

IT111 - Introduction to Information Technology

Bachelor of Science in Information Technology

LECTURE 5 OPERATION ON DATA

Prof. Lamiaa Fattouh Ibrahim
Everyone Smart University
Faculty of Informatics
Spring 2023

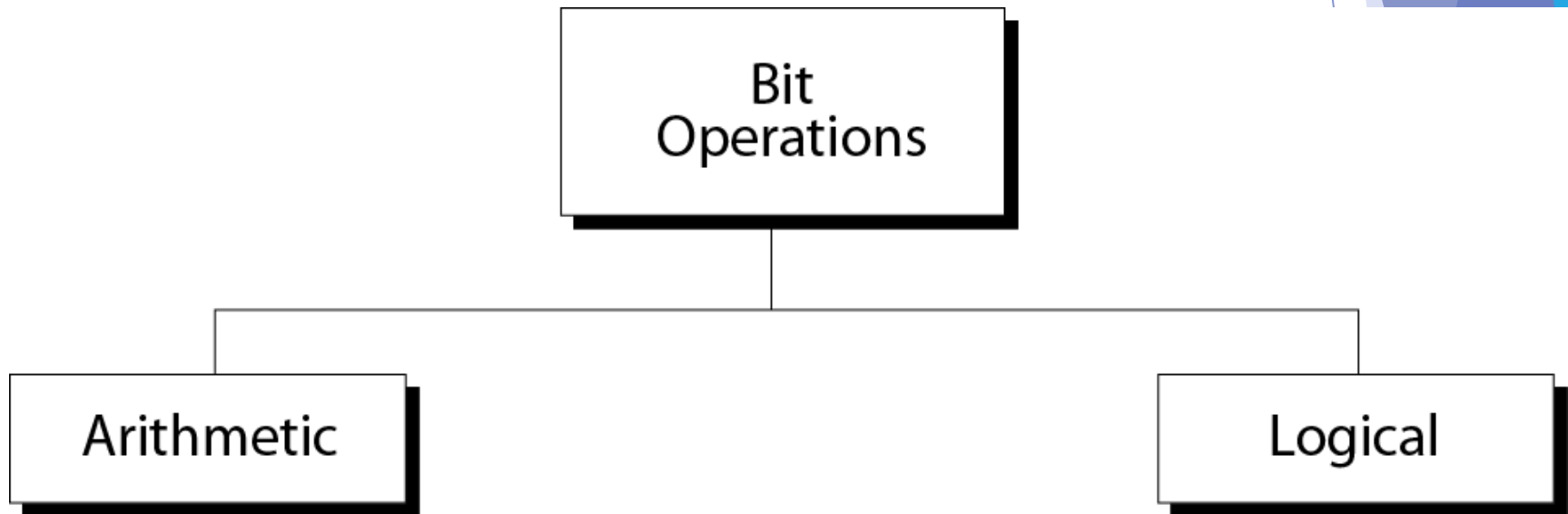
Objectives

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

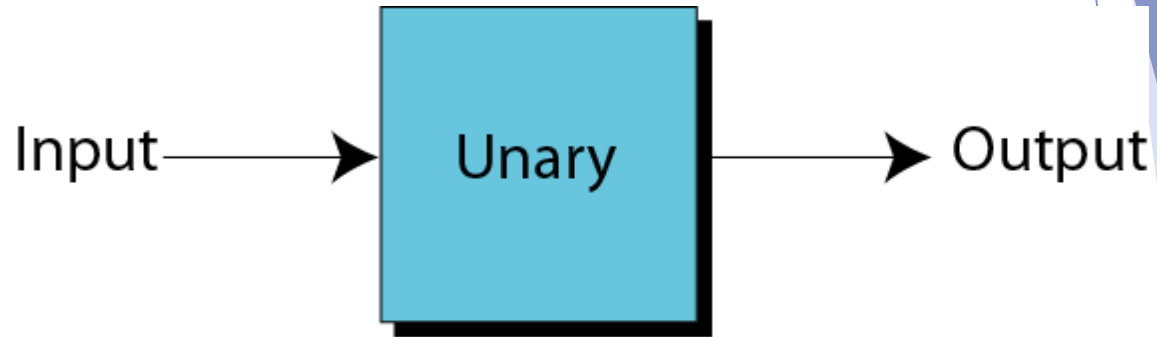
- ☐ List the three categories of operations performed on data.
- ☐ Perform unary and binary logic operations on bit patterns.
- ☐ Distinguish between logic shift operations and arithmetic shift operations.
- ☐ Perform logic shift operations on bit patterns.
- ☐ Understand some applications of logic and shift operations such as setting, unsetting, and flipping bits.
- ☐ Perform arithmetic shift operation on integers when they are stored in two's complement format.



