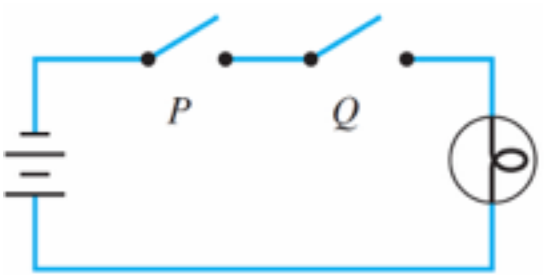
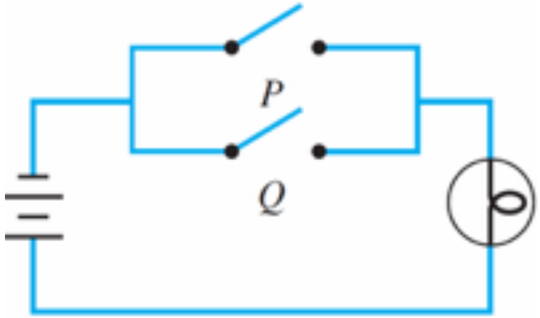
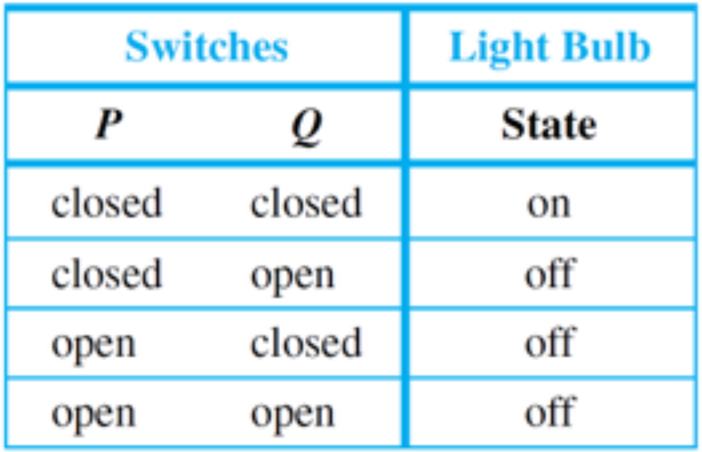
SECTION 2.4

# Application: Digital Logic Circuits

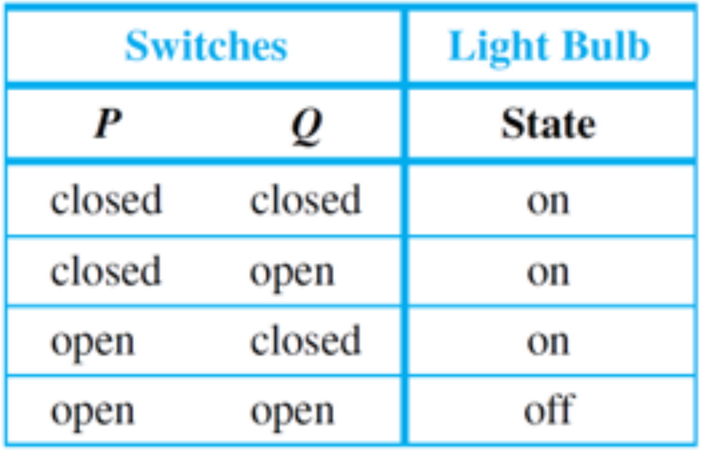
# Application: Digital Logic Circuits







| Switches | | Light Bulb |
|---|---|---|
| ***P*** | ***Q*** | **State** |
| closed | closed | on |
| closed | open | off |
| open | closed | off |
| open | open | off |



| Switches | | Light Bulb |
|---|---|---|
| ***P*** | ***Q*** | **State** |
| closed | closed | on |
| closed | open | on |
| open | closed | on |
| open | open | off |

Change *closed* and *on* are replaced by T, *open* and *off* are replac
by F?

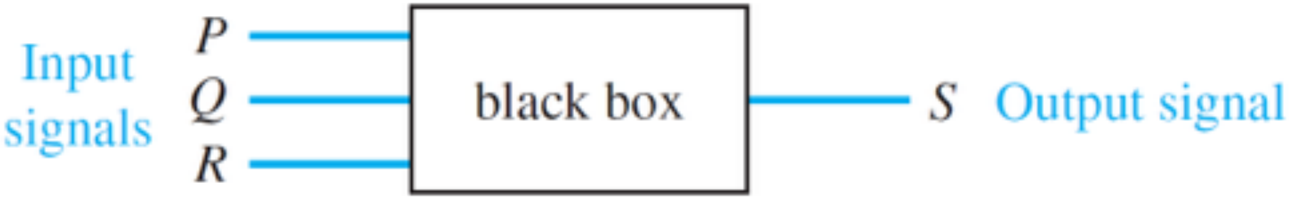
# Application: Digital Logic Circuits

- More complicated circuits correspond to more complicated logical expressions.
- This correspondence has been used extensively in design and study of circuits.
- Electrical engineers use language of logic when refer to values of signals produced by an electronic switch as being “true” or “false.”
  - Only that symbols 1 and 0 are used
  - symbols 0 and 1 are called **bits,** short for *b*inary dig*its*.
  - This terminology was introduced in 1946 by the statistician John Tukey.

# Black Boxes and Gates

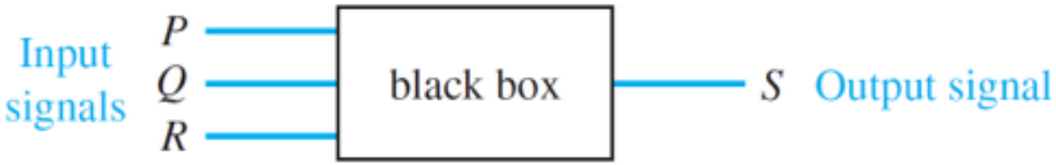
# Black Boxes and Gates

- **Circuits**: transform combinations of signal bits (1's and 0's) into other combinations of signal bits (1's and 0's).
- Computer engineers and digital system designers treat basic circuits as black boxes.
  - Ignore inside of a black box (detailed implementation of circuit)
  - focused on the relation between the **input** and the **output** signals.



