B	1	10000001	11000001100000000000000

Denormalization results in:

	S	*E*	*Denormalized M*
A	0	10000010	1011100000000000000000000
B	1	10000010	1110000011000000000000000

Alignment is not needed (both exponents are the same), so we apply addition operation on the combinations of sign and mantissa. The result is shown below, in which the sign of the result is negative:

	S	*E*	*Denormalized M*
R	1	10000010	0010100011000000000000000

Now we need to normalize. We decrement the exponent three times and shift the denormalized mantissa to the left three positions:

	S	*E*	*M*
R	1	01111111	010001100000000000000000

The mantissa is now 24 bits, so we round it to 23 bits.

	S	*E*	*M*
R	1	01111111	01000110000000000000000

The result is R = $-2^{127-127} \times 1.0100011$ = −1.2734375, as expected.

4.4 END-CHAPTER MATERIALS

4.4.1 Recommended Reading

For more details about the subjects discussed in this chapter, the following books are recommended:

❏ Mano, M. *Computer System Architecture*, Upper Saddle River, NJ: Prentice Hall, 1993

- Null, L. and Lobur, J. *Computer Organization and Architecture*, Sudbury, MA: Jones and Bartlett, 2003
- Stalling, W. *Computer Organization and Architecture*, Upper Saddle River, NJ: Prentice Hall, 2000.

4.4.2 Key terms

This chapter has introduced the following key terms, which are listed here with the pages on which they first occur:

AND operation 77	arithmetic operation 84
arithmetic shift operation 82	Boolean algebra 76
circular shift operation 82	**logical operation 76**
logical shift operation 81	mask 79
NOT operation 77	OR operation 77
truth table 76	XOR operation 77

4.4.3 Summary

- Operations on data can be divided into three broad categories: logic operations, shift operations, and arithmetic operations. Logic operations refer to those operations that apply the same basic operation to individual bits of a pattern or to two corresponding bits in two patterns. Shift operations move the bits in the pattern. Arithmetic operations involve adding, subtracting, multiplying, and dividing.
- The four logic operators discussed in this chapter (NOT, AND, OR, and XOR) can be used at the bit level or the pattern level. The NOT operator is a unary operator, while the AND, OR, and XOR operators are binary operators.
- The only application of the NOT operator is to complement the whole pattern. One of the applications of the AND operator is to unset (force to 0) specific bits in a bit pattern. One of the applications of the OR operator is to set (force to 1) specific bits in a bit pattern. One of the applications of the XOR operator is to flip (complement) specific bits in a bit pattern.
- Shift operations move the bits in the pattern: they change the positions of the bits. We can divide shift operations into two categories: logical shift operations and arithmetic shift operations. A logical shift operation is applied to a pattern that does not represent a signed number. Arithmetic shift operations assume that the bit pattern is a signed integer in two's complement format.
- All arithmetic operations such as addition, subtraction, multiplication, and division can be applied to integers. Integers are normally stored in two's complement format. One of the advantages of two's complement representation is that there is no difference between addition and subtraction. When the subtraction operation is encountered, the computer simply changes it to an addition operation, but forms the two's complement of the second number. Addition and subtraction for integers in sign-and-magnitude representation looks very complex. We have eight situations to consider.

- ❑ All arithmetic operations such as addition, subtraction, multiplication, and division can be applied to reals stored in floating-point format. Addition and subtraction of real numbers stored in floating-point numbers is reduced to addition and subtraction of two integers stored in sign and magnitude after the alignment of decimal points.

4.5 PRACTICE SET

4.5.1 Quizzes

A set of interactive quizzes for this chapter can be found on the book website. It is strongly recommended that the student take the quizzes to check their understanding of the materials before continuing with the practice set.

4.5.2 Review questions

Q4-1. What is the difference between an arithmetic operation and a logical operation?

Q4-2. What happens to the carry from the leftmost column in the addition of integers in two's complement format?

Q4-3. Can *n*, the bit allocation, equal 1? Why, or why not?

Q4-4. Define the term *overflow*.

Q4-5. In the addition of floating-point numbers, how do we adjust the representation of numbers with different exponents?

Q4-6. What is the difference between a unary operation and a binary operation?

Q4-7. Name the logical binary operations.

Q4-8. What is a truth table?

Q4-9. What does the NOT operator do?

Q4-10. When is the result of an AND operator true?

Q4-11. When is the result of an OR operator true?

Q4-12. When is the result of an XOR operator true?

Q4-13. Mention an important property of the AND operator discussed in this chapter.

Q4-14. Mention an important property of the OR operator discussed in this chapter.

