CHAPTER 4

Operations on Data

Objectives:

- To list the three categories of operations performed on data.
- To perform unary and binary logic operations on bit patterns.
- To distinguish between logic shift operations and arithmetic shift operations.
- To perform logic shift operations on bit patterns.
- To perform arithmetic shift operations on integers stored in two’s complement format.

Objectives (continued):

- ❑ To perform addition and subtraction on integers when they are stored in two's complement format.
- ❑ To perform addition and subtraction on integers when they are stored in sign-and-magnitude format.
- ❑ To perform addition and subtraction operations on reals when they are stored in floating-point format.
- ❑ To 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.

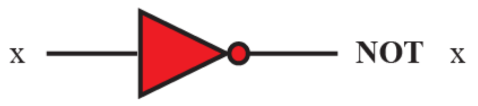
Logic operations at bit level

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.

i

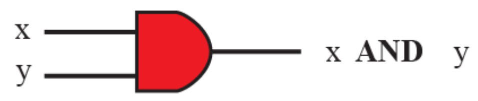
Boolean algebra and logic circuits are discussed in Appendix E.

Figure 4.1: **Logic operations at the bit level**



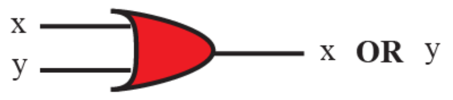
NOT

x	NOT x
0	1
1	0



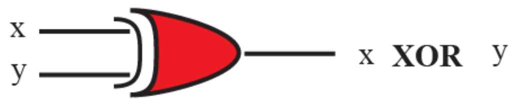
AND

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.

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.

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

OR

The OR operator is 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.

i

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

XOR

The XOR operator is a binary operator like the OR operator, with only one difference: the output is 0 if both inputs are 1s.

i

For x = 0 or 1

1 XOR x $\rightarrow$ NOT x $\quad$ x XOR 1 $\rightarrow$ NOT x

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

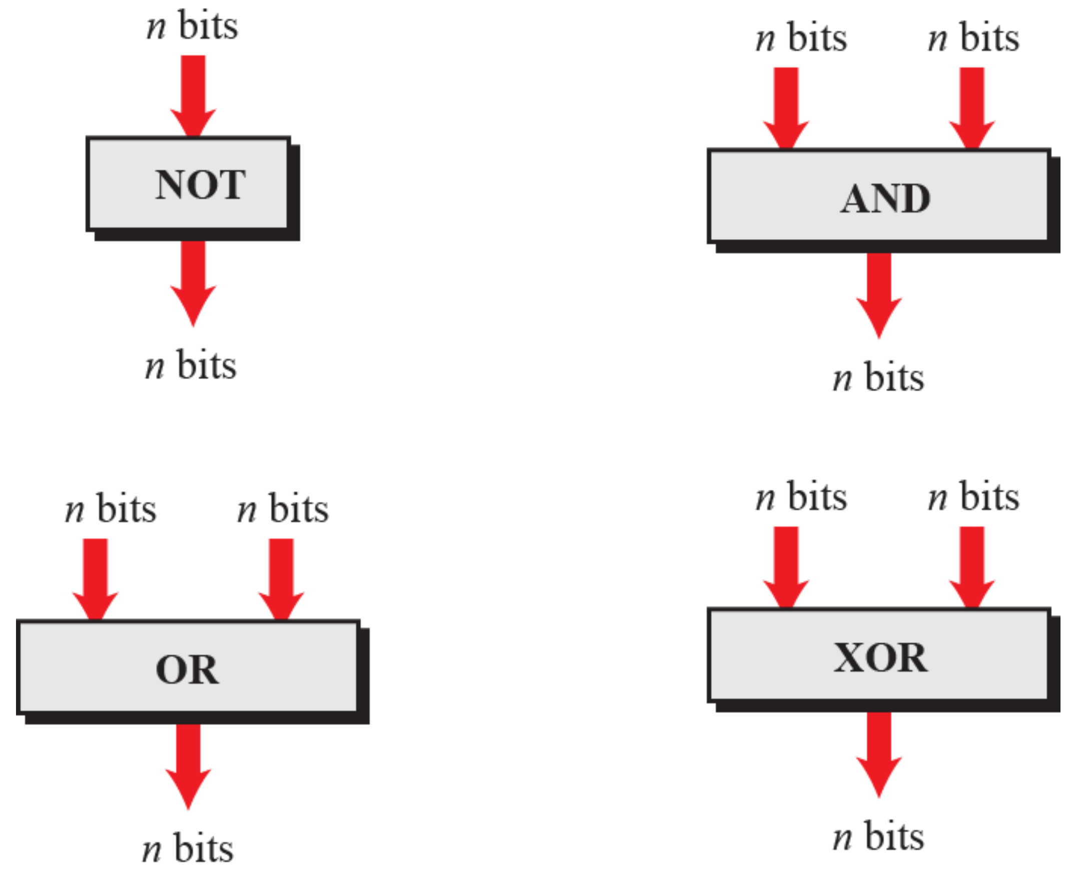
$$x \text{ XOR } y \quad \leftrightarrow \quad [x \text{ AND } (\text{NOT } y)] \text{ OR } [(\text{NOT } x) \text{ AND } y]$$