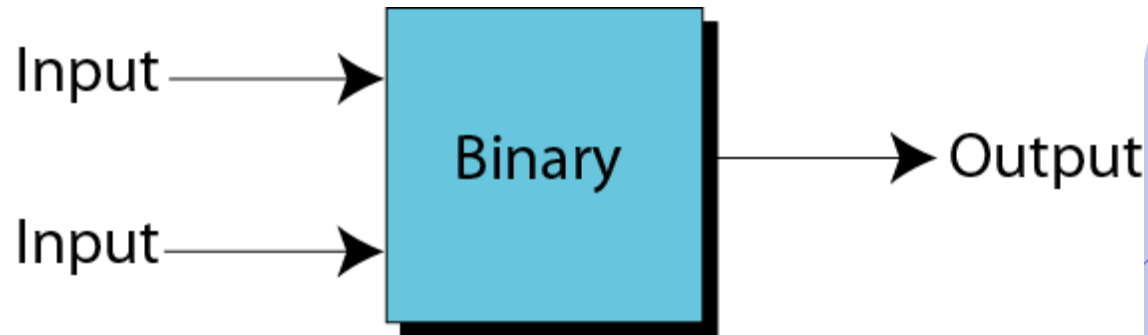
Operations on bits



Unary and binary operations

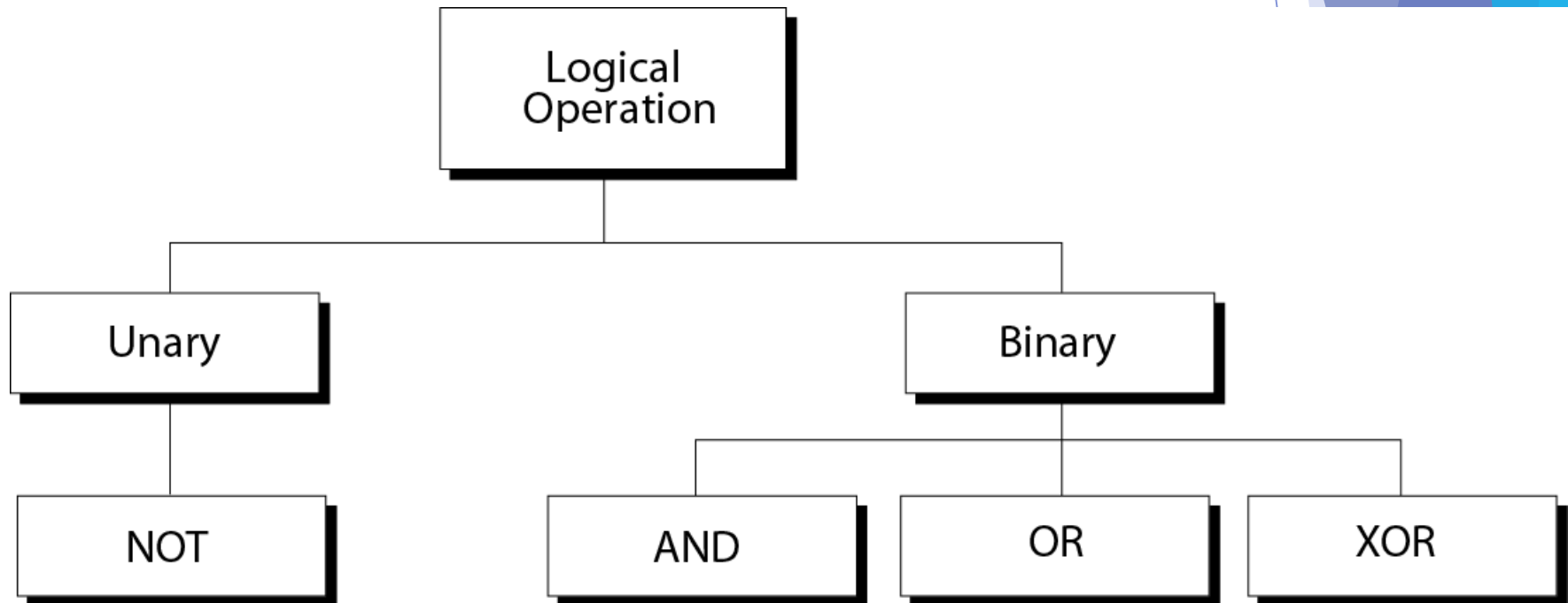


a. Unary operator



b. Binary operator

Logical operations



4-1 LOGIC OPERATIONS

In Chapter 3 we discussed the fact that data inside a computer is stored as patterns of bits.

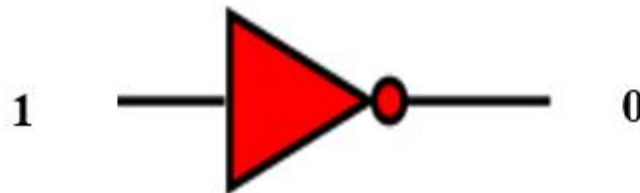
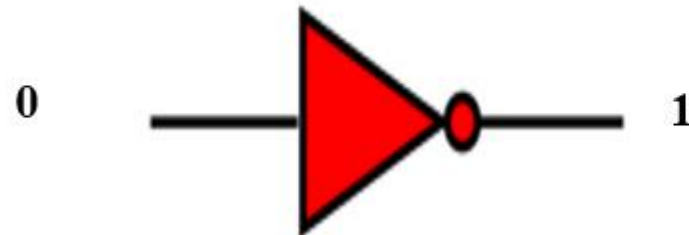
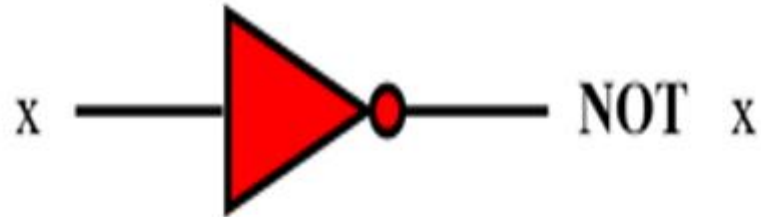
Logic operations refer to those operations that apply the same basic operation on individual bits of a pattern, or on two corresponding bits in two patterns.

This means that we can define logic operations:

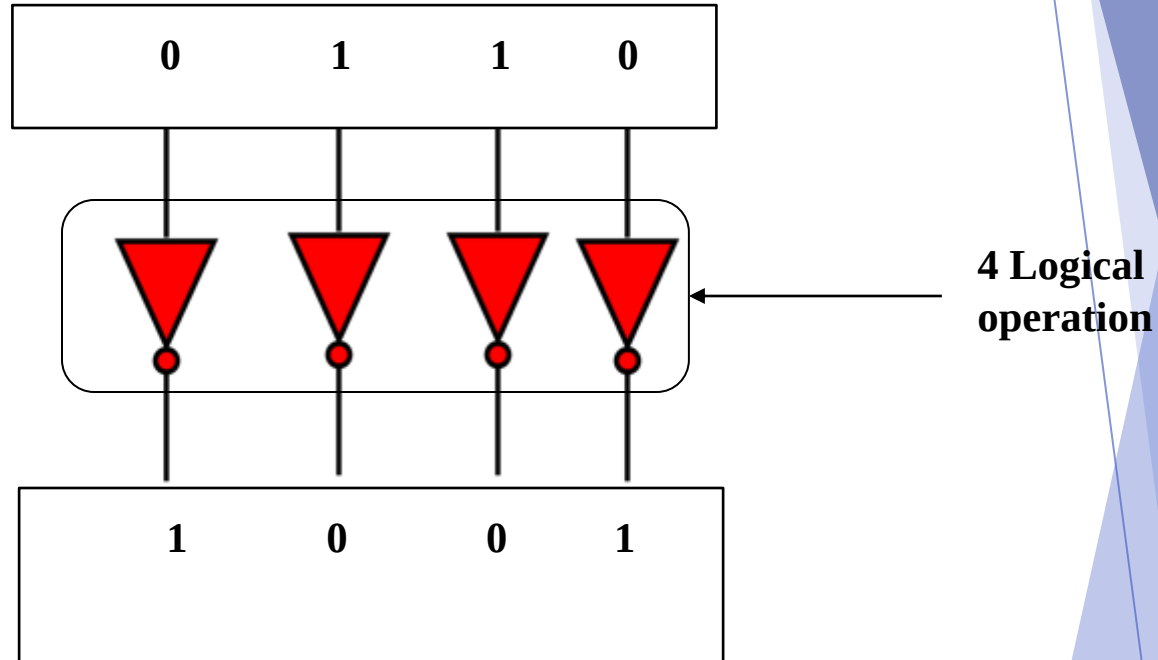
- at the **bit level**
- at the **pattern level**



Logic operations at bit level



4.1.2 Logic operations at 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.
- **A Boolean expression** is a logical statement that is either TRUE or FALSE.
- Boolean algebra and its application to building logic circuits in computers are briefly discussed in Appendix E.



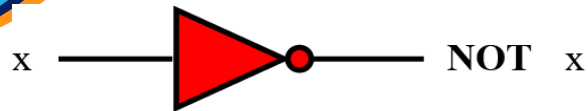
Boolean algebra

Boolean algebra is a branch of algebra. It differs from elementary algebra in two ways.

- First, the values of the variables are the truth values *true* and *false*, usually denoted 1 and 0, whereas in elementary algebra the values of the variables are numbers.
- Second, Boolean algebra uses logical operators such as
 - conjunction (*and*) denoted as $\wedge$,
 - disjunction (*or*) denoted as $\vee$,
 - negation (*not*) denoted as $\neg$.
- Elementary algebra, on the other hand, uses arithmetic operators such as addition, multiplication, subtraction and division.
- Boolean algebra is therefore a formal way of describing logical operations, in the same way that elementary algebra describes numerical operations.

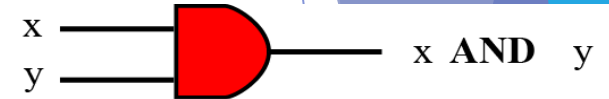


Figure 4.1 Logic operations at the bit level



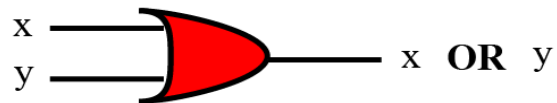
NOT

x	NOT x
0	1
1	0



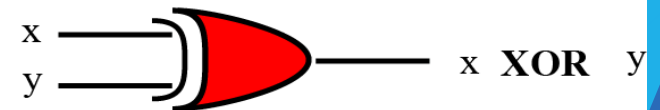
AND

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 $x \text{ AND } 0 \rightarrow 0$ $0 \text{ AND } x \rightarrow 0$



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.

