Chapter 4

Operations on Data

OBJECTIVES

After reading this chapter, the reader 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 addition and subtraction on integers when they are stored in two's complement format.**
- **Perform addition and subtraction on integers when stored in sign-and-magnitude format.**
- **Perform addition and subtraction operations on reals stored in floating-point format.**

Operations on Data

- Three Categories:
  - Logic Operations
  - Shift Operations
  - Arithmetic Operations

4.1

LOGICAL OPERATIONS

Logic Operations

- Boolean algebra belongs to a special field of mathematics called logic.
  - A bit can take one of the two values: 0 or 1.
  - A truth table defines the values of the output for each possible input or inputs.
    - The output of each operator is always one bit.

- Operation: A + B
  - Operator : +
  - Operands : A, B

Truth Tables

- 1 - True
- 0 - False

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

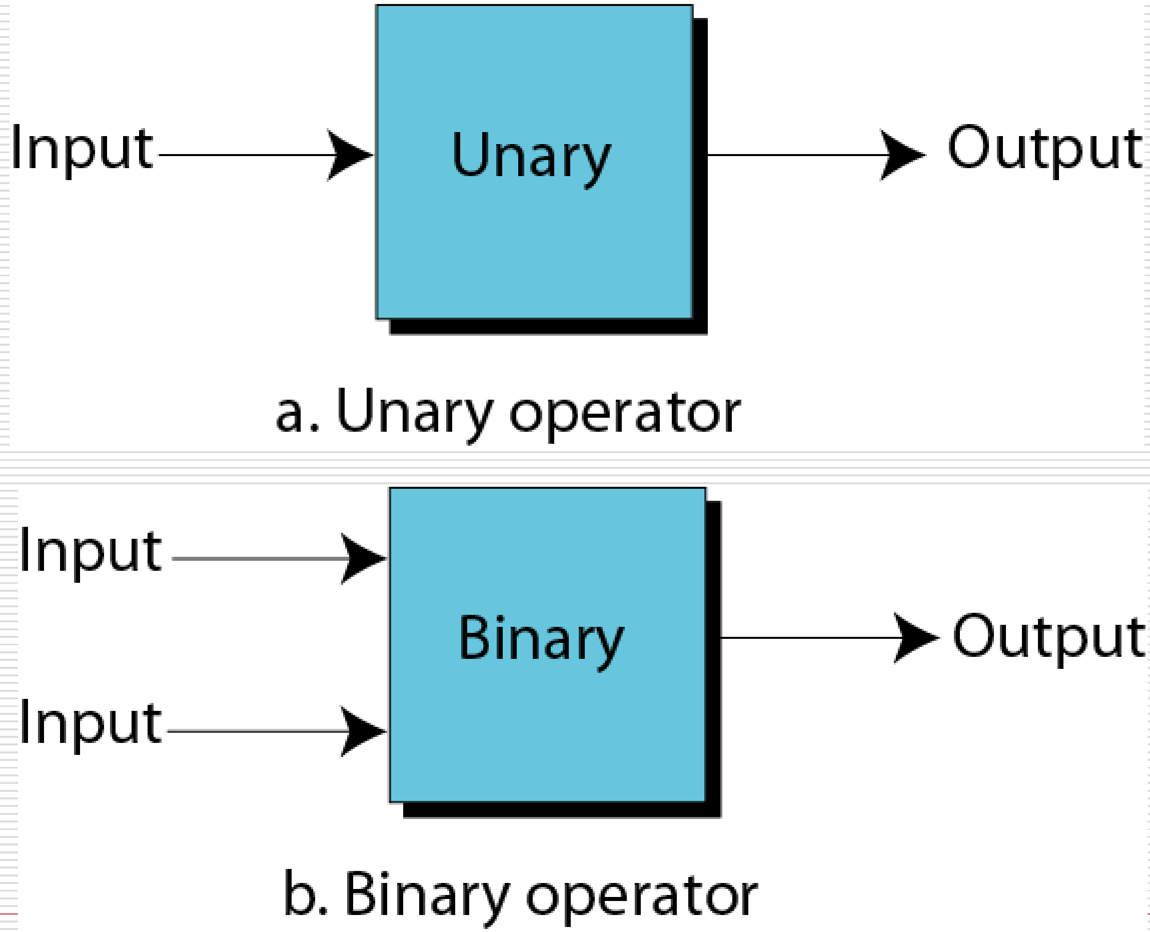
OR

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

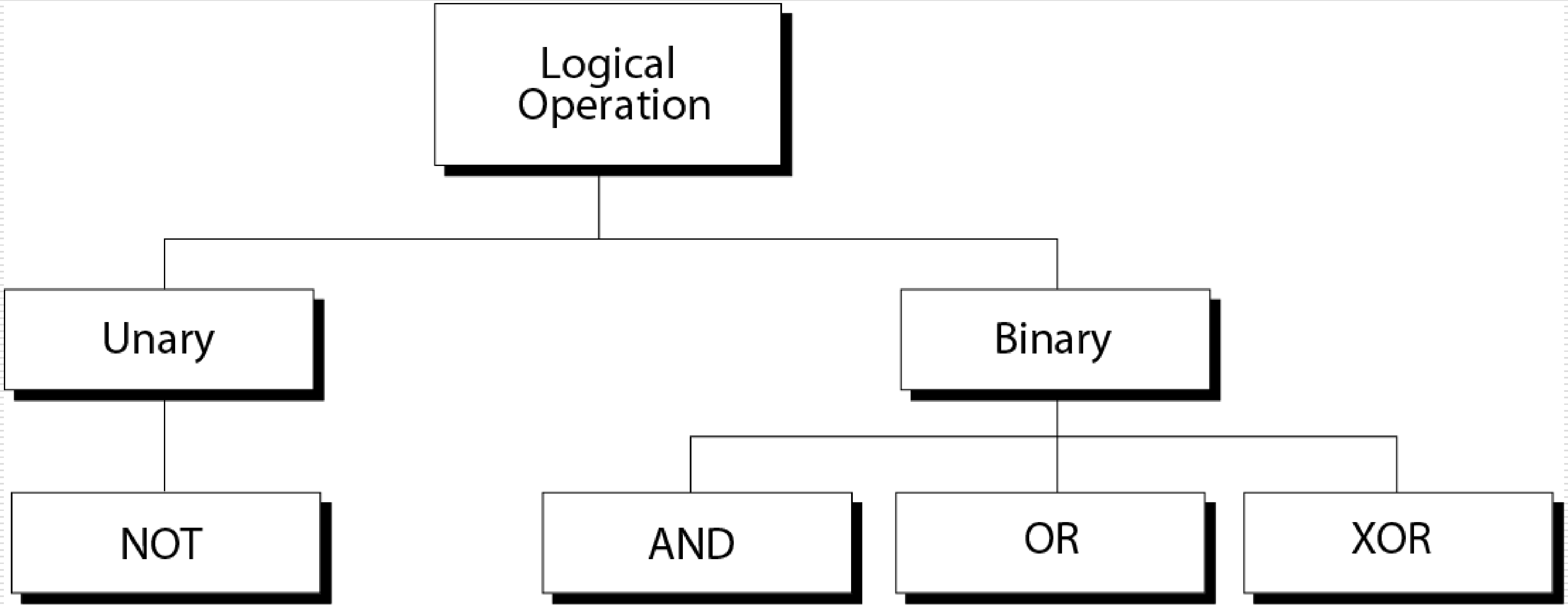
XOR

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

Unary and Binary Operations

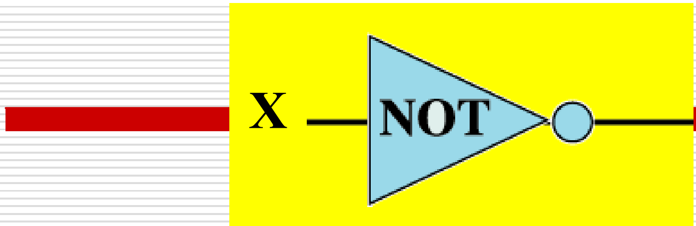


Logical Operations



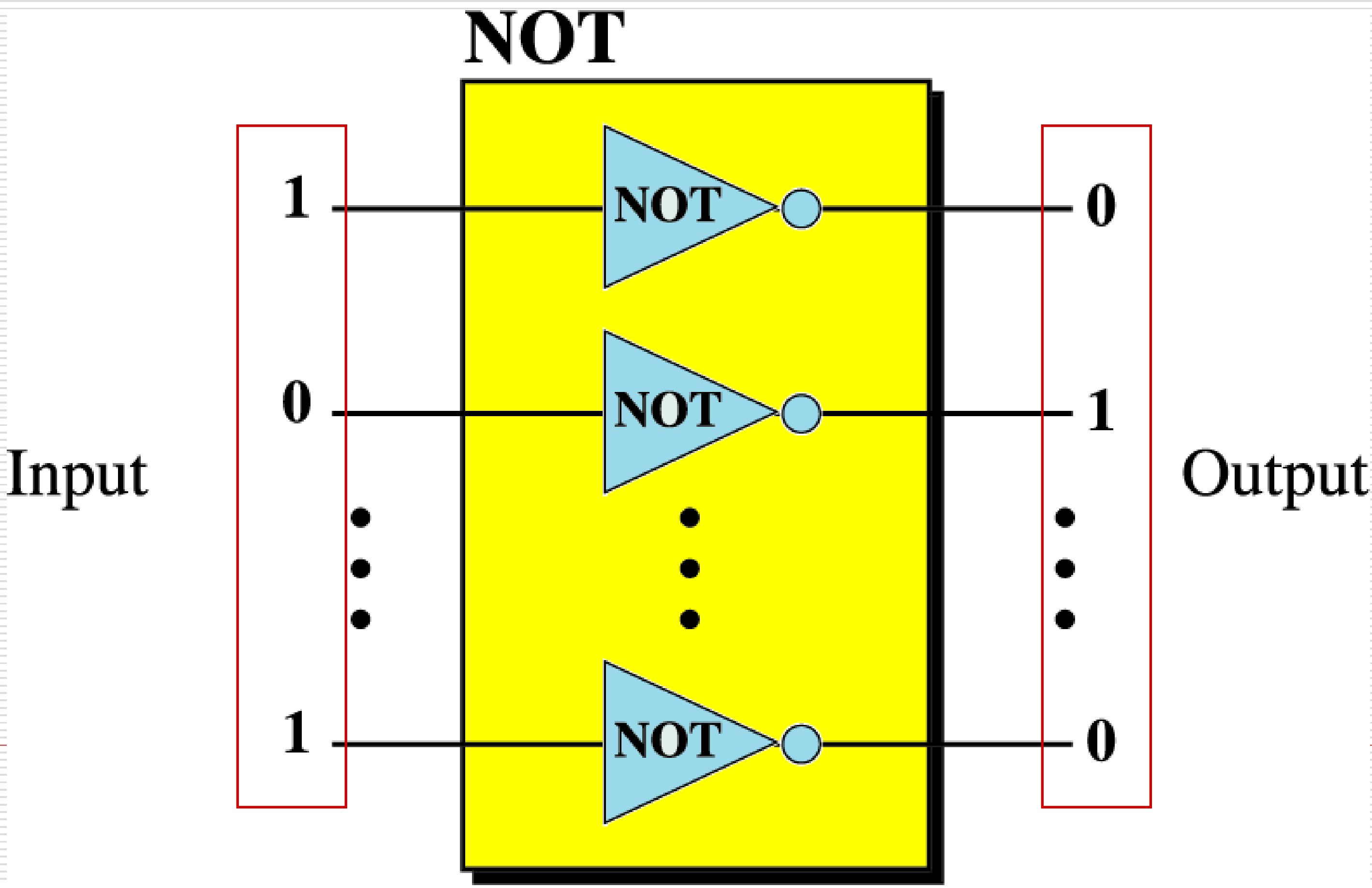