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

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 the 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

The four logic operations can be used to modify a bit pattern.

- Complementing (NOT)
- Unsetting (AND)
- Setting (OR)
- Flipping (XOR)

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.

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

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

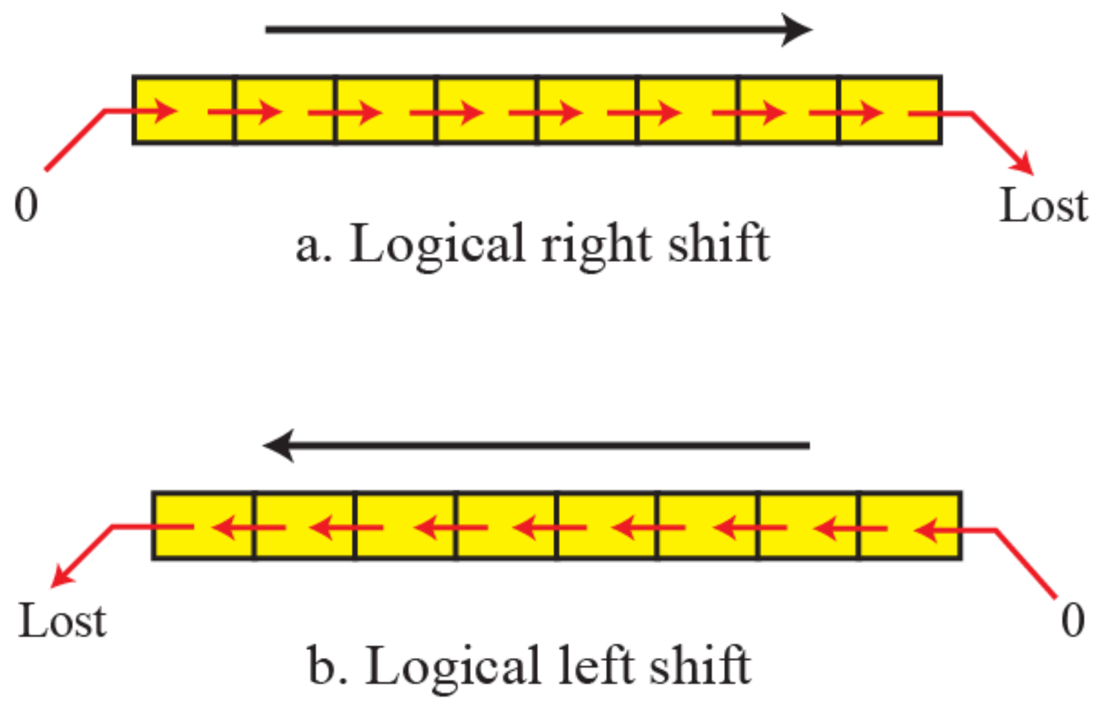
Shift operations move the bits in a pattern, changing the positions of the bits. They can move bits to the left or to the right. We can divide shift operations into two categories: logical shift operations and arithmetic shift operations.

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 operations 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
- Logical circular shift (Rotate)

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).