For $x = 0$ or 1

$x \text{ OR } 1 \rightarrow 1$

$1 \text{ OR } x \rightarrow 1$

XOR

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

For $x = 0$ or 1

$1 \text{ XOR } x \rightarrow \text{NOT } x$

$1 \text{ XOR } x \rightarrow \text{NOT } x$

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

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

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

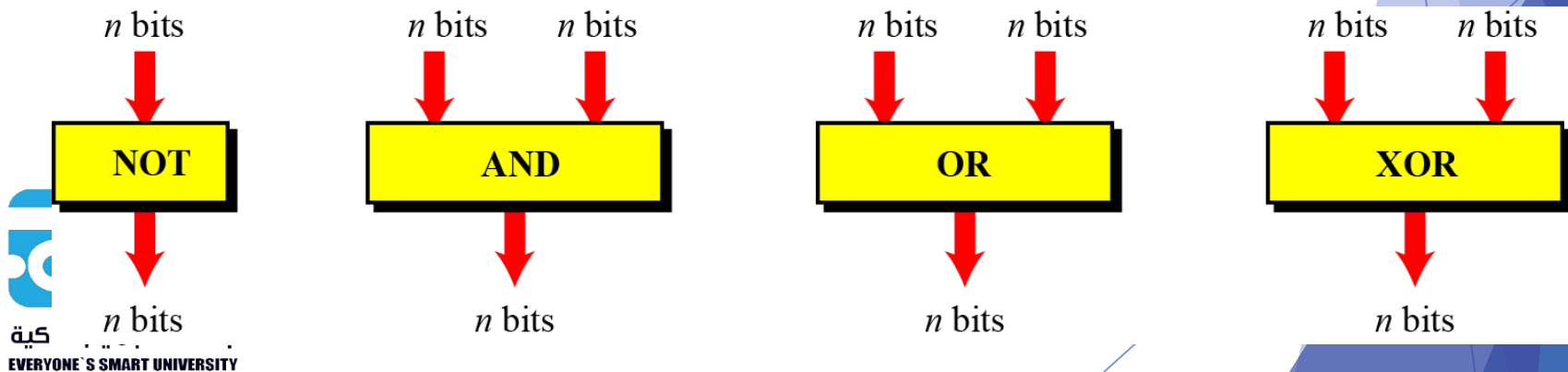
$$x \text{ XOR } y \leftrightarrow [x \text{ AND } (\text{NOT } y)] \text{ OR } [(\text{NOT } x) \text{ AND } y]$$

x	y	$x \text{ XOR } y$	$[x \text{ AND } (\text{NOT } y)] \text{ OR } [(\text{NOT } x) \text{ AND } y]$
0	0	0	$((0 \text{ AND } 1) \text{ OR } (1 \text{ AND } 0)) = 0$
0	1	1	1
1	0	1	1
1	1	0	0