Logic Operations at Bit Level

NOT Operator

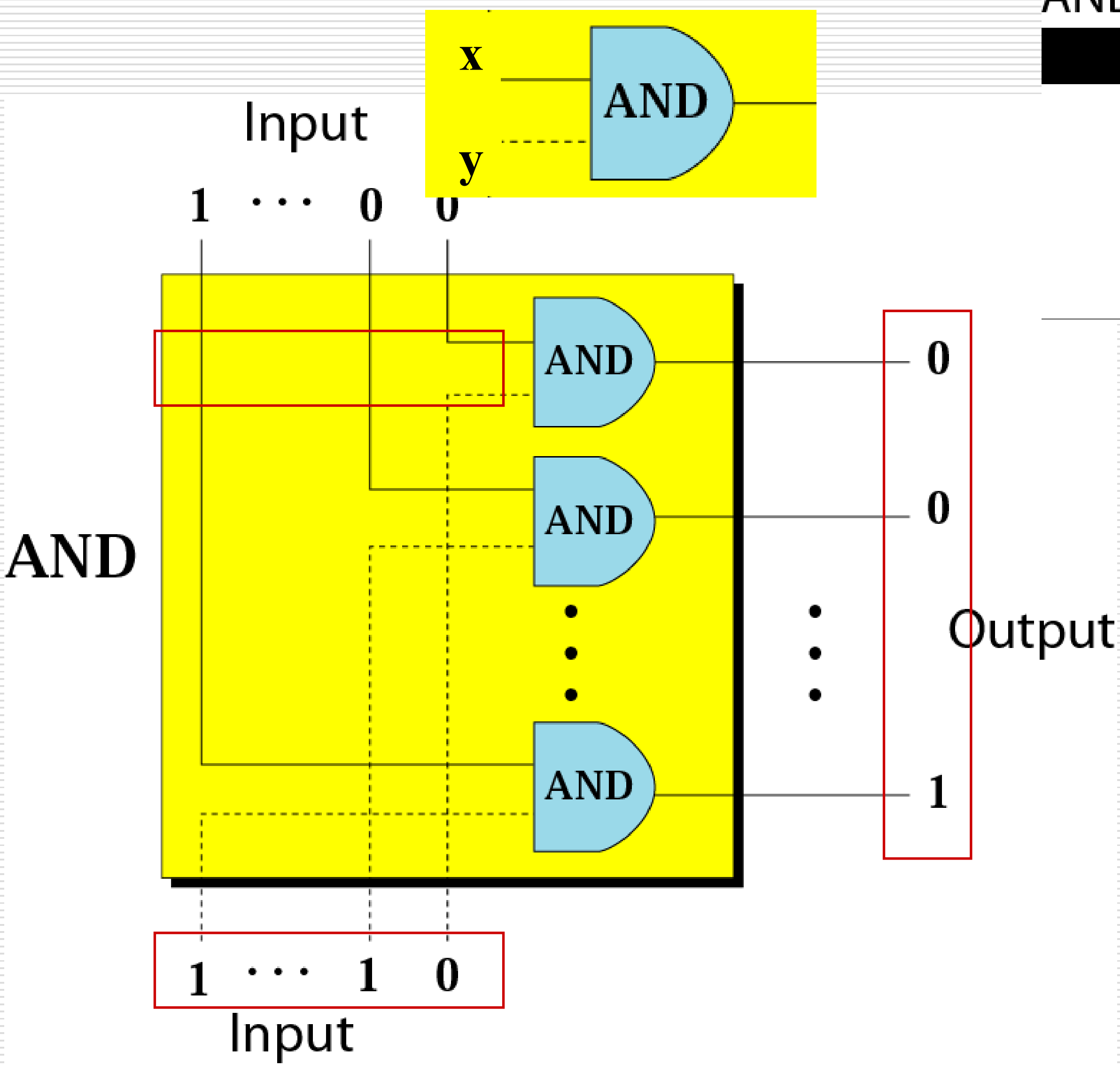


NOT

x	NOT x
0	1
1	0



AND Operator

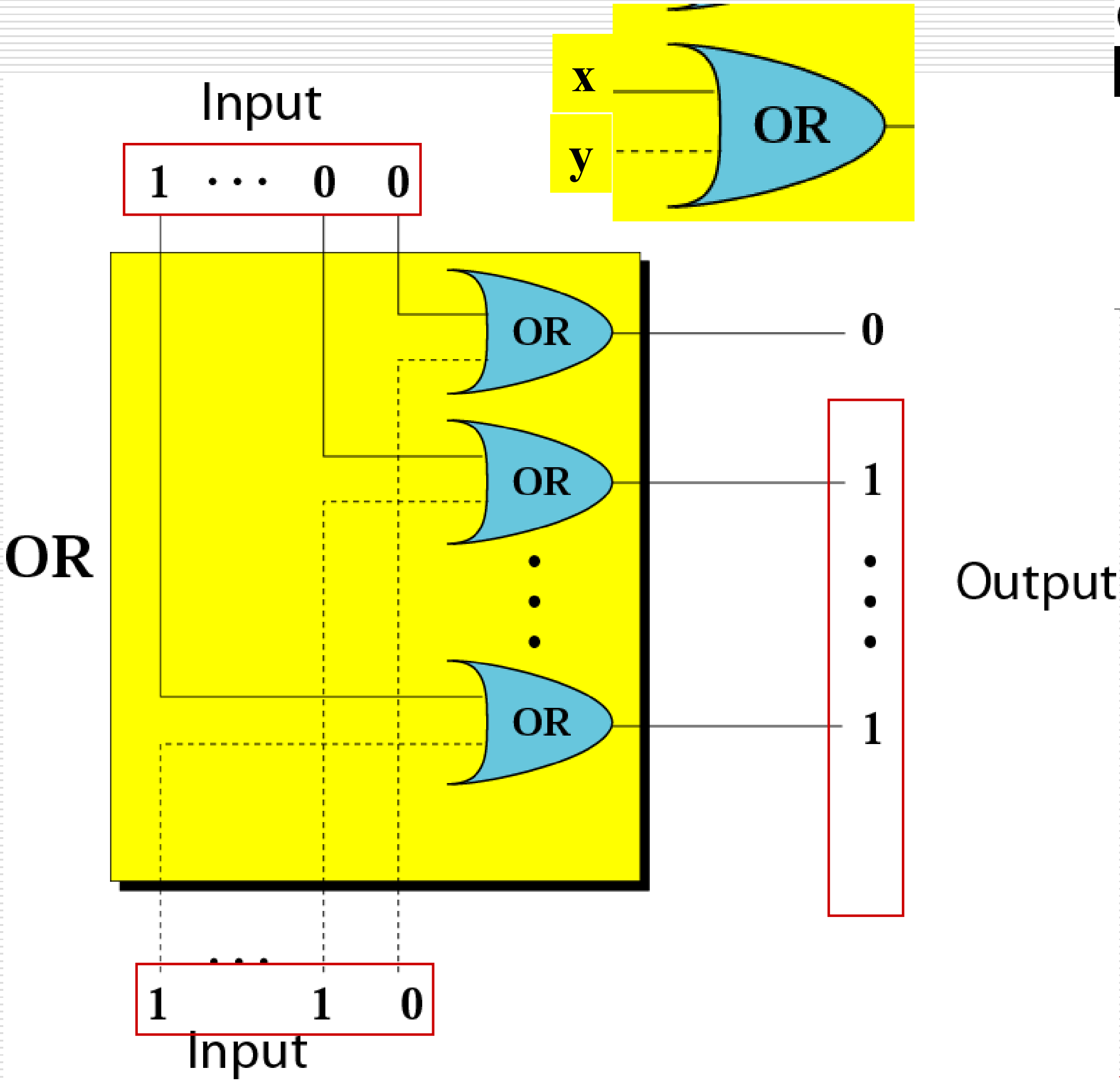


AND

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

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

OR Operator

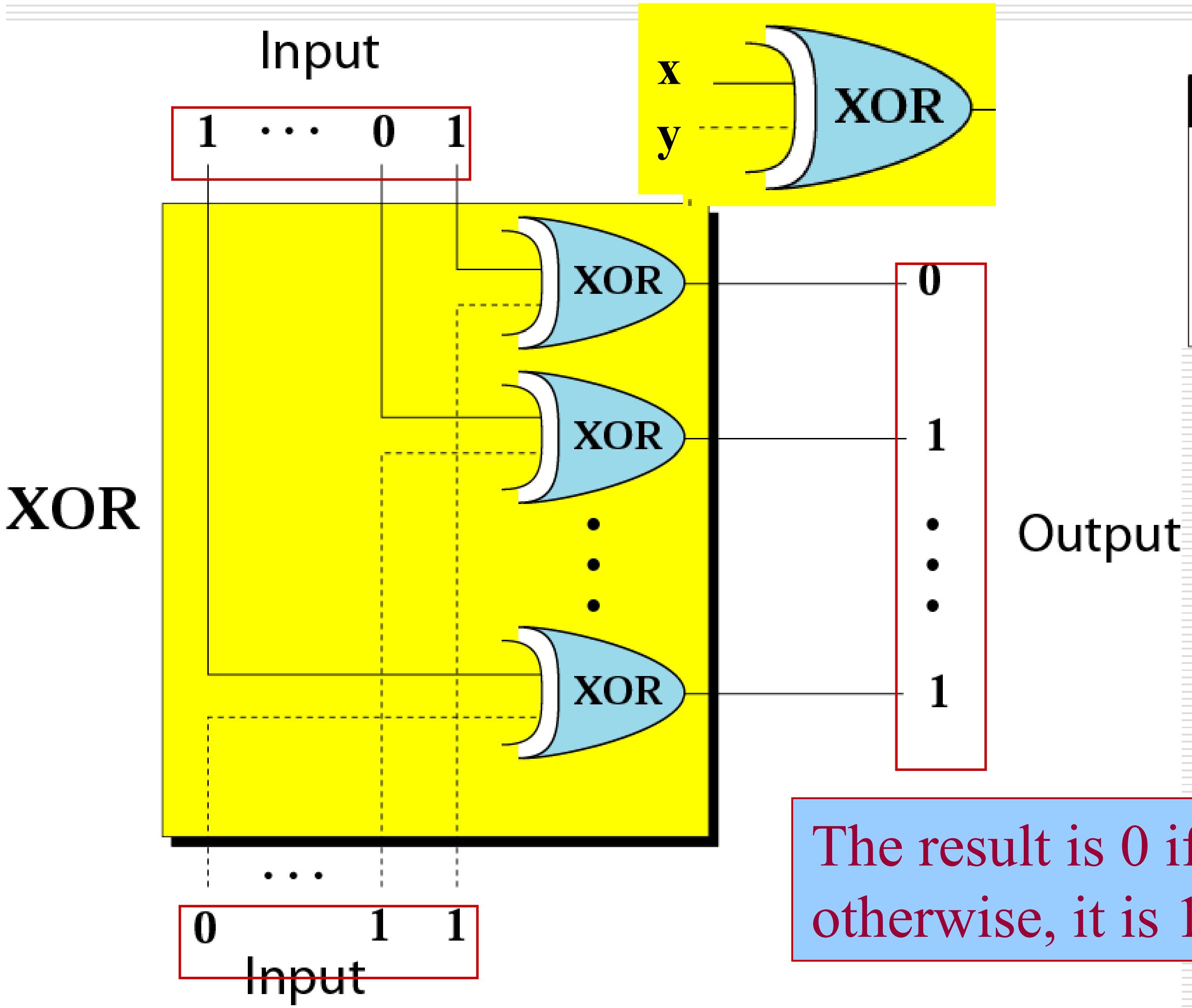


OR

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

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

XOR Operator



XOR

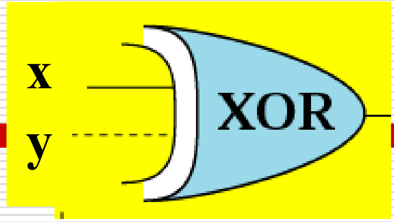
x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

The result is 0 iff both bits are equal; otherwise, it is 1.