Q4-15. Mention an important property of the XOR operator discussed in this chapter.

Q4-16. What binary operation can be used to set bits? What bit pattern should the mask have?

Q4-17. What binary operation can be used to unset bits? What bit pattern should the mask have?

Q4-18. What binary operation can be used to flip bits? What bit pattern should the mask have?

Q4-19. What is the difference between logical and arithmetic shifts?

4.5.3 Problems

P4-1. Show the result of the following operations:

a. NOT $(99)_{16}$
b. NOT $(FF)_{16}$
c. NOT $(00)_{16}$
d. NOT $(01)_{16}$

P4-2. Show the result of the following operations:

a. $(99)_{16}$ AND $(99)_{16}$
b. $(99)_{16}$ AND $(00)_{16}$
c. $(99)_{16}$ AND $(FF)_{16}$
d. $(FF)_{16}$ AND $(FF)_{16}$

P4-3. Show the result of the following operations:

a. $(99)_{16}$ OR $(99)_{16}$
b. $(99)_{16}$ OR $(00)_{16}$
c. $(99)_{16}$ OR $(FF)_{16}$
d. $(FF)_{16}$ OR $(FF)_{16}$

P4-4. Show the result of the following operations:

a. NOT [$(99)_{16}$ OR $(99)_{16}$]
b. $(99)_{16}$ OR [NOT $(00)_{16}$]
c. [$(99)_{16}$ AND $(33)_{16}$] OR [$(00)_{16}$ AND $(FF)_{16}$]
d. $(99)_{16}$ OR $(33)_{16}$ AND [$(00)_{16}$ OR $(FF)_{16}$]

P4-5. We need to unset (force to 0) the four leftmost bits of a pattern. Show the mask and the operation.

P4-6. We need to set (force to 1) the four rightmost bits of a pattern. Show the mask and the operation.

P4-7. We need to flip the three rightmost and the two leftmost bits of a pattern. Show the mask and the operation.

P4-8. We need to unset the three leftmost bits and set the two rightmost bits of a pattern. Show the masks and operations.

P4-9. Use the shift operation to divide an integer by 4.

P4-10. Use the shift operation to multiply an integer by 8.

P4-11. Use a combination of logical and shift operations to extract the fourth and fifth bits from the left of an unsigned integer.

P4-12. Using an 8-bit allocation, first convert each of the following integers to two's complement, do the operation, and then convert the result to decimal.

a. 19 + 23
b. 19 − 23
c. −19 + 23
d. −19 − 23

P4-13. Using a 16-bit allocation, first convert each of the following numbers to two's complement, do the operation, and then convert the result to decimal.

a. 161 + 1023
b. 161 − 1023
c. −161 + 1023
d. −161 − 1023

P4-14. Which of the following operations creates an overflow if the numbers and the result are represented in 8-bit two's complement representation?

a. 11000010 + 00111111
b. 00000010 + 00111111
c. 11000010 + 11111111
d. 00000010 + 11111111

P4-15. Without actually doing the calculation, can we tell which of the following creates an overflow if the numbers and the result are in 8-bit two's complement representation?

a. 32 + 105
b. 32 − 105
c. −32 + 105
d. −32 − 105

P4-16. Show the result of the following operations assuming that the numbers are stored in 16-bit two's complement representation. Show the result in hexadecimal notation.

a. $(012A)_{16} + (0E27)_{16}$
b. $(712A)_{16} + (9E00)_{16}$
c. $(8011)_{16} + (0001)_{16}$
d. $(E12A)_{16} + (9E27)_{16}$

P4-17. Using an 8-bit allocation, first convert each of the following numbers to sign-and-magnitude representation, do the operation, and then convert the result to decimal.

a. 19 + 23
b. 19 − 23
c. −19 + 23
d. −19 − 23

P4-18. Show the result of the following floating-point operations using IEEE_127—see Chapter 3.

a. 34.75 + 23.125
b. −12.625 + 451.00
c. 33.1875 − 0.4375
d. −344.3125 − 123.5625

P4-19. In which of the following situations does an overflow never occur? Justify the answer.

a. Adding two positive integers.
b. Adding one positive integer to a negative integer.
c. Subtracting one positive integer from a negative integer.
d. Subtracting two negative integers.

P4-20. What is the result of adding an integer to its one's complement?

P4-21. What is the result of adding an integer to its two's complement?

4.5.4 Applets

We have created some applets to show some of the main concepts discussed in this chapter. It is strongly recommended that the students active these applets on the book website and carefully examine the protocols in action.