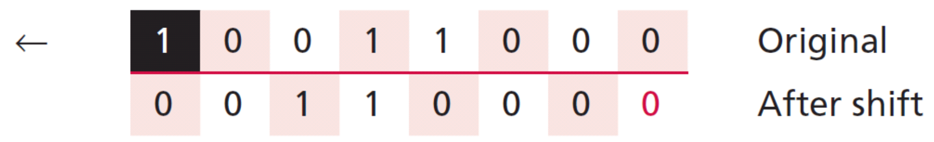
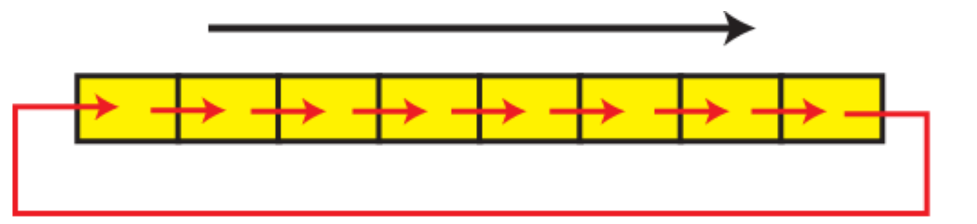
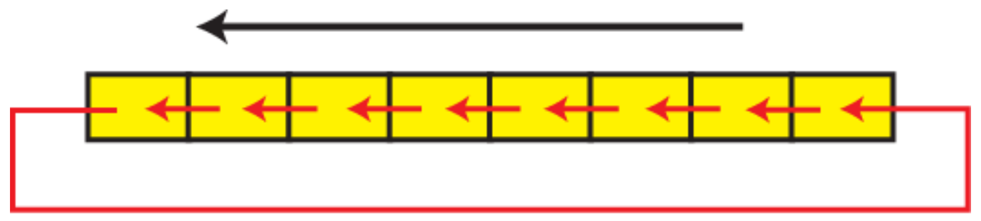


Figure 4.4: Circular shift operations


a. Circular right shift


b. Circular left shift

Example 4.11

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

Solution

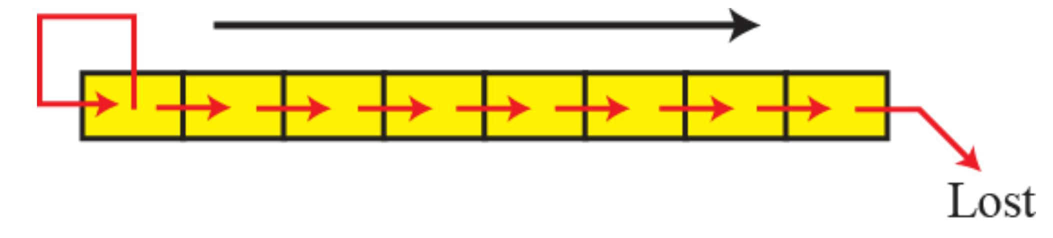
The solution is shown below. The leftmost bit (the white bit in the black background) is circulated and becomes the rightmost bit.

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

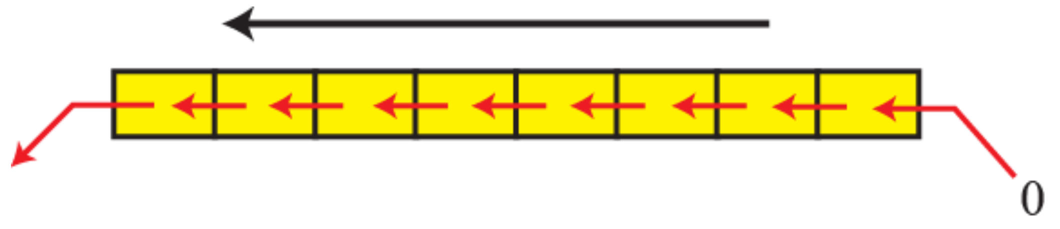
Arithmetic shift operations

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.

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.