For x = 0 or 1

1 XOR $x \rightarrow$ NOT x **x XOR 1 $\rightarrow$ NOT x**

The Application of XOR Operator

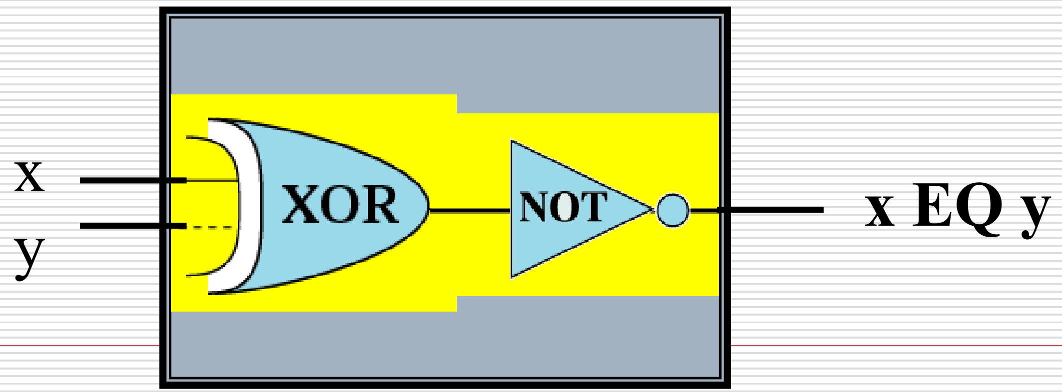


XOR

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

$$x \text{ XOR } y = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$$

EQuality



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 XOR y ↔ [x AND (NOT y)] OR [(NOT x) 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. Figure 4.2 shows these four operators with input and output patterns.

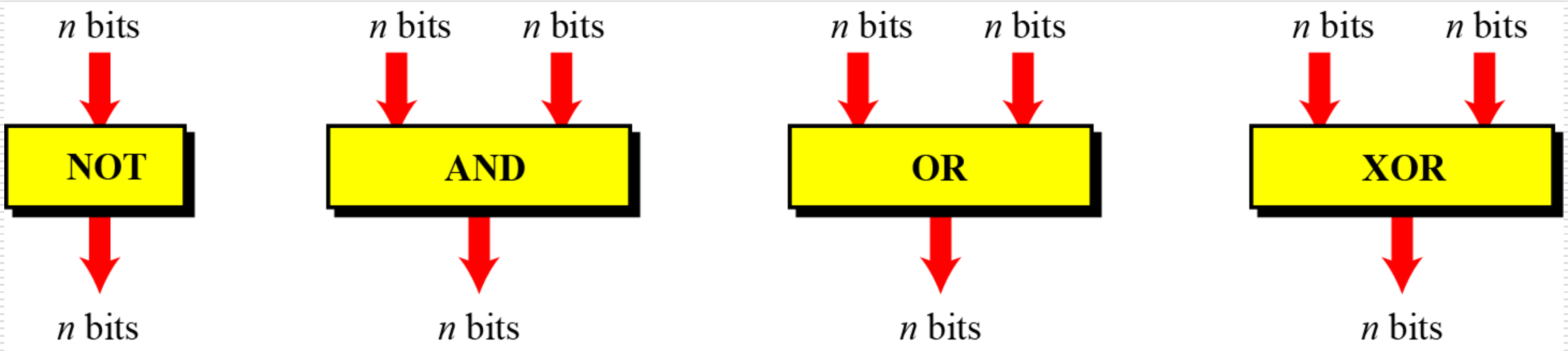


Figure 4.2 **Logic operators applied to bit patterns**

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

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

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

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).

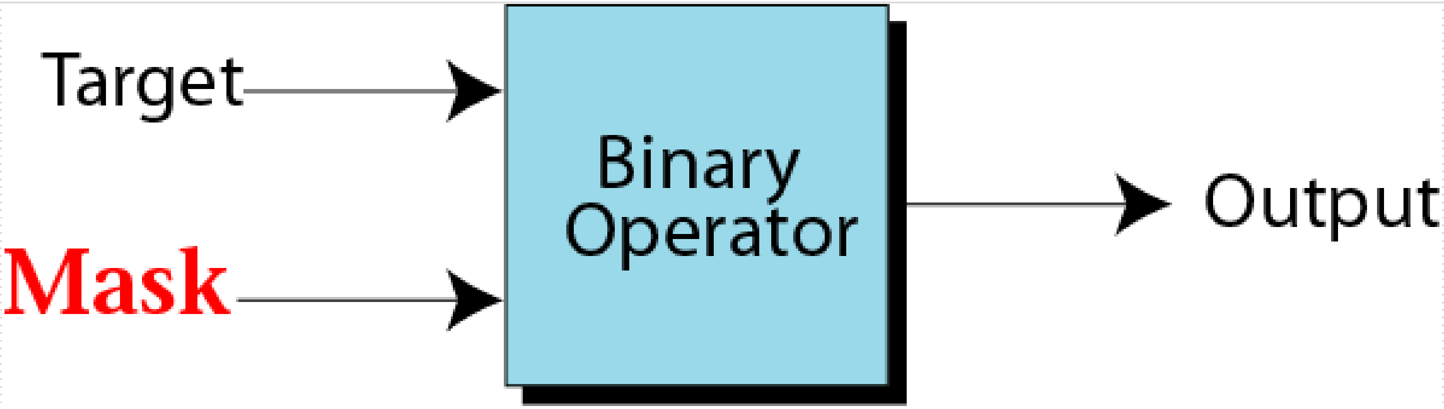
	1	0	0	1	1	0	0	1	Input 1
XOR	0	0	1	0	1	1	1	0	Input 2
	1	0	1	1	0	1	1	1	Output

Applications

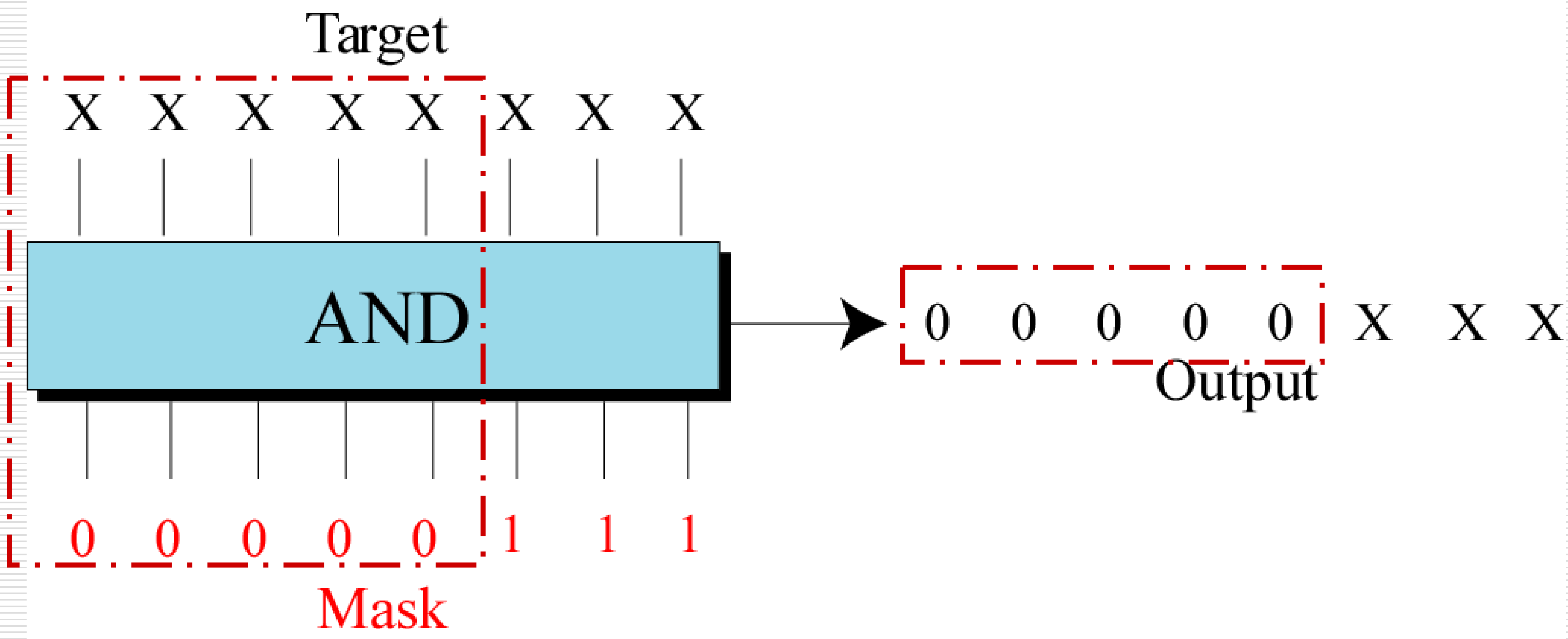
- The four logic operations can be used to modify a bit pattern.
  - ✓ Complementing (NOT)
  - ✓ Unsetting (AND)
  - ✓ Setting (OR)
  - ✓ Flipping (XOR)