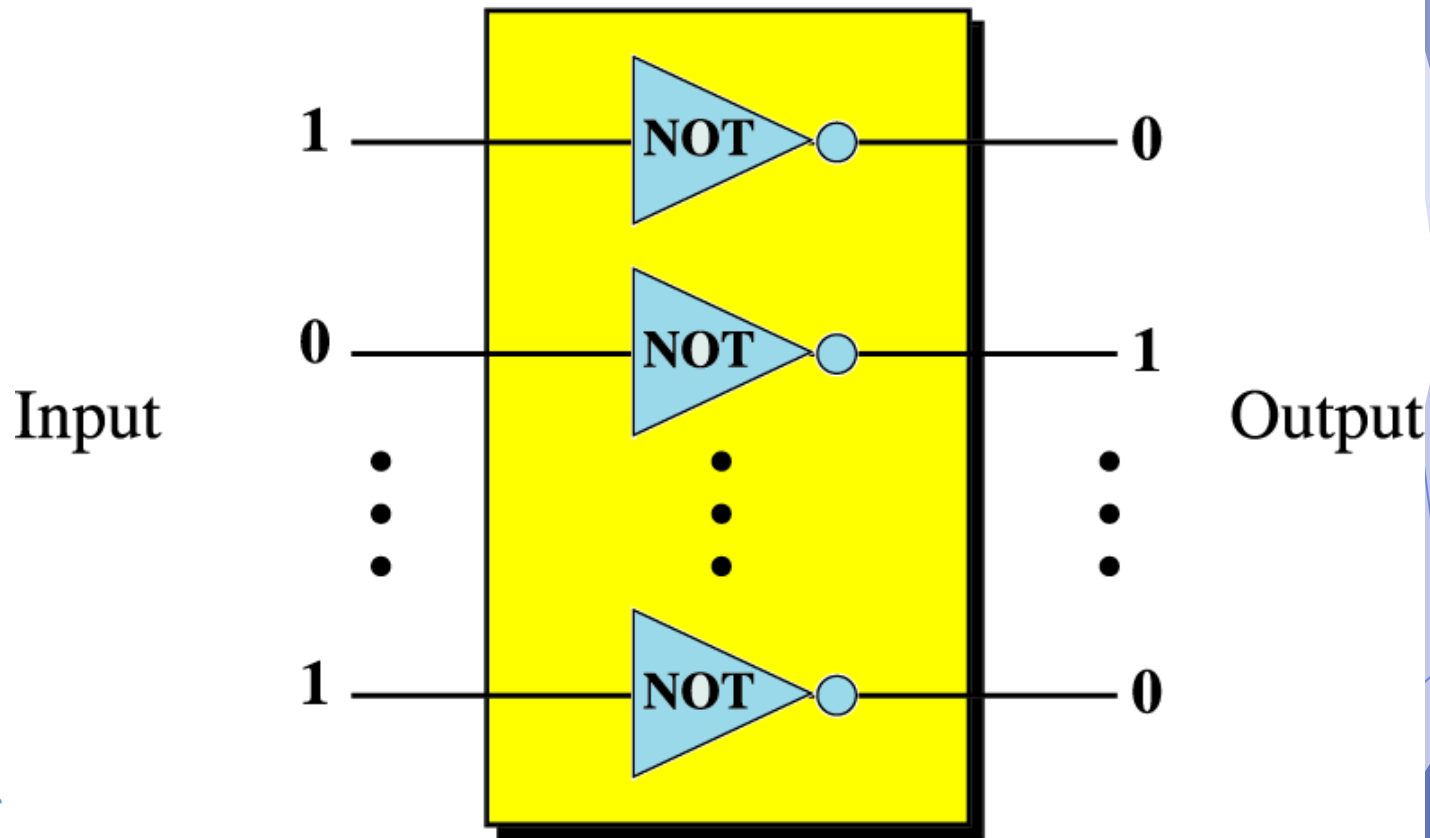
Logic operations at pattern level

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



NOT



Example 4.3

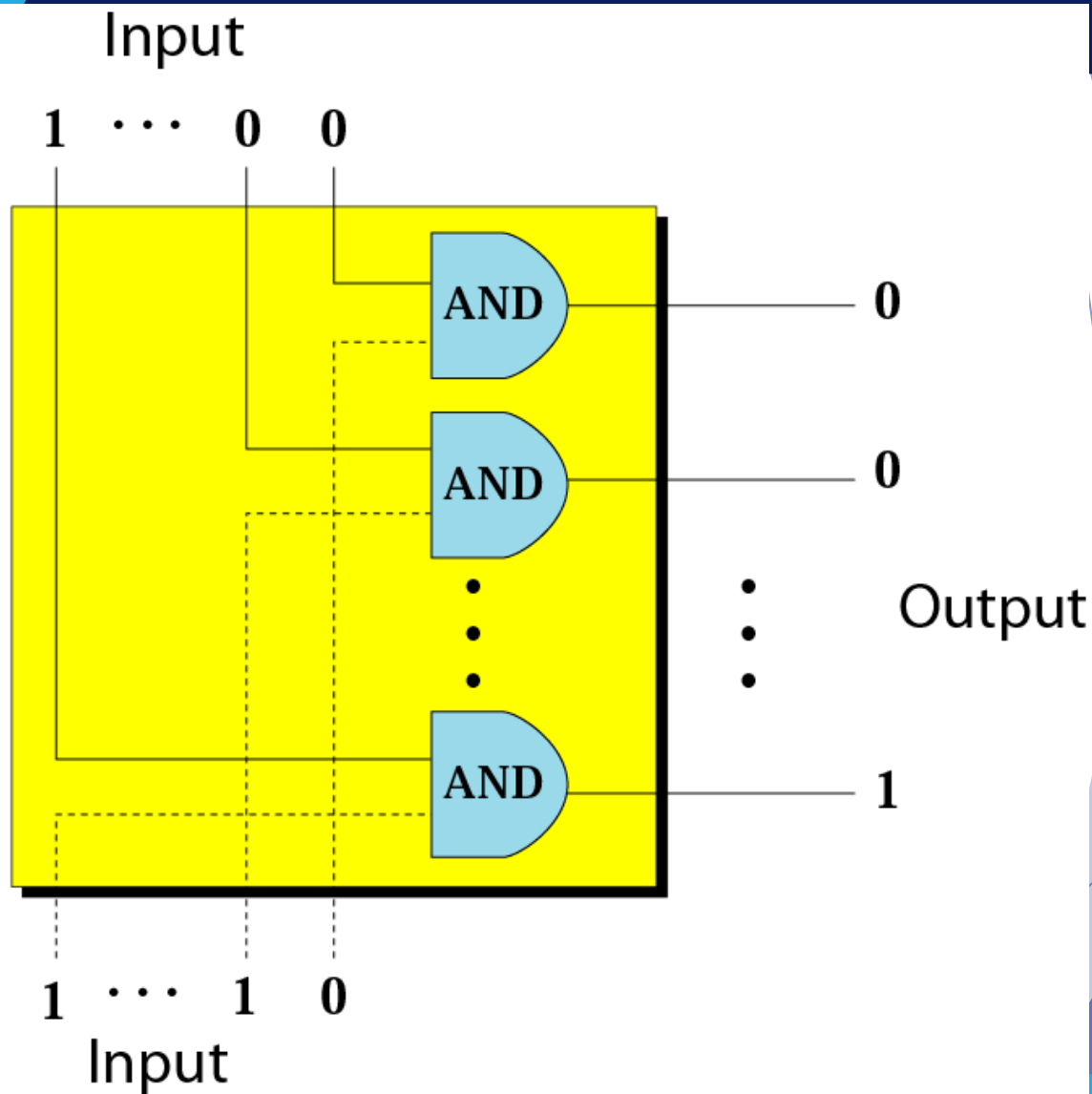
Use the NOT operator on the bit pattern 10011000.