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

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:

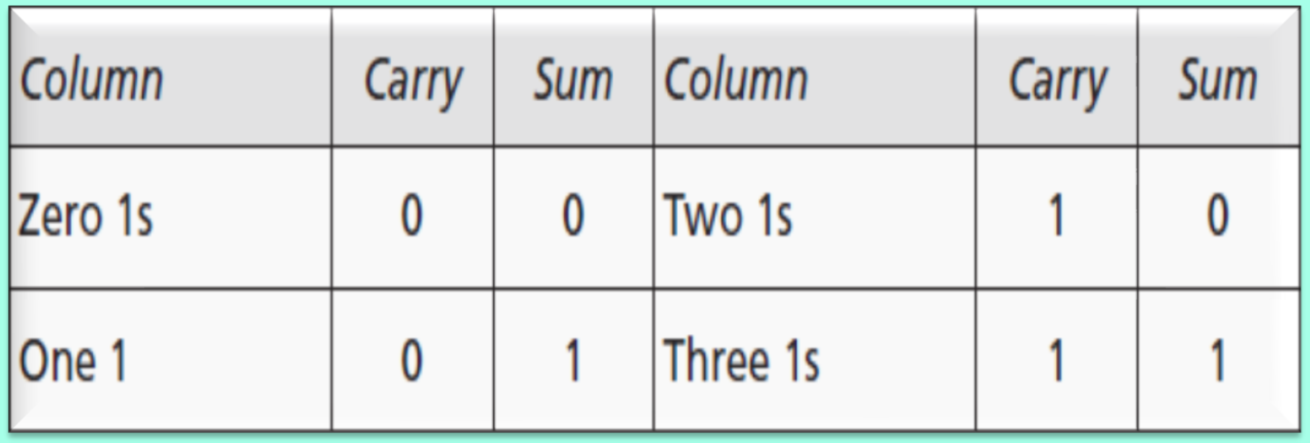
$$\boxed{A - B \leftrightarrow A + (\overline{B} + 1)}$$

Where $\overline{B}$ is the one’s complement of B and

$(\overline{B} + 1)$ means the two’s complement of B

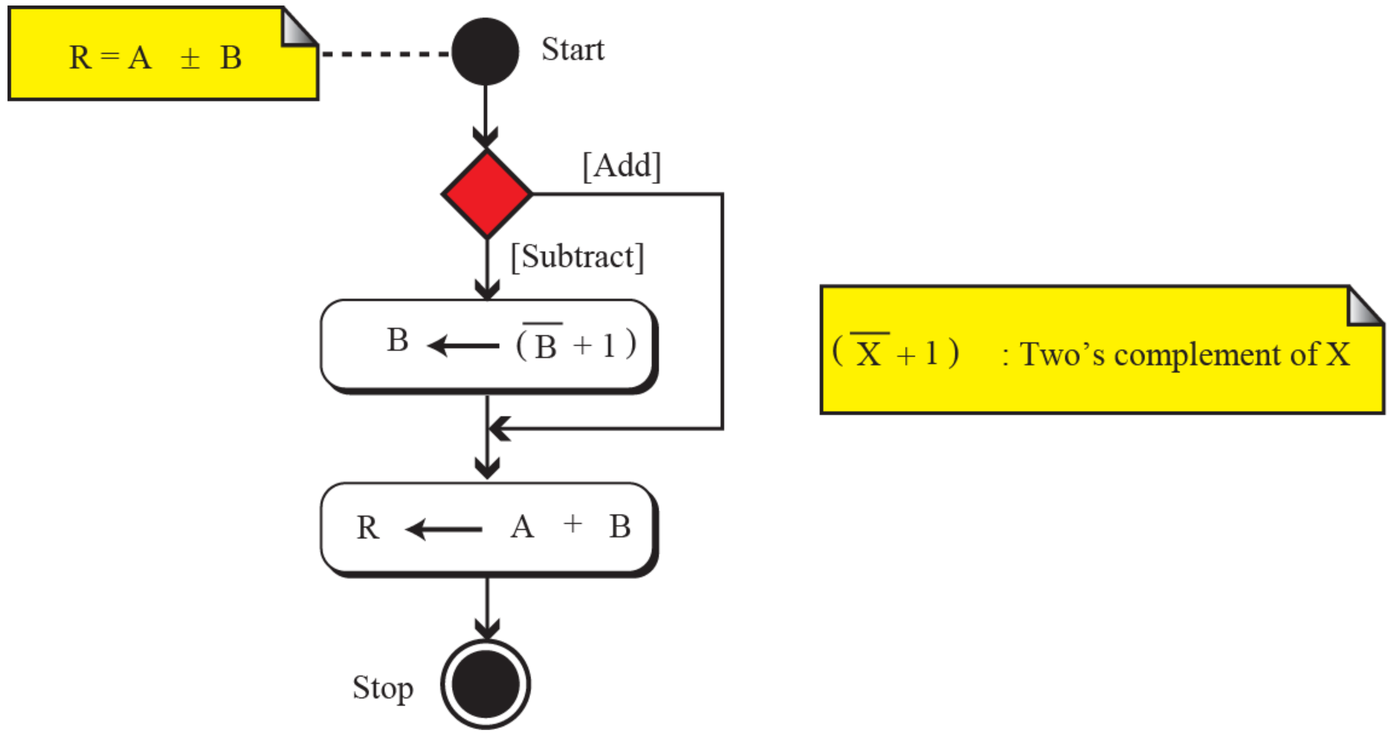
We should remember that we add integers column by column. The following table shows the sum and carry (C).