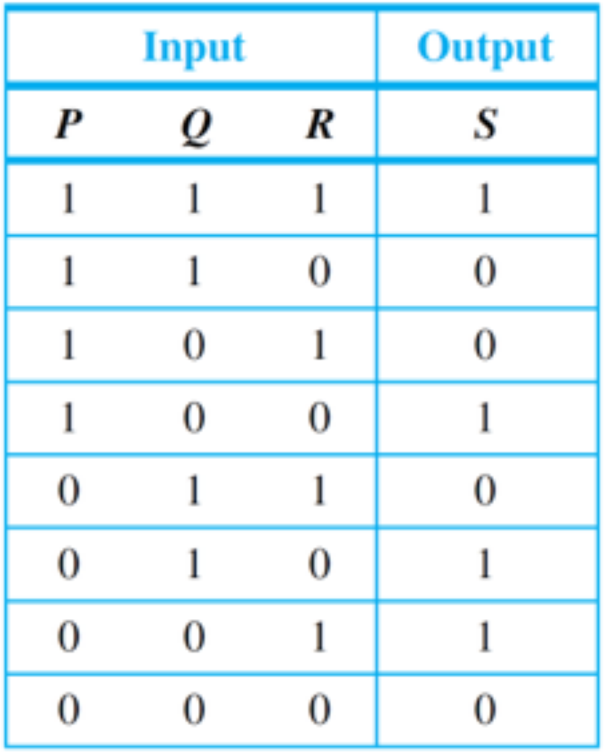
- Operation of a black box is completely specified by constructing an **input/output table** that lists all its possible input signals together with their corresponding output signals.

# Black Boxes and Gates



One possible correspondence of input to output signals is as follows:

An Input/Output Table



| Input | | | Output |
|---|---|---|---|
| ***P*** | ***Q*** | ***R*** | ***S*** |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 |

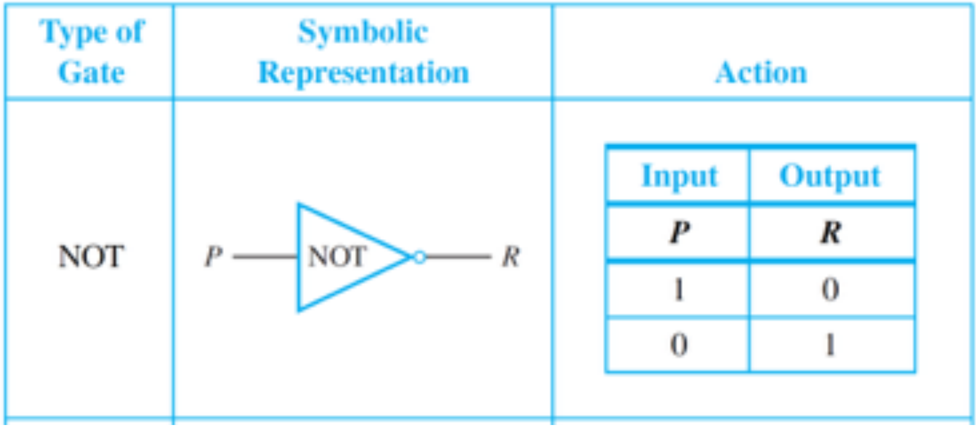
# Black Boxes and Gates

An efficient method for designing more complicated circuits is to build them by connecting less complicated black box circuits.

Gates can be combined into circuits in a variety of ways. If the rules shown on the next page are obeyed, the result is a **combinational circuit**, one whose output at any time is determined entirely by its input at that time without regard to previous inputs.

# Black Boxes and Gates

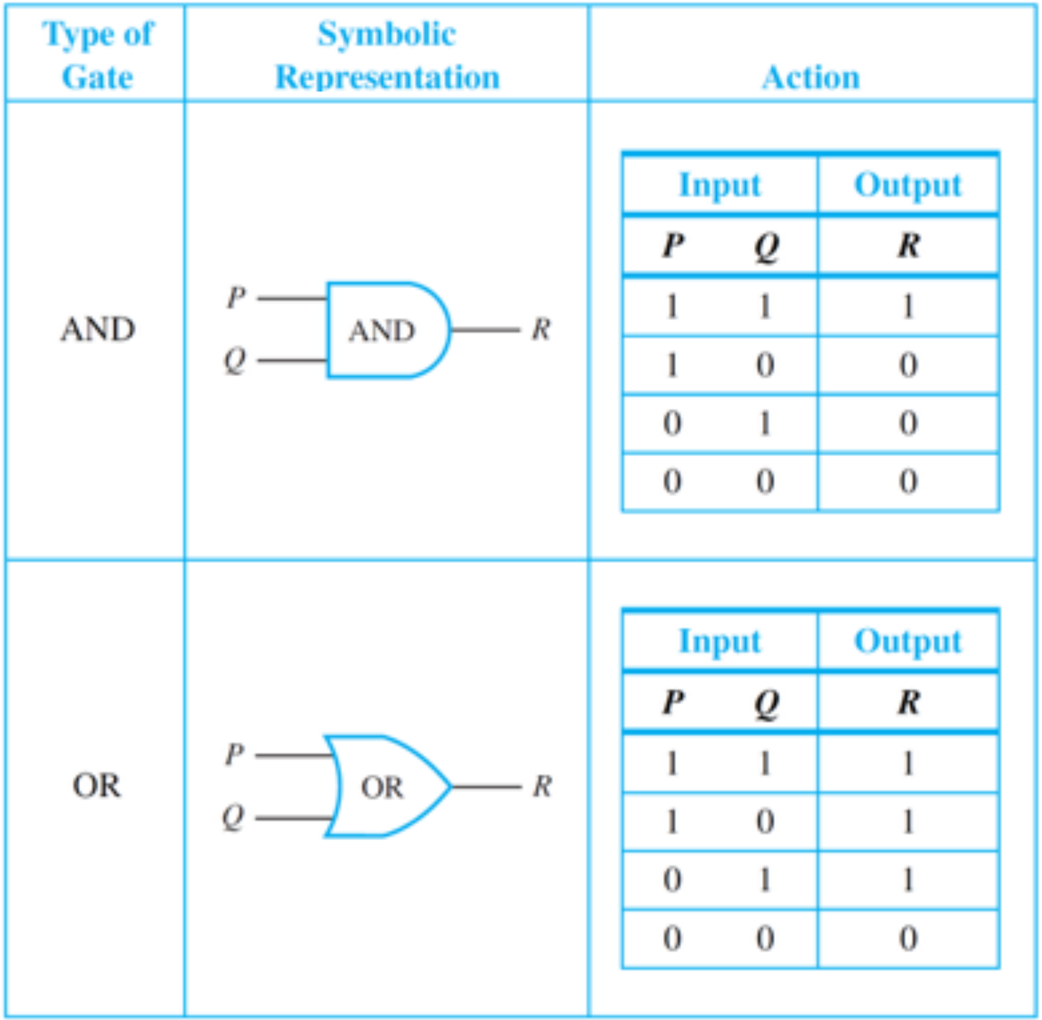
- A **NOT-gate** (or **inverter**) is a circuit with one input signal and one output signal. If the input signal is 1, the output signal is 0. Conversely, if the input signal is 0, then the output signal is 1.