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

AND



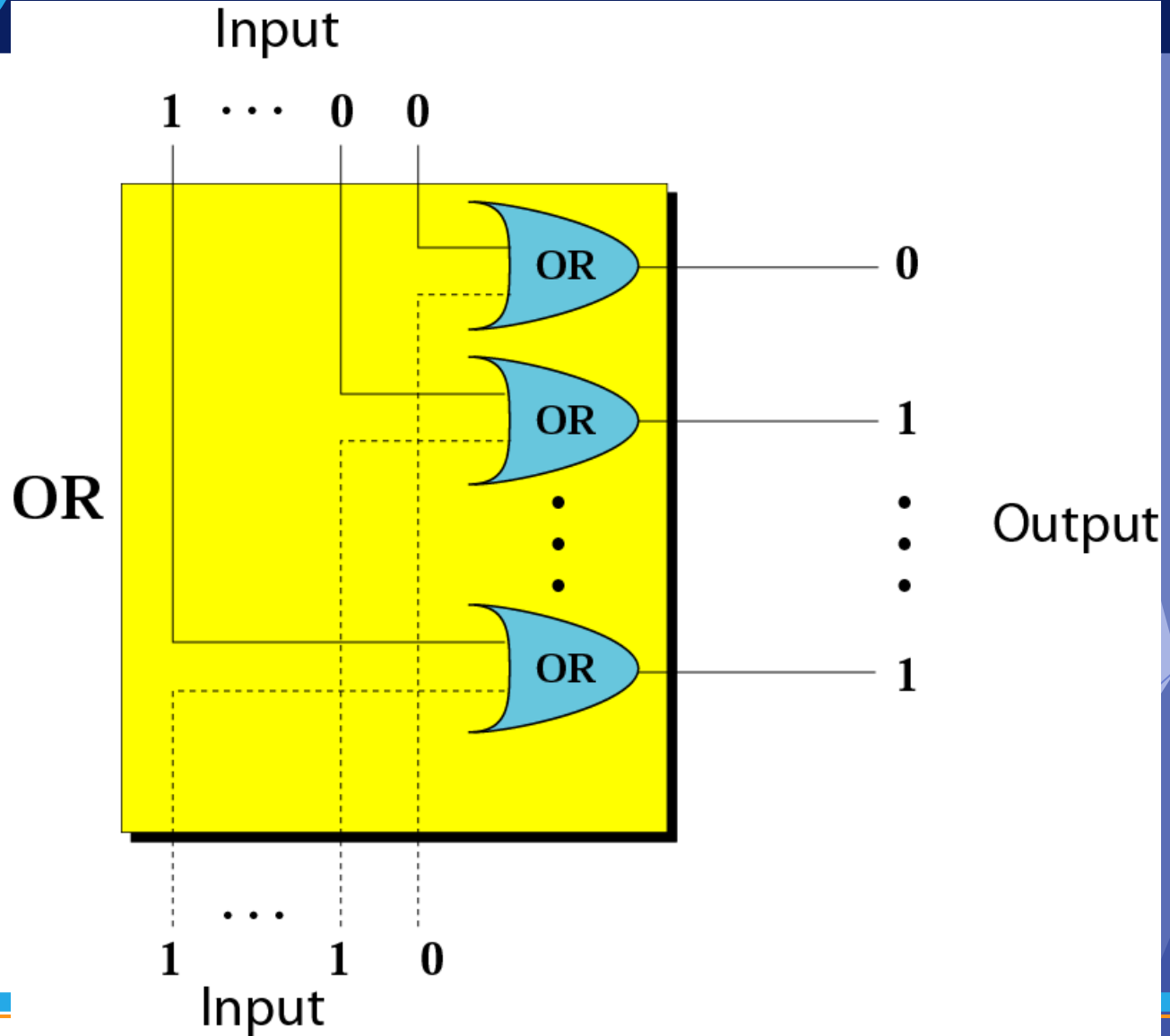
Example 4.4

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



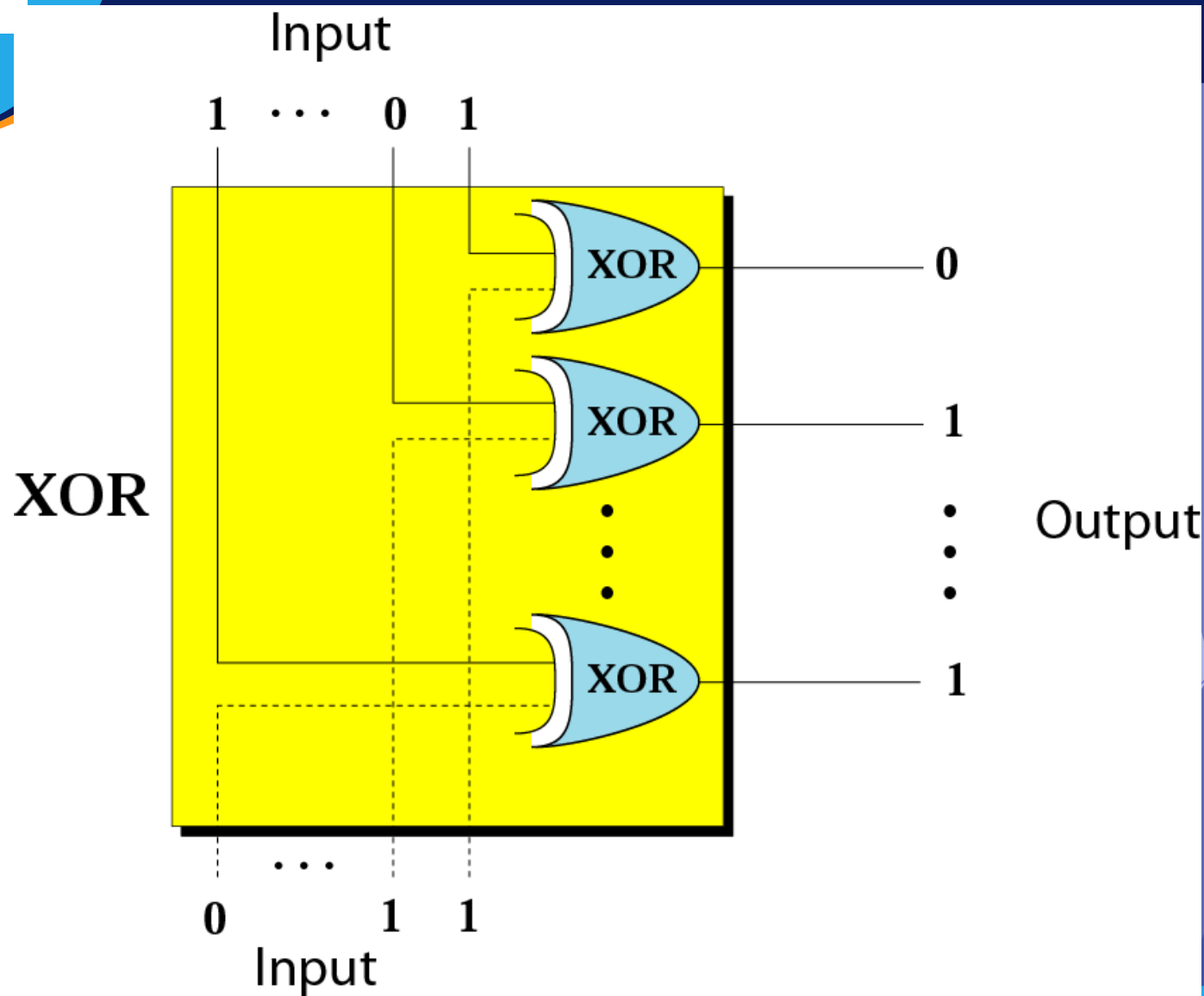
Example 4.5

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

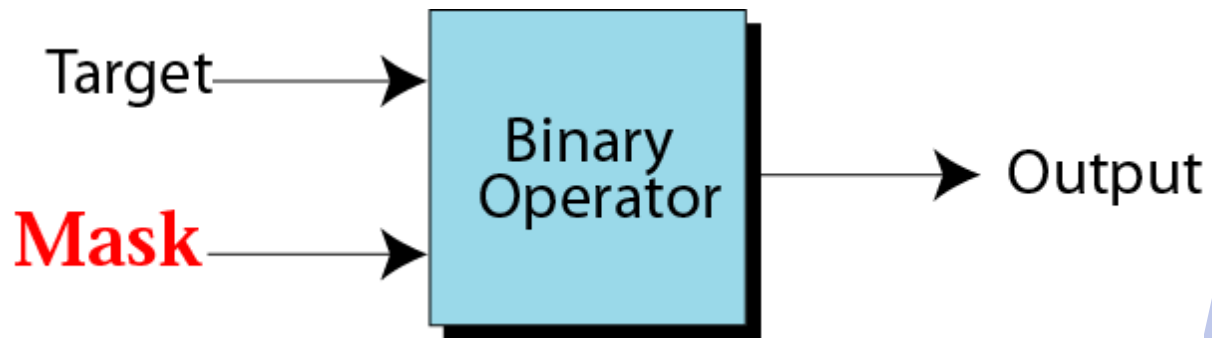
Applications

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

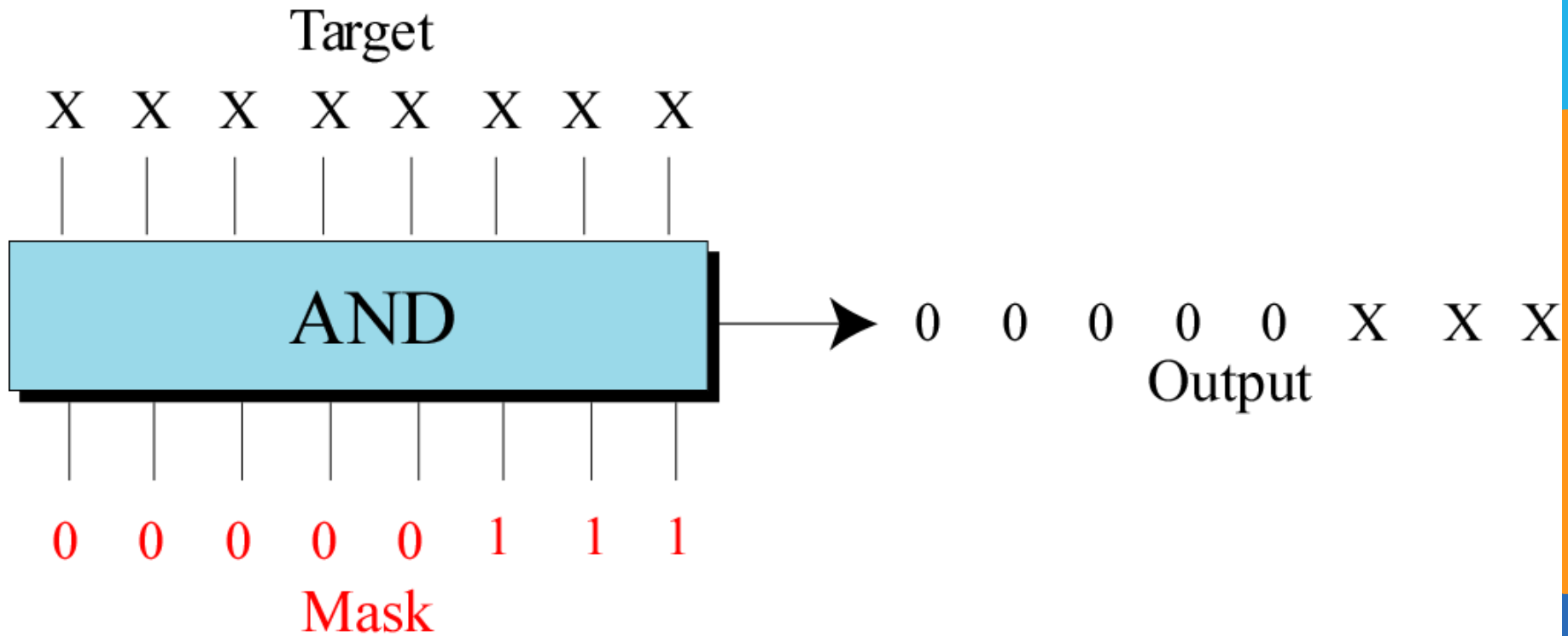
- ☐ **Complementing (NOT)**
- ☐ **Unsetting (AND)**
- ☐ **Setting (OR)**
- ☐ **Flipping (XOR)**



Mask



Example of unsetting specific bits



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

Application

Imagine a power plant that pumps water to a city using eight pumps. The state of the pumps (on or off) can be represented by an 8-bit pattern. For example, the pattern 11000111 shows that pumps 1 to 3 (from the right), 7 and 8 are on while pumps 4, 5, and 6 are off. Now assume pump 7 shuts down. How can a mask show this situation?

Solution on the next slide.