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

Figure 4.6: **Addition and subtraction in two's complement**


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

Example 4.16

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

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

Solution

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

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

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.

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

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.

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

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 = $(11011101)_2$ 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.

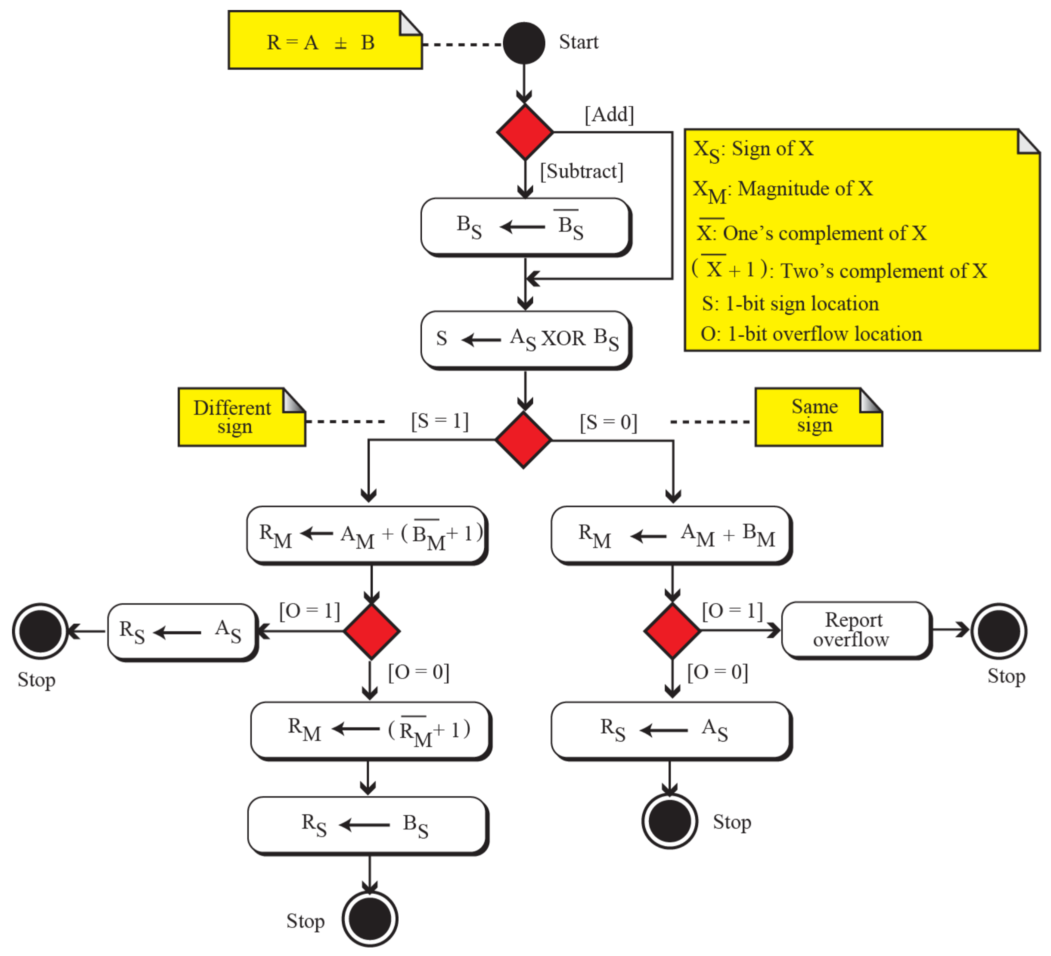
i

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.