<table>
<tr><th>Type of Gate</th><th>Symbolic Representation</th><th>Action</th></tr>
<tr><td>NOT</td><td>P NOT R</td><td><table><tr><th>Input</th><th>Output</th></tr><tr><th><i>P</i></th><th><i>R</i></th></tr><tr><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td></tr></table></td></tr>
</table>

# Black Boxes and Gates

- An **AND-gate** is a circuit with two input signals and one output signal. If both input signals are 1, then the output signal is 1.Otherwise, the output signal is 0.
- An **OR-gate** also has two input signals and one output signal. If both input signals are 0, then the output signal is 0. Otherwise, the output signal is 1.


P
Q
AND
R
P
Q
OR
R

| Type of Gate | Symbolic Representation | Action | | |
|---|---|---|---|---|
| | | Input | | Output |
| | | *P* | *Q* | *R* |
| AND | | 1 | 1 | 1 |
| | | 1 | 0 | 0 |
| | | 0 | 1 | 0 |
| | | 0 | 0 | 0 |
| OR | | 1 | 1 | 1 |
| | | 1 | 0 | 1 |
| | | 0 | 1 | 1 |
| | | 0 | 0 | 0 |

Figure 2.4.3 (continued)

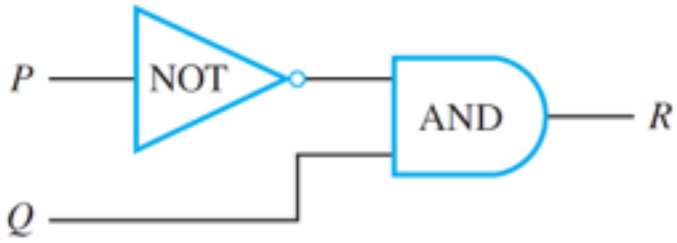
# The Input/Output Table for a Circuit

# *Determining Output for a Given Input*

Indicate the output of the circuits shown below for the given input signals.

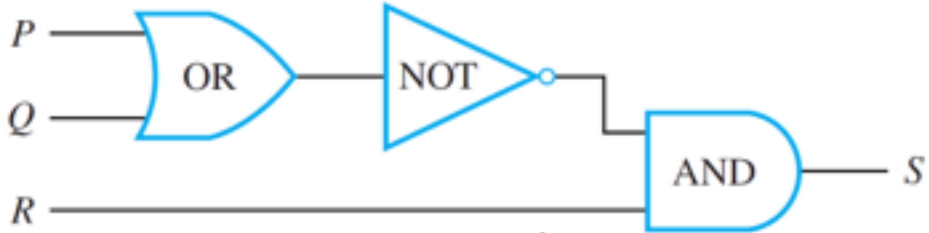
**a.**

Input signals: $P = 0$ and $Q = 1$



**b.**

Input signals: $P = 1,\quad Q = 0,\quad R = 1$



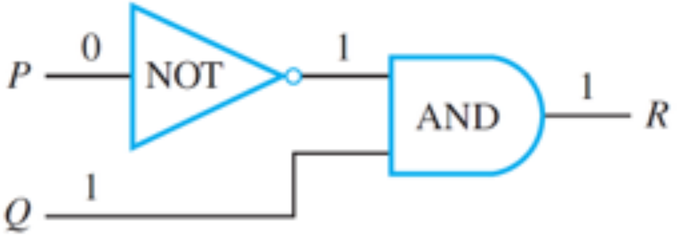
If you are given a set of input signals for a circuit, you can find its output by tracing through the circuit gate by gate.

# Example 1(a) – *Solution*

Move from left to right through the diagram, tracing the action of each gate on the input signals.

The NOT-gate changes *P* = *0* to a 1, so both inputs to the AND-gate are 1; hence the output *R* is 1.

This is illustrated by annotating the diagram as shown below.

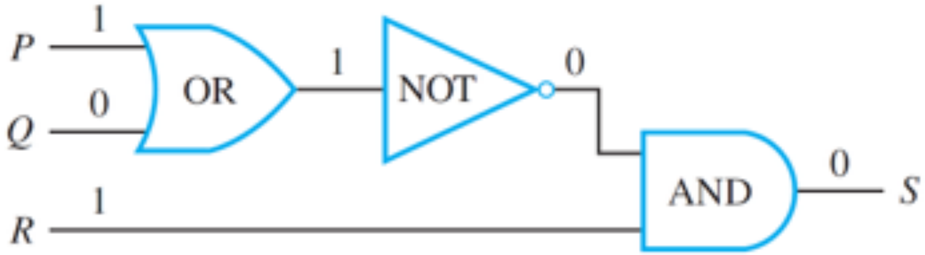


# Example 1(b) – *Solution*

cont’d

The output of the OR-gate is 1 since one of the input signals, *P*, is 1. The NOT-gate changes this 1 into a 0, so the two inputs to the AND-gate are 0 and *R* = 1.

Hence the output *S* is 0. The trace is shown below.



# Circuit => Boolean Expression

# Boolean Expression Corresponding to a Circuit

In logic, variables such as *p*, *q* and *r* represent statements, and a statement can have one of only two truth values: T(true) or F(false).

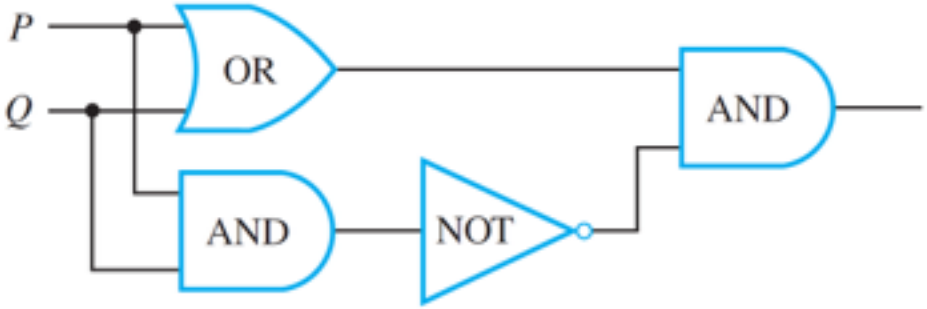
A **statement form** is an expression, such as $p \wedge (\sim q \vee r)$, composed of statement variables and logical connectives.