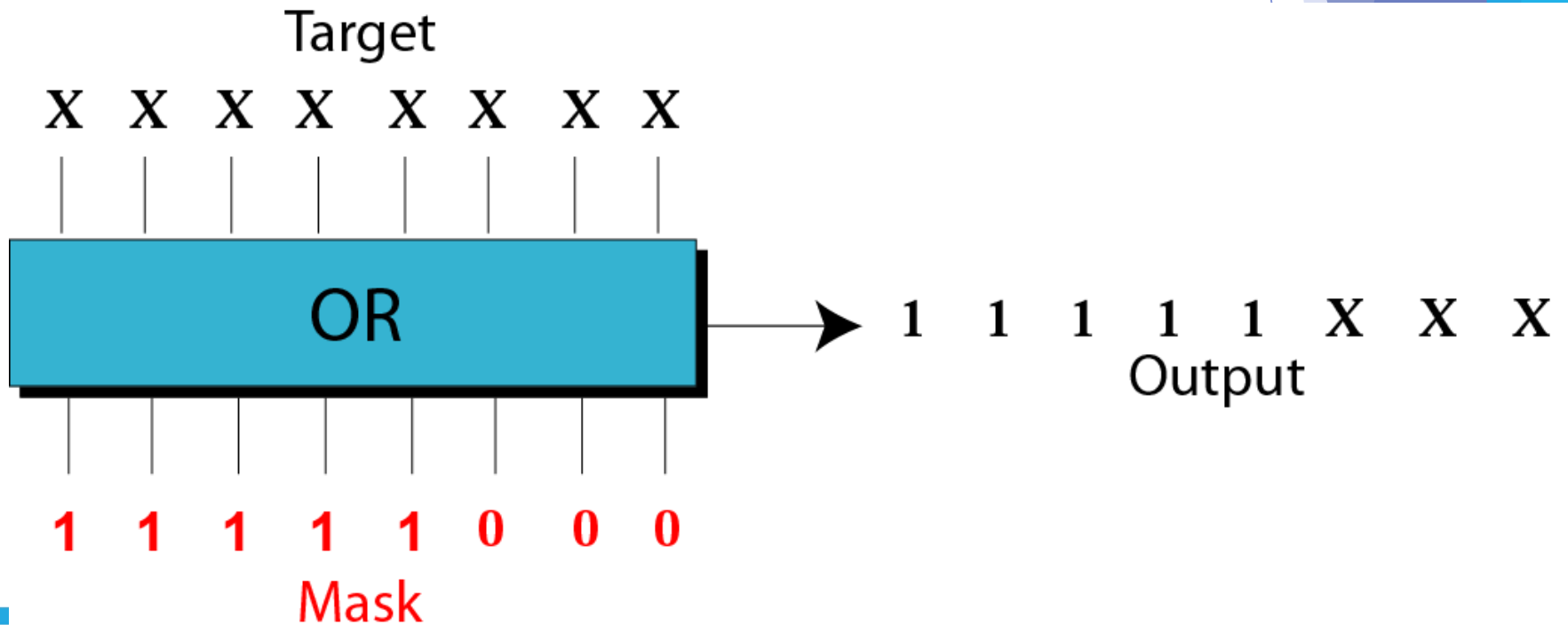
Solution

Use the mask 10111111 to AND with the target pattern. The only 0 bit (bit 7) in the mask turns off the seventh bit in the target.

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

Result	1 0 0 0 0 1 1 1	

Example of setting specific bits



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

Application

Using the power plant example, how can you use a mask to show that pump 6 is now turned on?

Solution

Use the mask **00100000**.

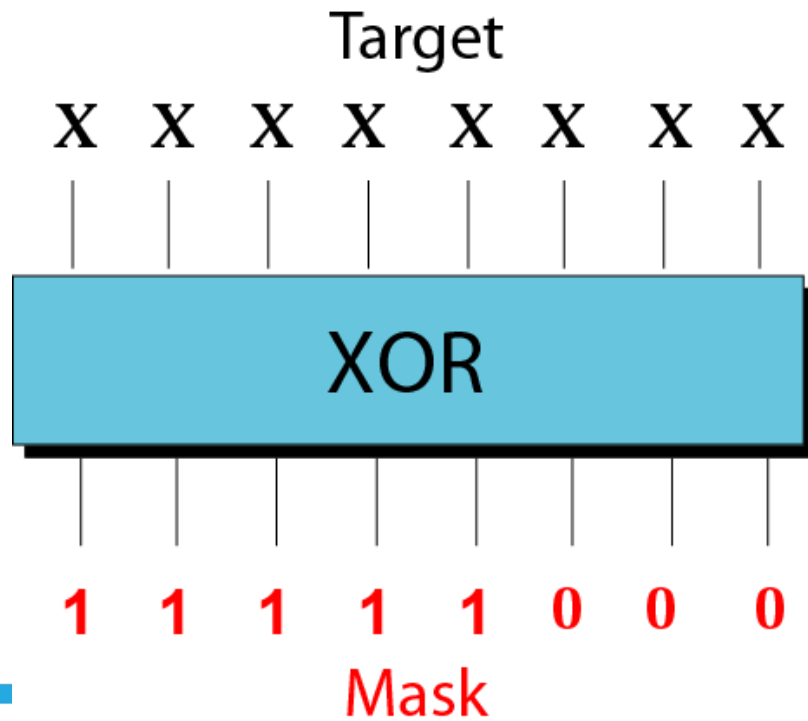
Target 1 0 0 0 0 1 1 1 *OR*

Mask 0 0 1 0 0 0 0 0

Result 1 0 1 0 0 1 1 1



Example of flipping specific bits



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

$\overline{X}$ $\overline{X}$ $\overline{X}$ $\overline{X}$ $\overline{X}$ X X X
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**.



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 distinguishes two types of logical shift operations, as described below:

- ❑ **Logical shift**
- ❑ **Logical circular shift (Rotate)**



Shift operations

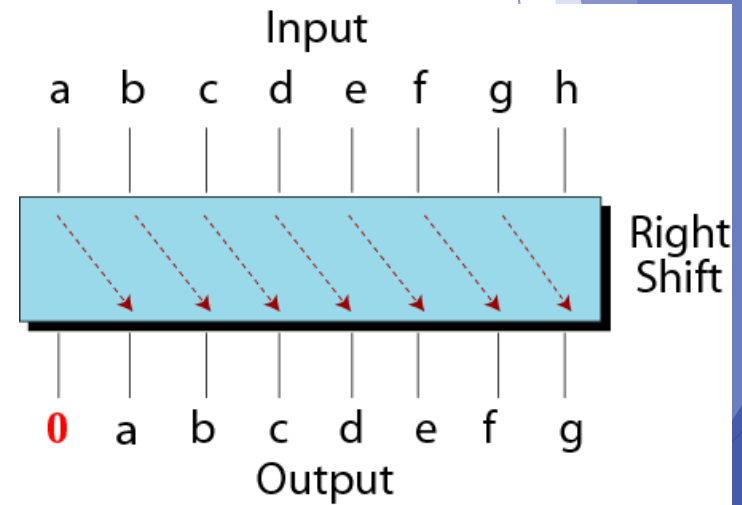
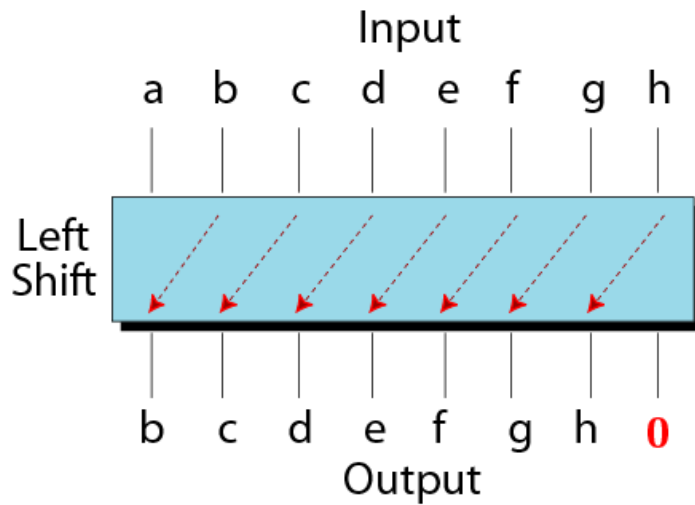
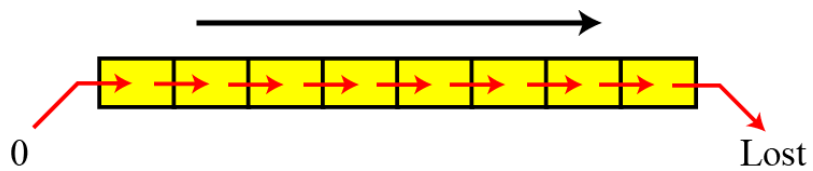
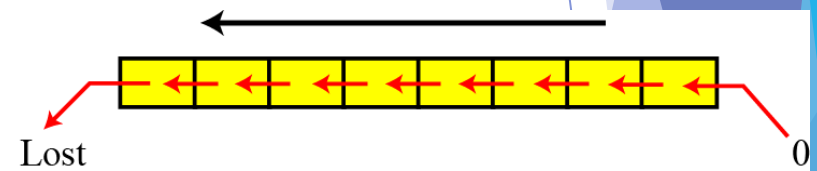