sign-and-magnitude integers

Addition and subtraction for integers in sign-and-magnitude representation looks very complex. We have four different combination 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 in sign-and-magnitude


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]
[O = 0]
$R_S \leftarrow A_S$
Stop
Report overflow
Stop
$R_M \leftarrow (\overline{R_M} + 1)$
$R_S \leftarrow A_S$
$R_S \leftarrow B_S$
Stop
Stop

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)$_2$ 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)$_2$ B = (**1** 0010110)$_2$

Solution

The operation is adding: the sign of B is not changed. S = A_S XOR B_S = 1; $R_M = A_M + (\overline{B}_M + 1)$. 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	$(\overline{B}_M+1)$
				1	1	1	1	0	1	1	R_M
R_S	1			0	0	0	0	1	0	1	$R_M = (\overline{R}_M + 1)$

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.

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

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).

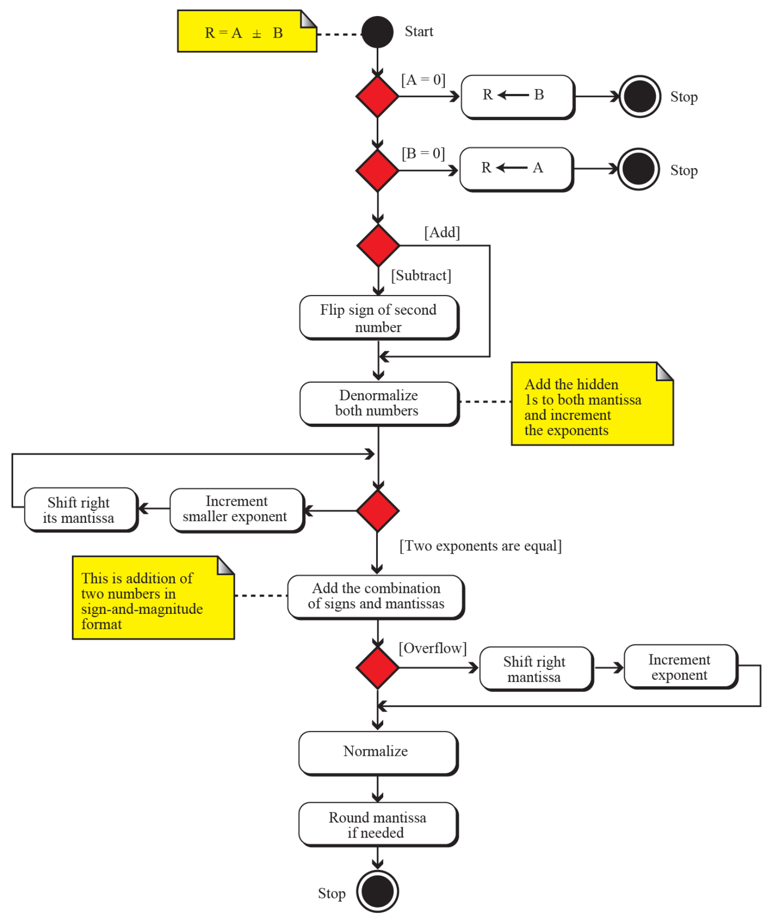
Arithmetic operations on reals

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).

Figure 4.8: Addition and subtraction of reals


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	00000**1**011100000000000000
B	0	10000111	**1**0100001111000000000000 0

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	101001111101000000000000

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

	S	E	M
R	0	10000110	01001111101000000000000

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	**1**01110000000000000000000
B	1	10000010	**1**11000001100000000000000

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	001010001100000000000000

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	0100011000000000000000000

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

	S	*E*	*M*
R	1	01111111	010001100000000000000000

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