In honor of English mathematician George Boole:

- any variable, such as a statement variable or an input signal, that can take one of only two values is called a **Boolean variable.**
- An expression composed of Boolean variables and connectives $\sim$, $\wedge$, and $\vee$ is called a **Boolean expression.**

# *Finding a Boolean Expression for a Circuit*

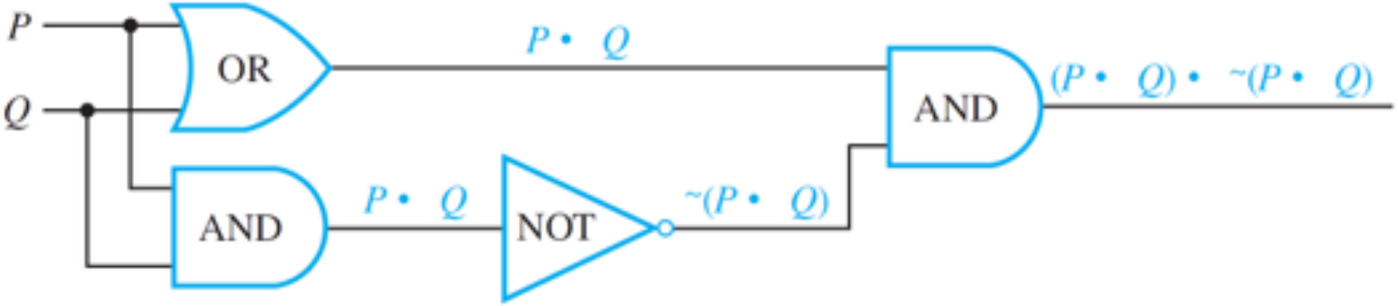
Find Boolean expressions that correspond to circuits shown below. A dot indicates a soldering of two wires; wires that cross without a dot are assumed not to touch.



Trace through the circuit from left (input) to right (output), indicating output of each gate symbolically…

# Solution

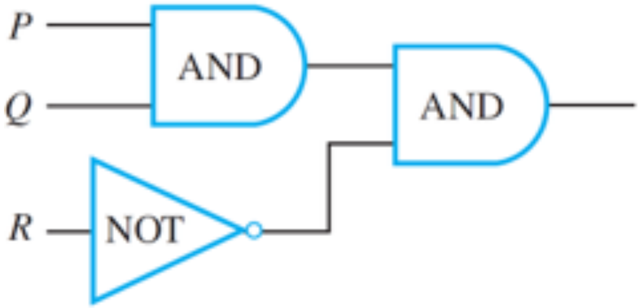
Trace through the circuit from left to right, indicating the output of each gate symbolically, as shown below.



The final expression obtained, $(P \vee Q) \wedge \sim(P \wedge Q)$, is the expression for exclusive or: *P* or *Q* but not both.

# *Finding a Boolean Expression for a Circuit (your turn)*

Find Boolean expressions that correspond to circuits shown below. A dot indicates a soldering of two wires; wires that cross without a dot are assumed not to touch.

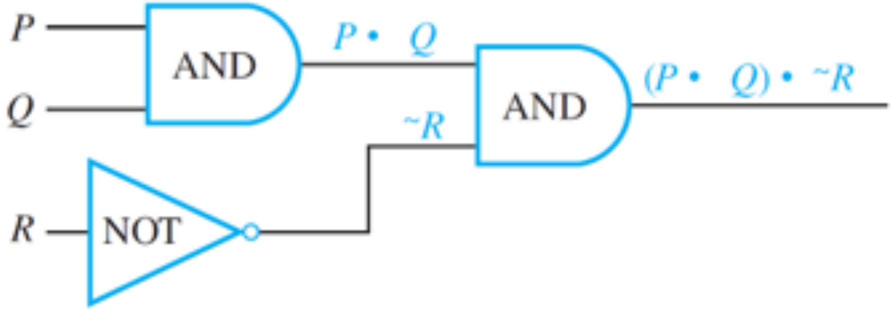


Trace through the circuit from left to right, indicating the output of each gate symbolically,

# Example 3(b) – *Solution*

cont'd

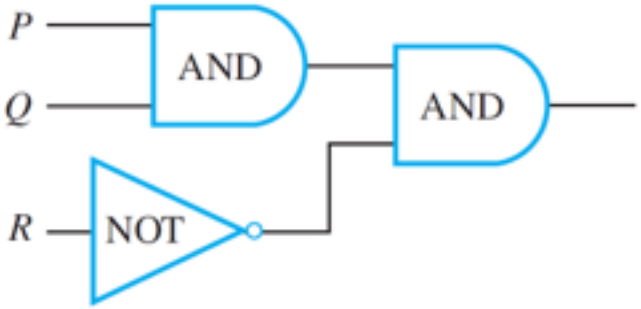
The Boolean expression corresponding to the circuit is $(P \wedge Q) \wedge {\sim}R$, as shown below.



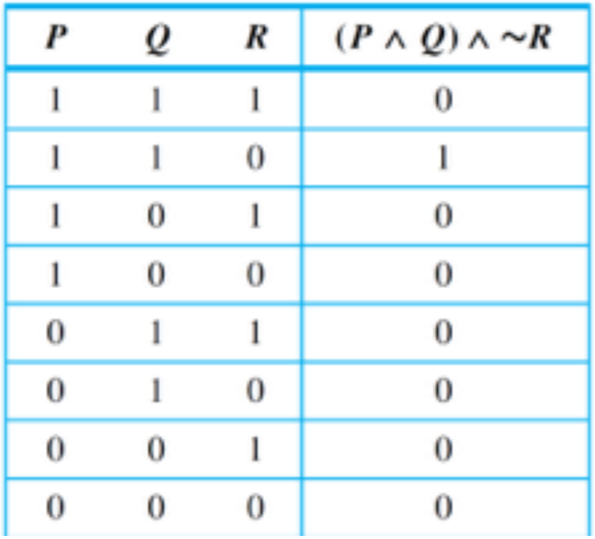
Observe the output is 1 for exactly one combination of inputs ($P$ = 1, $Q$ = 1, and $R$ = 0) and is 0 for all other combinations of inputs.

# The Boolean Expression Corresponding to a Circuit

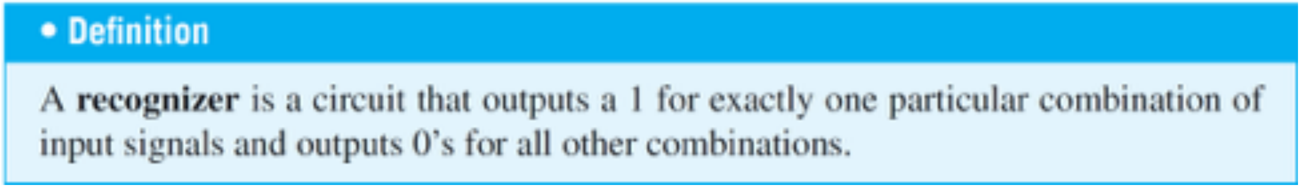
This circuit can be said to “recognize” one particular combination of inputs.