Figure 4.3 Logical shift operations



a. Logical right shift



b. Logical left shift

Example 4.10

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

Solution

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

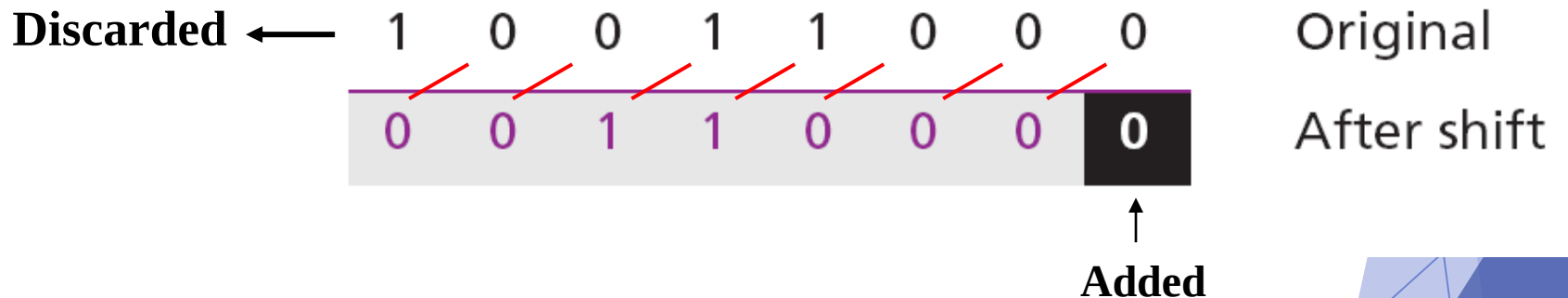
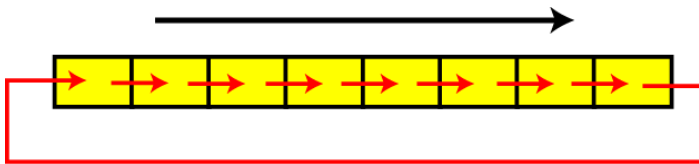
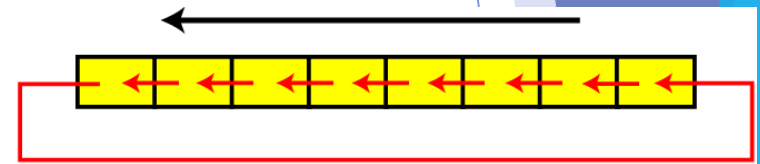


Figure 4.4 Circular shift operations



a. Circular right shift



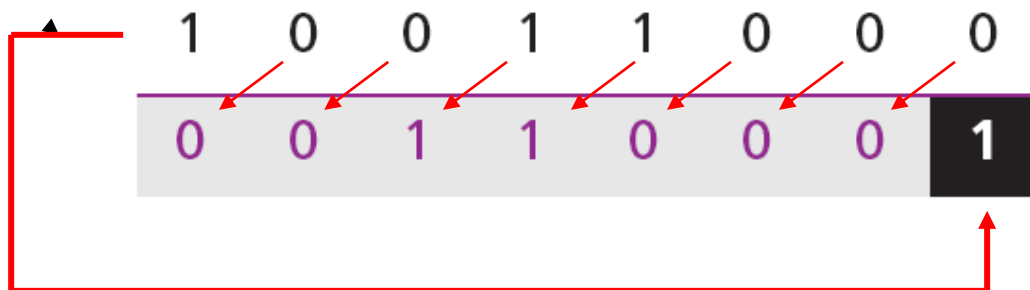
b. Circular left shift

Example 4.11

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

Solution

The solution is shown below. The leftmost bit is circulated and becomes the rightmost bit.



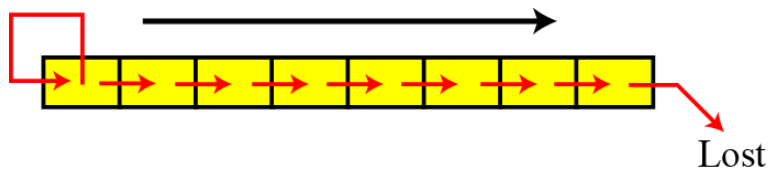
Original

After shift

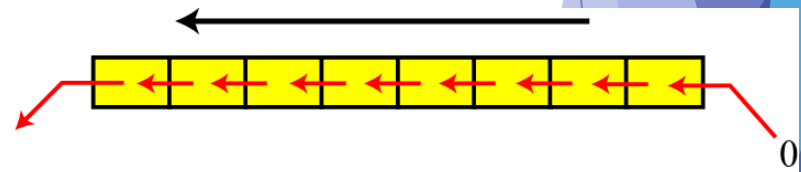
4.2.2 Arithmetic shift operations

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



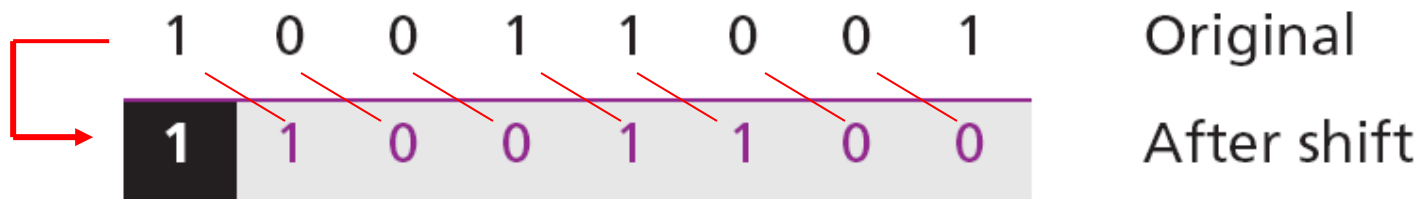
b. Arithmetic left shift

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

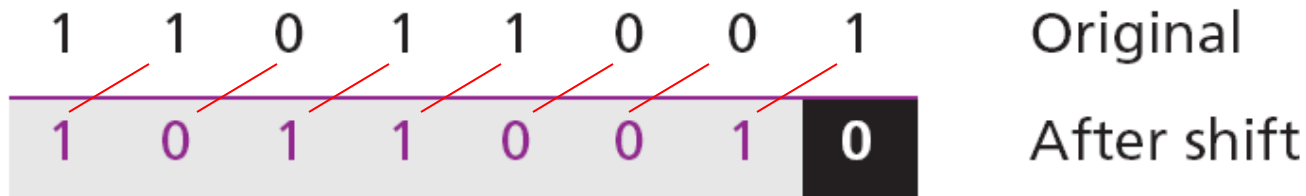


Example 4.13

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 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

Show how you can divide or multiply a number by 2 using shift operations.

Solution

If a bit pattern represents an unsigned number, a right-shift operation divides the number by two. The pattern 00111011 represents 59. When you shift the number to the right, you get 00011101, which is 29. If you shift the original number to the left, you get 01110110, which is 118.