Mask

- The **Target** can be modified (unset/set/reverse specific bits) by the **Binary Operator** (AND/OR/XOR) with a **Mask**.



Example of Unsetting Specific Bits



Example 4.7

Use a mask to unset (clear) the 5 leftmost bits of a pattern. Test the mask with the pattern 10100110.

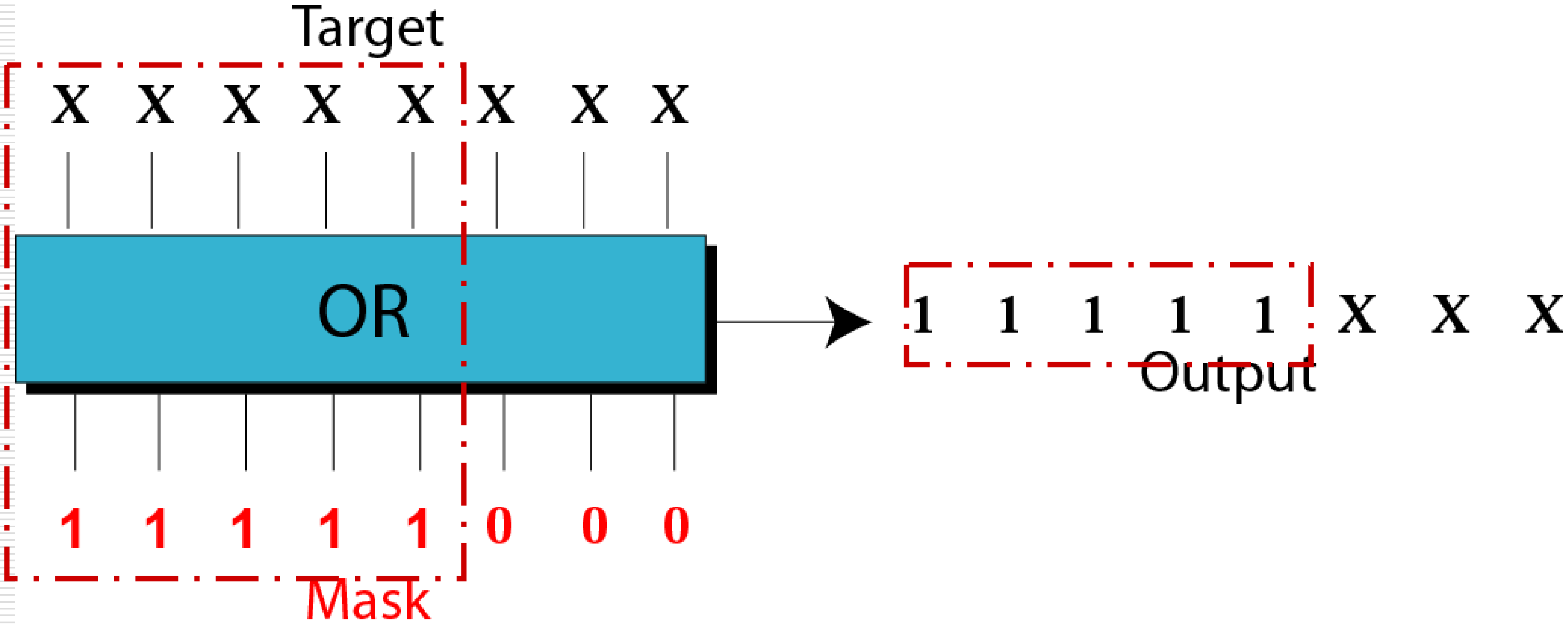
Solution

The mask is 00000111.

Target	**1 0 1 0 0**	**1 1 0**	***AND***
Mask	**0 0 0 0 0**	**1 1 1**	

Result	**0 0 0 0 0**	**1 1 0**	

Example of Setting Specific Bits



Example 4.8

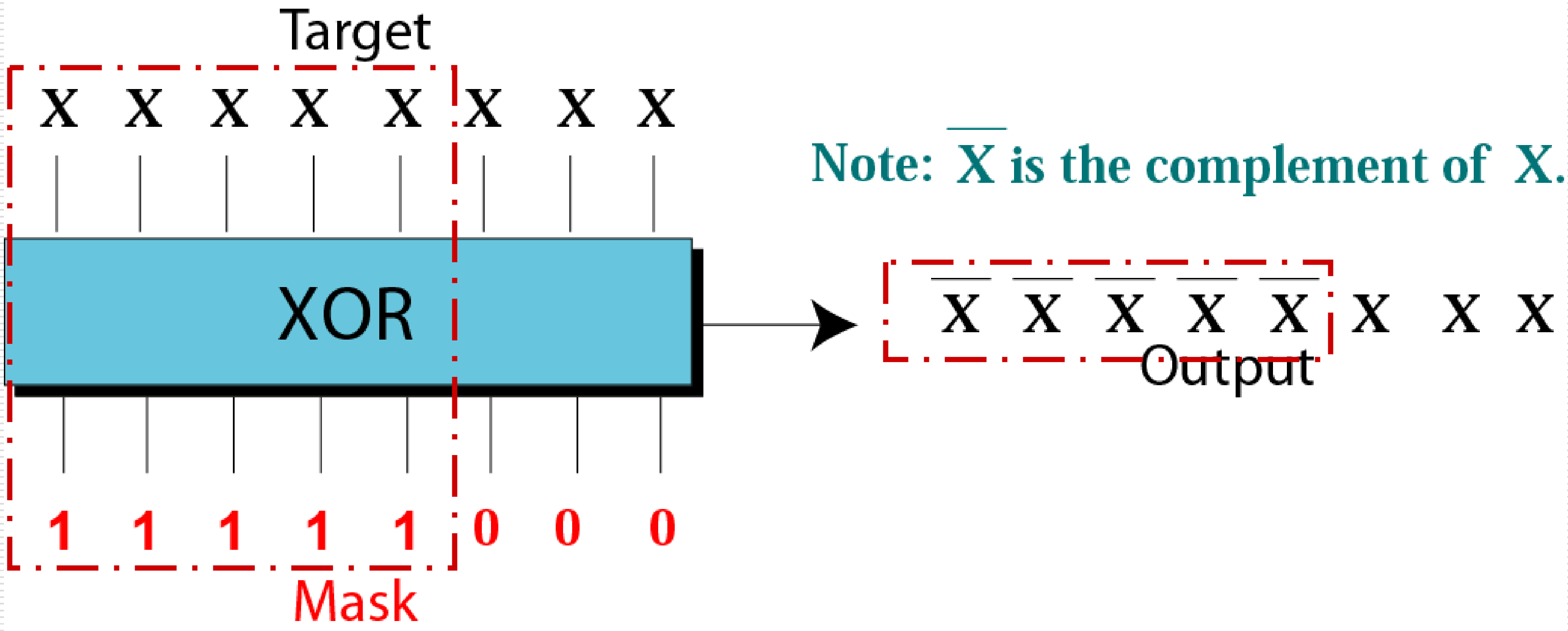
Use a mask to set the 5 leftmost bits of a pattern. Test the mask with the pattern 10100110.