Input/Output Table for a Recognizer



| $P$ | $Q$ | $R$ | $(P \wedge Q) \wedge \sim R$ |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |



- **Definition**

A **recognizer** is a circuit that outputs a 1 for exactly one particular combination of input signals and outputs 0’s for all other combinations.

Boolean Expression ==> Circuit

# *Constructing Circuits for Boolean Expressions*

Construct circuits for following Boolean expressions.

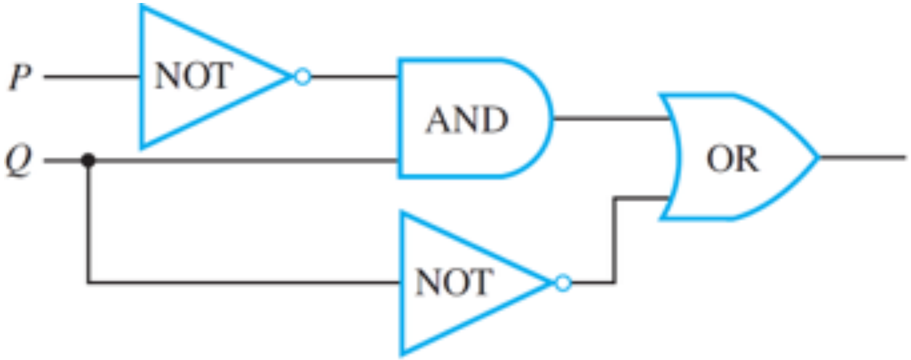
$(\sim P \wedge Q) \vee \sim Q$

Solution (from right to left)

Write input variables in a column on left side of diagram. Then go from right side of the diagram to the left, working from outermost part of the expression to the innermost part.

# Solution

cont’d



# *Constructing Circuits for Boolean Expressions*

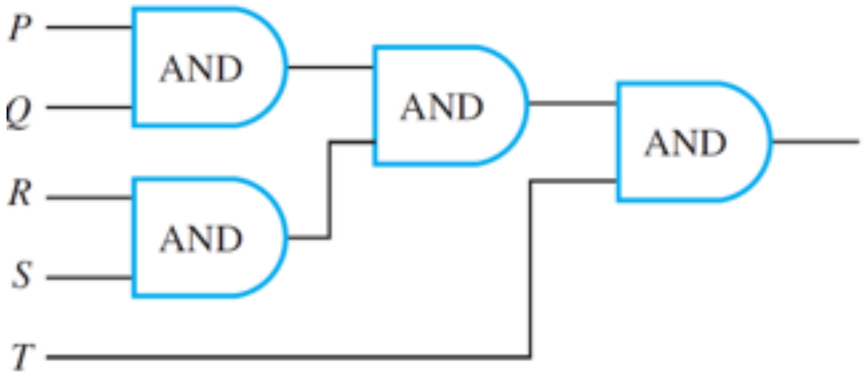
Construct circuits for following Boolean expressions.

$$((P \wedge Q) \wedge (R \wedge S)) \wedge T$$

# *Constructing Circuits for Boolean Expressions*

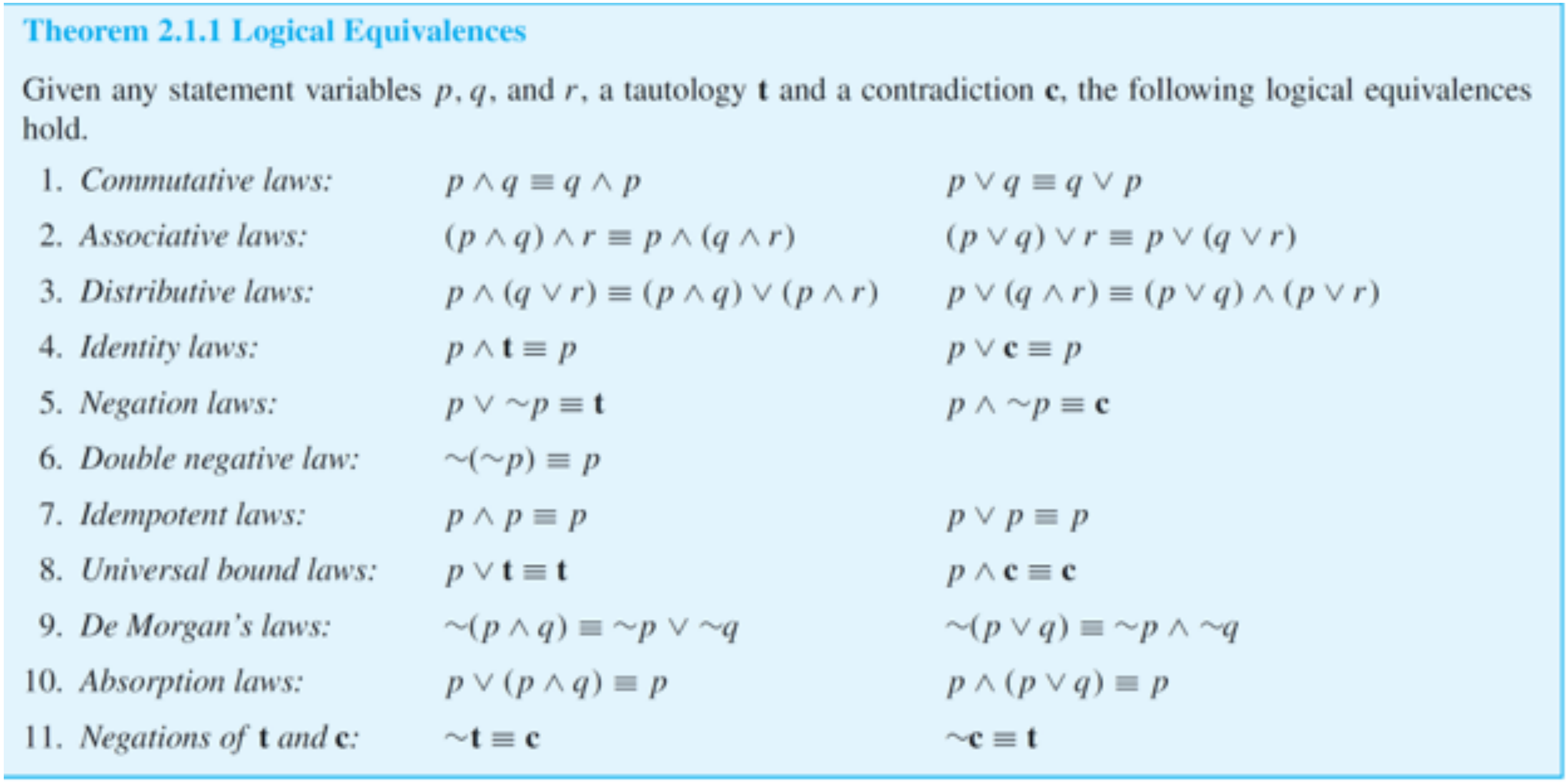
Construct circuits for following Boolean expressions.