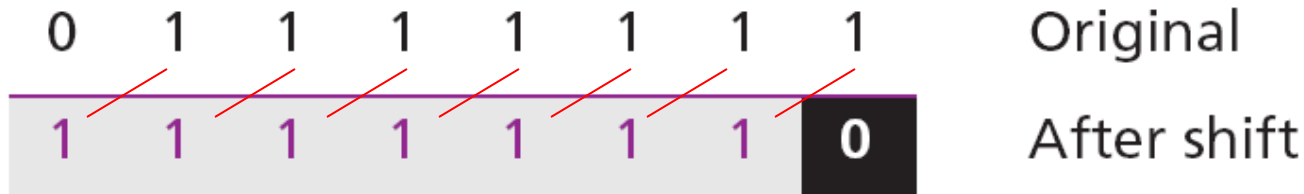


Example 4.14

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 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.



Adding bits

<i>Number of 1s</i> -----	<i>Result</i> -----	<i>Carry</i> -----
None	0	
One	1	
Two	0	1
Three	1	1



Note:

Rule of Adding Integers in Two's Complement

***Add 2 bits and propagate the carry to the next column.
If there is a final carry after the leftmost column
addition, discard it.***

Add two numbers in two's complement representation:

$$(+17) + (+22) \rightarrow (+39)$$

Solution

0 64 32 16 8 4 2 1

(+17)= 0 0 0 1 0 0 0 1

(+22)= 0 0 0 1 0 1 1 0

Add two numbers in two's complement representation:

$$(+17) + (+22) \rightarrow (+39)$$

Solution

Carry

1

$$\begin{array}{r} 00010001 + \\ 00010110 \\ \hline \end{array}$$

Result

$$00100111 \rightarrow 39$$



Example

Add two numbers in two's complement representation:

$$(+24) + (-17) \rightarrow (+7)$$

Solution

0 64 32 16 8 4 2 1

$$(+24) = 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0$$

$$(+17) = 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1$$

$$(-17) = 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1$$

Example

Add two numbers in two's complement representation: $(+24) + (-17)$
 $\rightarrow (+7)$

Solution

Carry

1 1 1 1 1

0 0 0 1 1 0 0 0 +

1 1 1 0 1 1 1 1

Result

0 0 0 0 0 1 1 1 $\rightarrow +7$

Example

Add two numbers in two's complement representation:

$$(-35) + (+20) \rightarrow (-15)$$

Solution

0 64 32 16 8 4 2 1

$$(+35) = 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1$$

$$(-35) = 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1$$

$$(+20) = 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0$$

Example

Add two numbers in two's complement representation: $(-35) + (+20)$
→ (-15)

Solution

Carry

$$\begin{array}{r}
 111 \\
 11011101+ \\
 00010100 \\
 \hline
 \end{array}$$

Result

$$\begin{array}{r}
 11110001 \rightarrow -15
 \end{array}$$

Example 4

Add two numbers in two's complement representation: $(+127) + (+3)$

→ $(+130)$

Solution

Carry	1	1	1	1	1	1	1	
	0	1	1	1	1	1	1	+
	0	0	0	0	0	0	1	1

Result	1	0	0	0	0	0	1	0

→ **-126 (Error)**

An overflow has occurred.





Note:

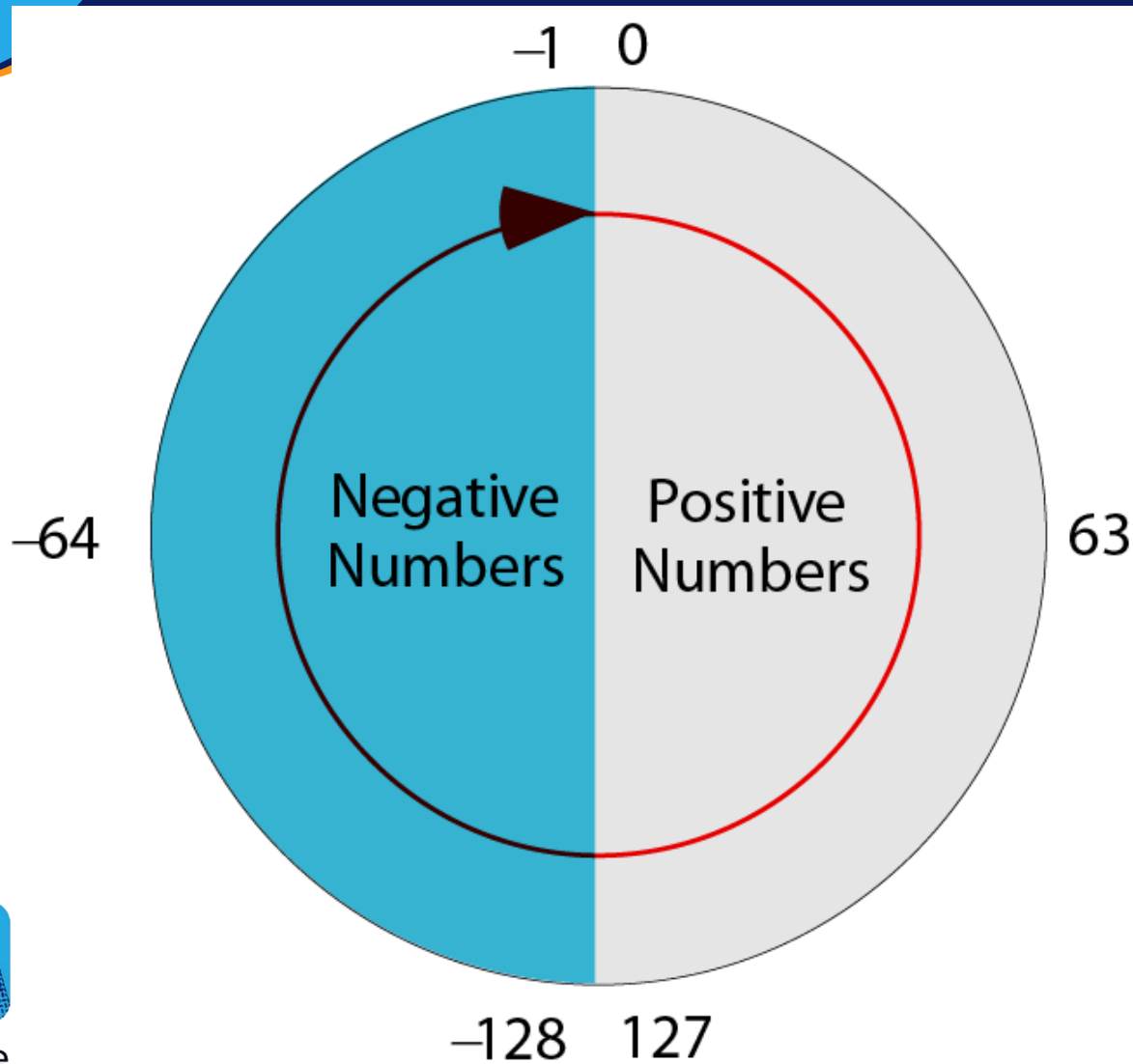
Range of numbers in two's complement representation

$$-(2^{N-1}) \text{ ----- } 0 \text{ ----- } +(2^{N-1} - 1)$$

Figure 4-2



Two's complement numbers visualization



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY



Note:

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



Example

Subtract 62 from 101 in two's complement:

$$(+101) - (+62) \leftrightarrow (+101) + (-62)$$

Solution

0 64 32 16 8 4 2 1

$$(+101) = \begin{matrix} 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{matrix}$$

$$(+62) = \begin{matrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{matrix}$$

$$(-62) = \begin{matrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{matrix}$$

Example

Subtract 62 from 101 in two's complement:

$$(+101) - (+62) \leftrightarrow (+101) + (-62)$$

Solution

Carry

1 1

0	1	1	0	0	1	0	1	+
1	1	0	0	0	0	1	0	

Result

0	0	1	0	0	1	1	1	→ 39
---	---	---	---	---	---	---	---	------

The leftmost carry is discarded.

Thank You