Solution

The mask is 11111000.

Target	1 0 1 0 0	1 1 0	***OR***
Mask	1 1 1 1 1	0 0 0	
Result	1 1 1 1 1	1 1 0	

Example of Flipping Specific Bits



Note: $\overline{X}$ is the complement of X.

Example 4.9

Use a mask to flip the 5 leftmost bits of a pattern. Test the mask with the pattern 10100110.

Solution

Target	1 0 1 0 0	1 1 0	***XOR***
Mask	1 1 1 1 1	0 0 0	
Result	0 1 0 1 1	1 1 0	

4.2

SHIFT OPERATIONS

Overview

- **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 (1)

A logical shift operation is applied to a pattern that does not represent a signed number.
We distinguish two types of logical shift operations, as described below:

- ❑ **Logical Shift**

- ❑ **Logical Circular Shift (Rotate)**

Logical Shift Operations (2)

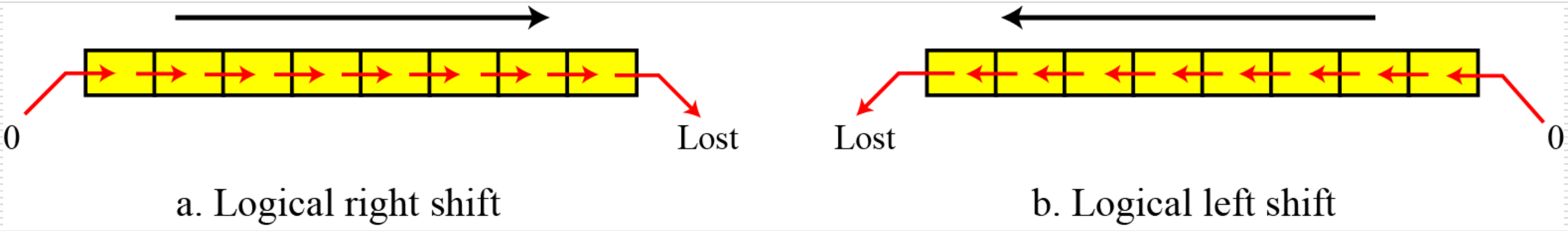


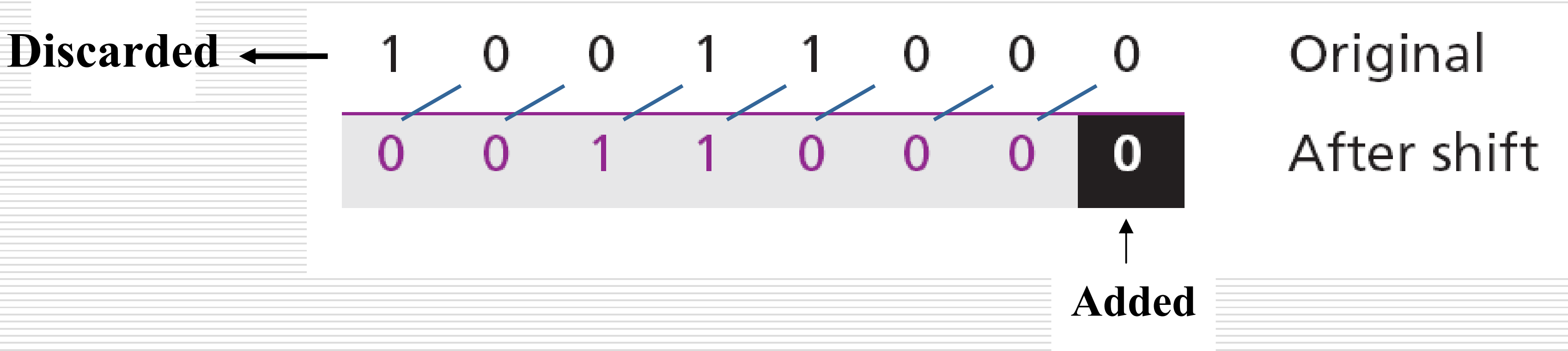
Figure 4.3 **Logical Shift Operations**

Example 4.10

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

Solution

The leftmost bit is lost and a 0 is inserted as the rightmost bit.



Circular Shift Operations

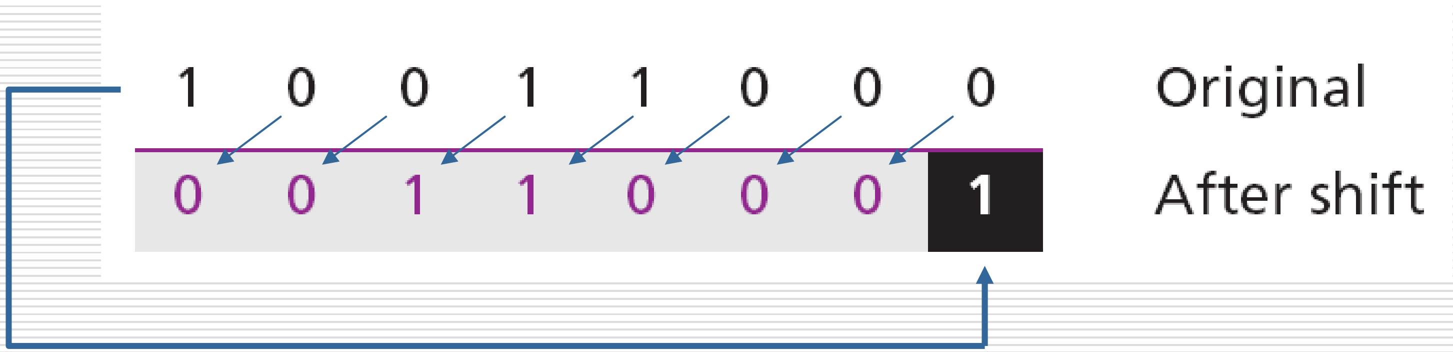
Figure 4.4 **Circular Shift Operations**

Example 4.11

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

Solution

The leftmost bit is circulated and becomes the rightmost bit.



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.

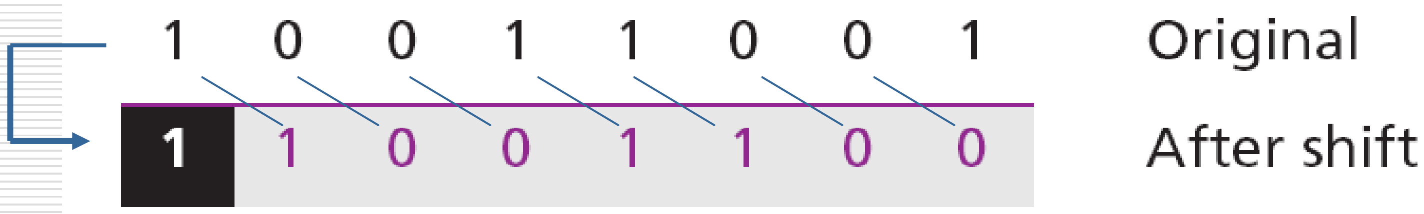
Figure 4.5 **Arithmetic Shift Operations**

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 leftmost bit is retained and also copied to its right neighbor bit.