$((P \wedge Q) \wedge (R \wedge S)) \wedge T$



# The Circuit Corresponding to a Boolean Expression

It follows from Theorem 2.1.1 that all the ways of adding parentheses to $P \wedge Q \wedge R \wedge S \wedge T$ are logically equivalent.



**Theorem 2.1.1 Logical Equivalences**

Given any statement variables $p$, $q$, and $r$, a tautology **t** and a contradiction **c**, the following logical equivalences hold.

| | | |
|---|---|---|
| 1. *Commutative laws:* | $p \wedge q \equiv q \wedge p$ | $p \vee q \equiv q \vee p$ |
| 2. *Associative laws:* | $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ | $(p \vee q) \vee r \equiv p \vee (q \vee r)$ |
| 3. *Distributive laws:* | $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ | $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ |
| 4. *Identity laws:* | $p \wedge \mathbf{t} \equiv p$ | $p \vee \mathbf{c} \equiv p$ |
| 5. *Negation laws:* | $p \vee {\sim}p \equiv \mathbf{t}$ | $p \wedge {\sim}p \equiv \mathbf{c}$ |
| 6. *Double negative law:* | ${\sim}({\sim}p) \equiv p$ | |
| 7. *Idempotent laws:* | $p \wedge p \equiv p$ | $p \vee p \equiv p$ |
| 8. *Universal bound laws:* | $p \vee \mathbf{t} \equiv \mathbf{t}$ | $p \wedge \mathbf{c} \equiv \mathbf{c}$ |
| 9. *De Morgan's laws:* | ${\sim}(p \wedge q) \equiv {\sim}p \vee {\sim}q$ | ${\sim}(p \vee q) \equiv {\sim}p \wedge {\sim}q$ |
| 10. *Absorption laws:* | $p \vee (p \wedge q) \equiv p$ | $p \wedge (p \vee q) \equiv p$ |
| 11. *Negations of* **t** *and* **c**: | ${\sim}\mathbf{t} \equiv \mathbf{c}$ | ${\sim}\mathbf{c} \equiv \mathbf{t}$ |

# The Circuit Corresponding to a Boolean Expression

Thus, for example,

$$((P \wedge Q) \wedge (R \wedge S)) \wedge T \equiv (P \wedge (Q \wedge R)) \wedge (S \wedge T).$$

It also follows that the circuit in Figure 2.4.5, which corresponds to $(P \wedge (Q \wedge R)) \wedge (S \wedge T)$, has the same input/output table as the circuit in Figure 2.4.4, which corresponds to $((P \wedge Q) \wedge (R \wedge S)) \wedge T$.

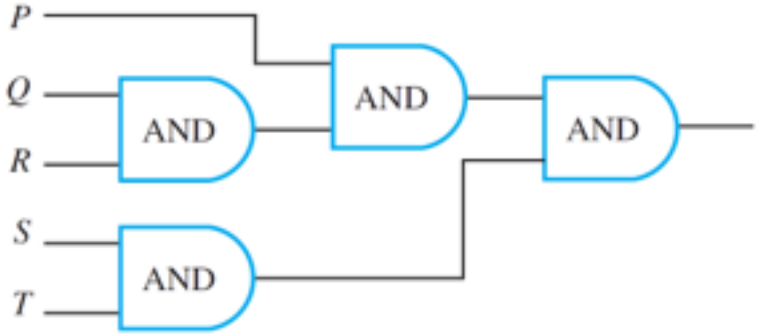


Figure 2.4.5

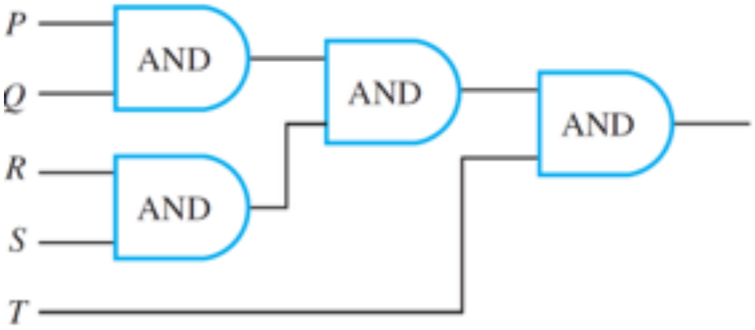


Figure 2.4.4

# The Circuit Corresponding to a Boolean Expression

Each of the circuits in Figures 2.4.4 and 2.4.5 is, therefore, an implementation of the expression $P \wedge Q \wedge R \wedge S \wedge T$. Such a circuit is called a **multiple-input AND-gate** and is represented by the diagram shown in Figure 2.4.6.

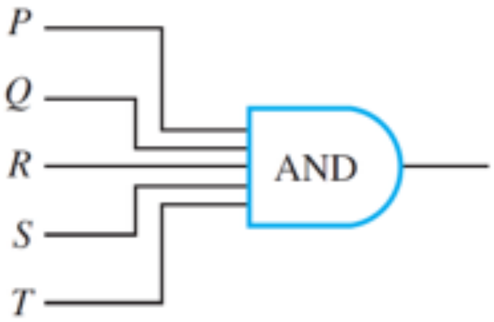


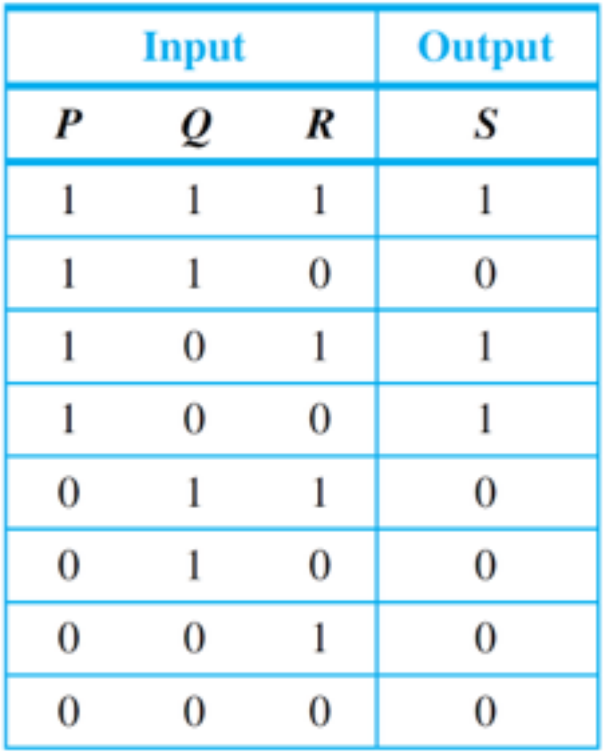
**Figure 2.4.6**

**Multiple-input OR-gates** are constructed similarly.

a Given Input/Output Table ==> Circuit

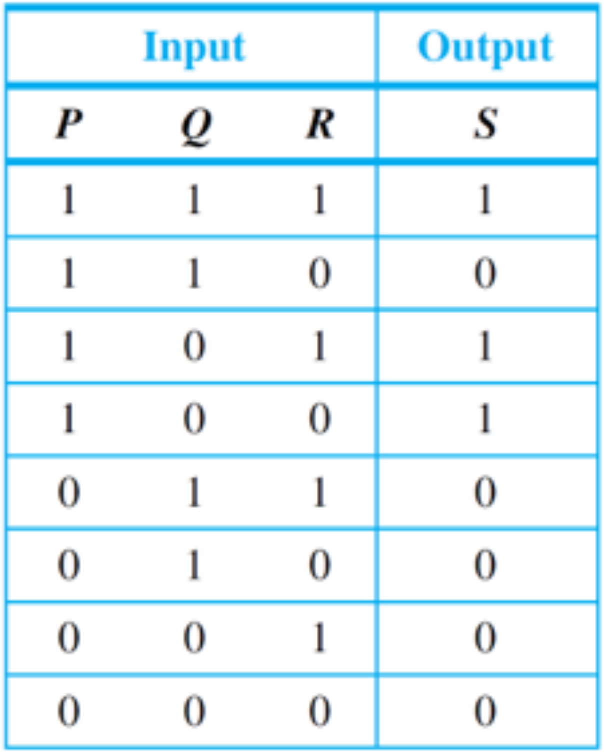
# *Designing a Circuit for a Given Input/Output Table*

Design a circuit for the following input/output table:



| Input | | | Output |
|---|---|---|---|
| *P* | *Q* | *R* | *S* |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |

# Example 5 – *Solution*



| Input | | | Output |
|---|---|---|---|
| *P* | *Q* | *R* | *S* |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |

First construct a Boolean expression with this table as its truth table.

- identify each row for which the output is 1—in this case, the first, third, and fourth rows.
- For each such row, construct an *and* expression that recognize that combination.
- It follows that one Boolean expression with the given truth table is
  $(P \wedge Q \wedge R) \vee (P \wedge {\sim}Q \wedge R) \vee (P \wedge {\sim}Q \wedge {\sim}R)$.

# Example 5 – *Solution* cont'd

The circuit corresponding to this expression has the diagram shown in Figure 2.4.7.

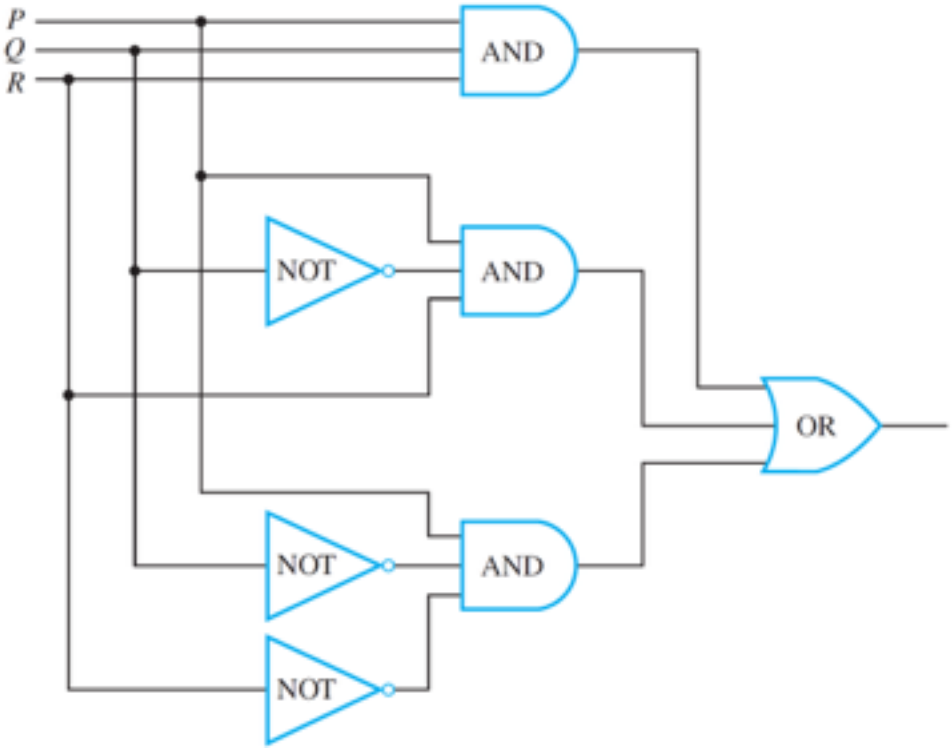


**Figure 2.4.7**

# Example 5 – *Solution*

cont'd

Observe that expression

$$(P \wedge Q \wedge R) \vee (P \wedge {\sim}Q \wedge R) \vee (P \wedge {\sim}Q \wedge {\sim}R). \qquad 2.4.5$$