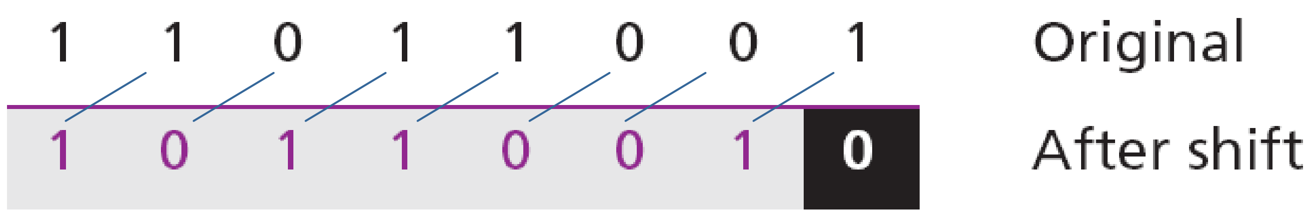
The original number was −103 and the new number is −52, which is the result of dividing −103 by 2 truncated to the smaller integer.

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 leftmost bit is lost and a 0 is inserted as the rightmost bit.



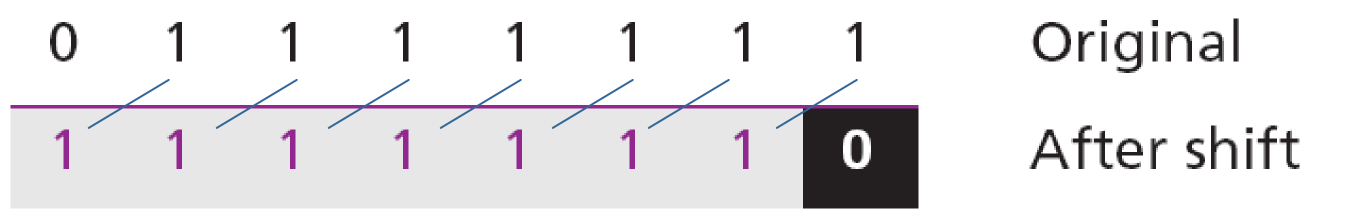
The original number was −39 and the new number is −78. The original number is multiplied by two. The operation is valid because no underflow occurred.

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 leftmost bit is lost and a 0 is inserted as the rightmost bit.



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 $127 \times 2 = 254$ 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 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 on Integers

- We focus only on addition and subtraction because
  - Multiplication is just repeated addition and
  - Division is just repeated subtraction.

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:

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

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

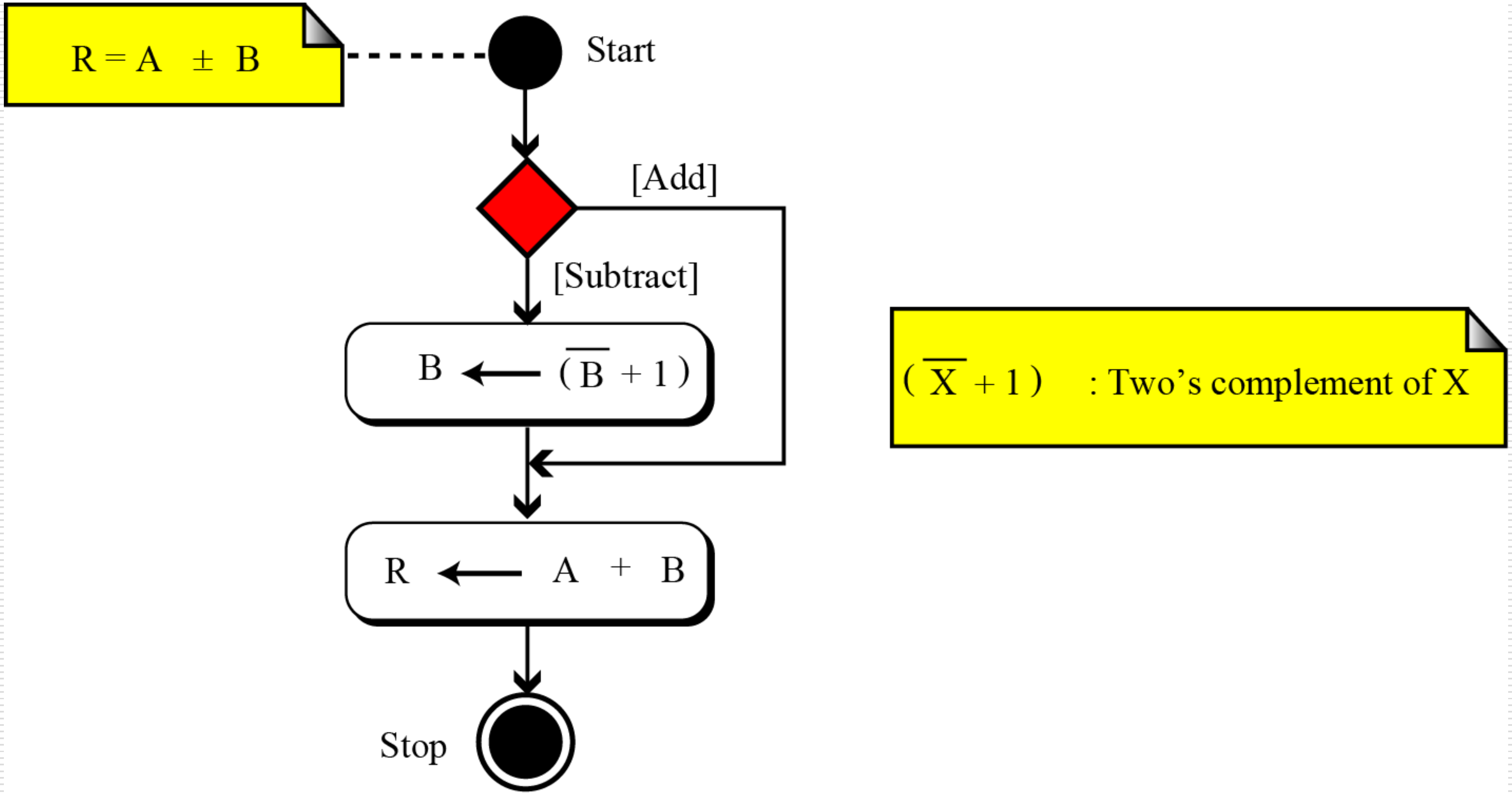


Figure 4.6 **Addition and Subtraction of Integers in Two's Complement Format**

Example 4.16

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

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

Solution

The operation is adding. A is added to B and the result is stored in R. (+17) + (+22) = (+39).

			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

Example 4.17

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

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

Solution

The operation is adding. A is added to B and the result is stored in R. (+24) + (–17) = (+7).

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

Example 4.18

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

A = $(00011000)_2$ **B = $(11101111)_2$**

Solution

The operation is subtracting. A is added to $(\overline{B} + 1)$ and the result is stored in R. (+24) – (–17) = (+41).

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

Example 4.19

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

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

Solution

The operation is subtracting. A is added to $(\overline{B} + 1)$ and the result is stored in R. (−35) − (+20) = (−55).

	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

Example 4.20

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

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

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.

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.

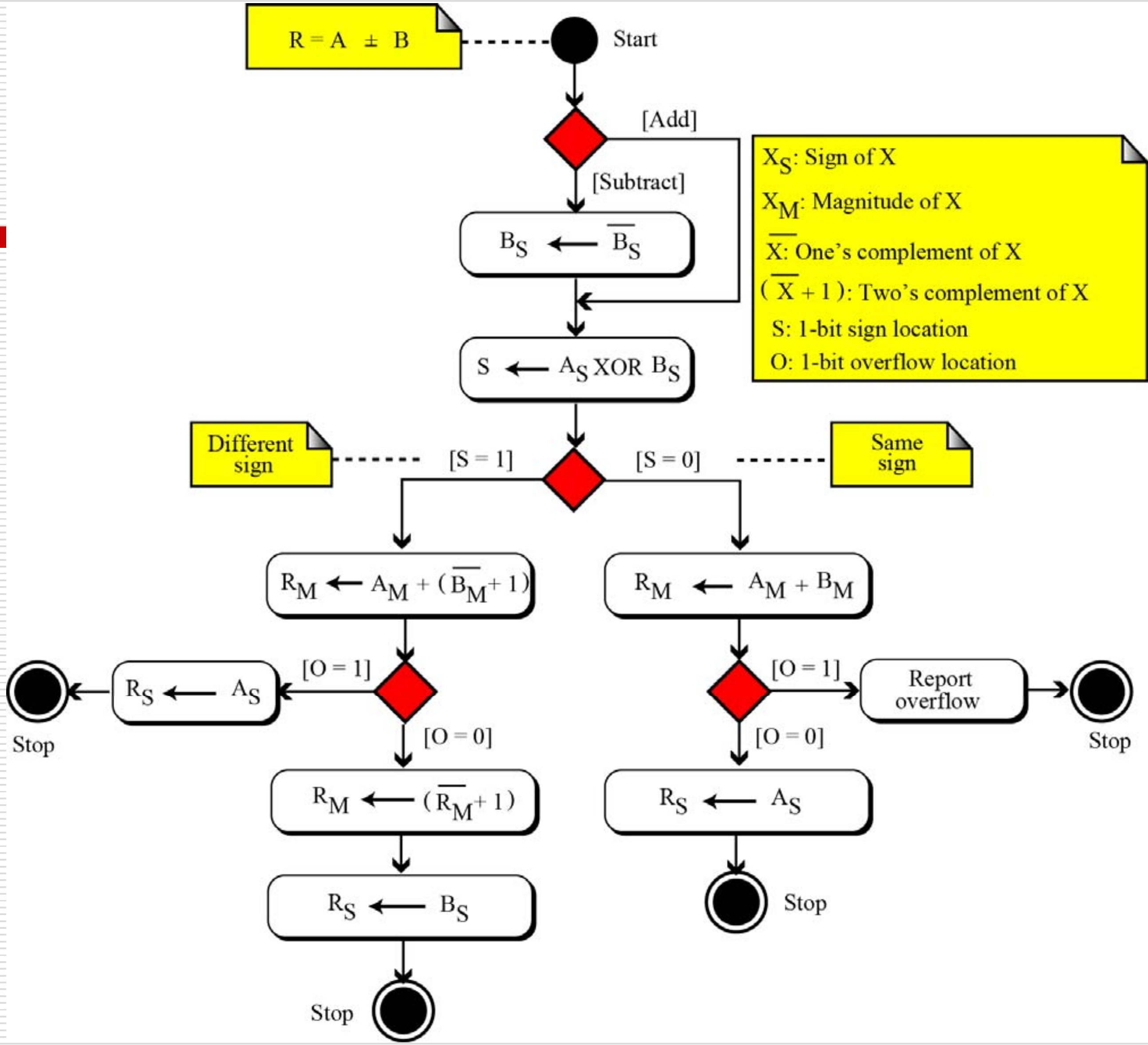


Figure 4.7 Addition and Subtraction of Integers in Sign-and-Magnitude Format

Example 4.22

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

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

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. (+17) + (−22) = (−5).

		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	$R_M = (\overline{R_M} + 1)$

Example 4.23

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

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

Solution

The operation is subtracting: $S_B = \overline{S_B}$. $S = A_S$ 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. $(-81) - (-22) = (-59)$.

		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

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. We 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).

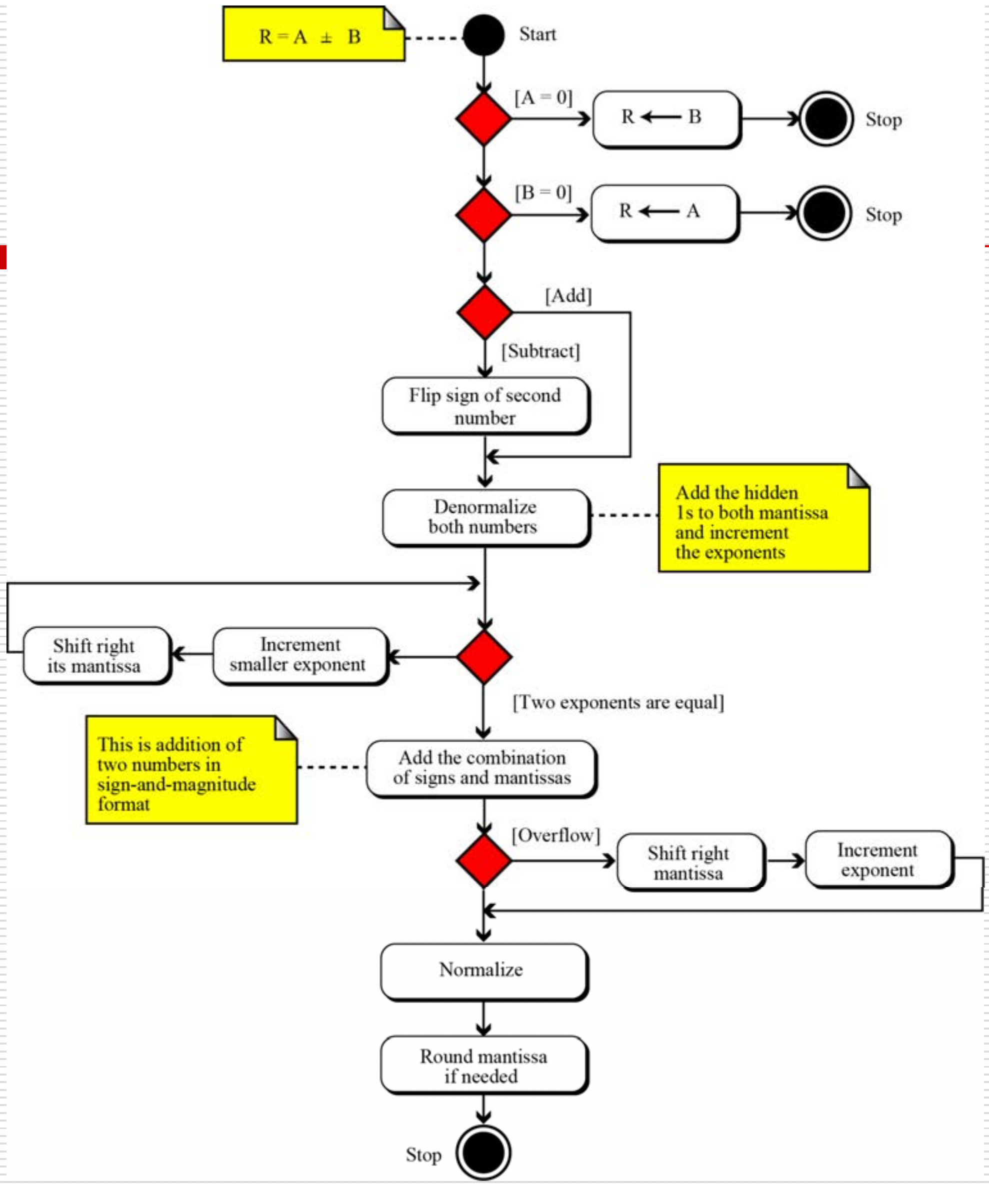


Figure 4.8 **Addition and Subtraction of Reals in Floating-Point Format**

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).

	S	E	M
A	0	10000001	01110000000000000000000
B	0	10000110	01000011110000000000000

Example 4.24 (Continued)

The first few steps in the UML diagram (Figure 4.8) are not needed. We de-normalize the numbers by adding the hidden 1s to the mantissa and incrementing the exponent. Now both de-normalized 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

Example 4.24 (Continued)

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	10100111101000000000000

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

De-normalization results in:

	S	E	Denormalized M
A	0	10000010	101110000000000000000000
B	1	10000010	111000001100000000000000

Example 4.25 (Continued)

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	00101000110000000000000

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

	S	E	M
R	1	01111111	01000110000000000000000

Example 4.25 (Continued)

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.