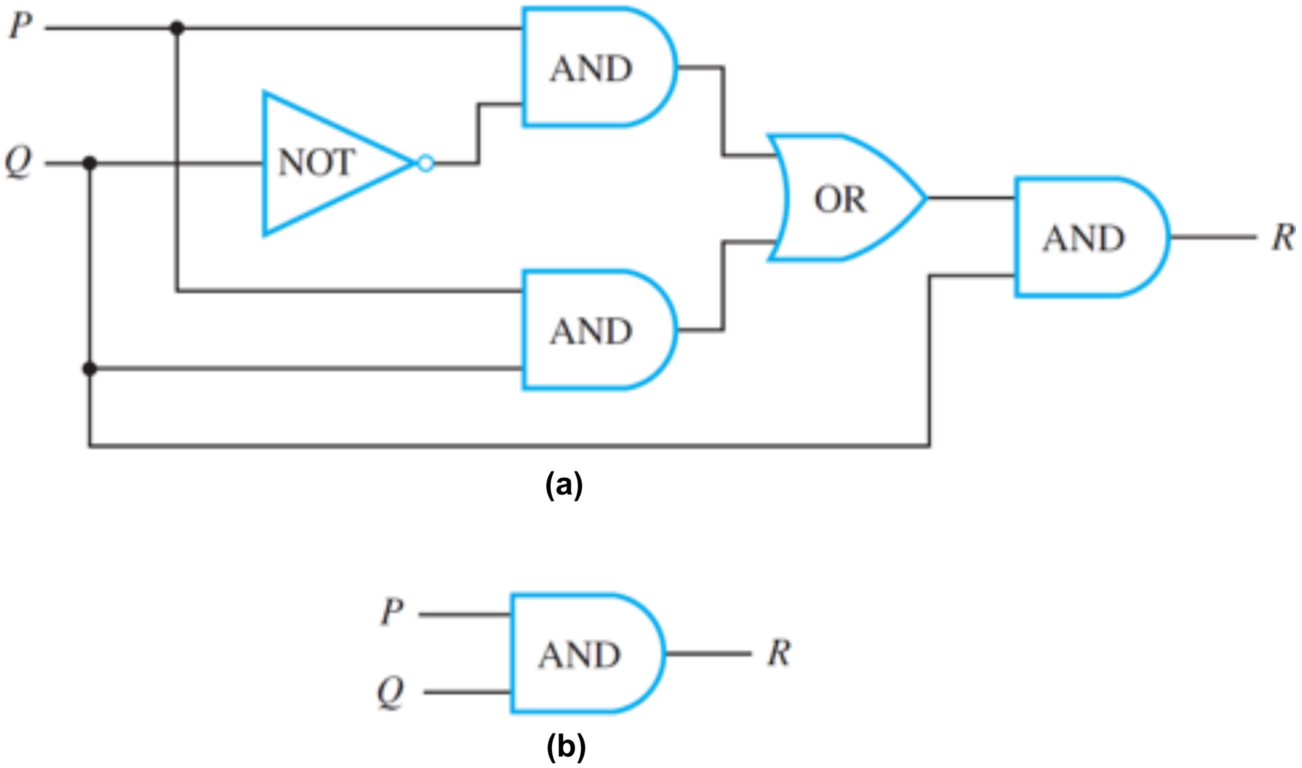
is a disjunction of terms that are themselves conjunctions in which one of $P$ or $\sim P$, one of $Q$ or $\sim Q$, and one of $R$ or $\sim R$ all appear.

Such expressions are said to be in **disjunctive normal form** or **sum-of-products form**.

# Simplifying Combinational Circuits

# Simplifying Combinational Circuits

Consider the two combinational circuits shown in Figure 2.4.8.



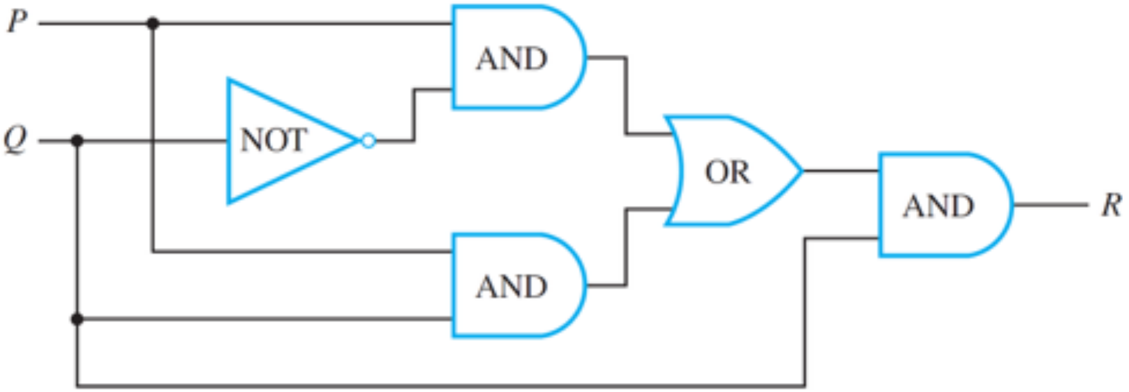
(a)

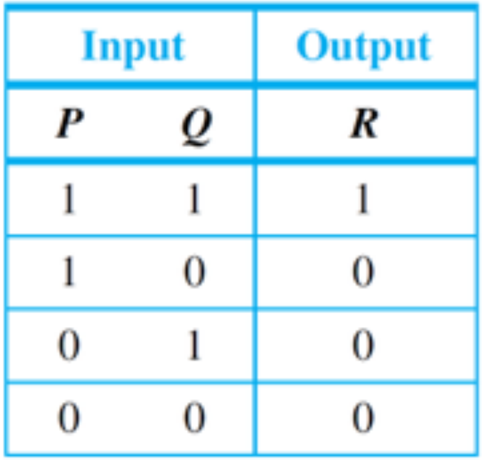
(b)

**Figure 2.4.8**

# Simplifying Combinational Circuits

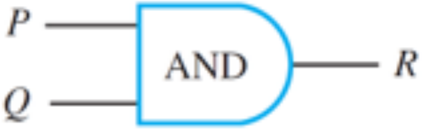
Trace through following circuit, you will find that its input/ output table is





| Input | | Output |
|---|---|---|
| *P* | *Q* | *R* |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

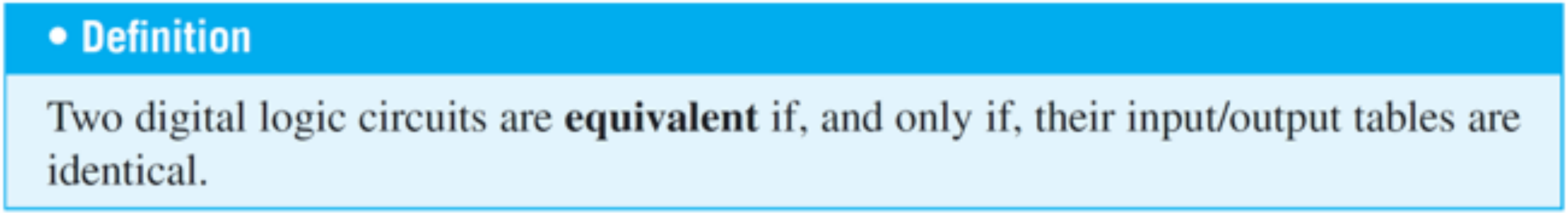
which is the same as the input/output table for following circuit



the two circuits do same job in the sense that they transform combinations of input signals into same output signals.

# Simplifying Combinational Circuits

Yet circuit (b) is simpler than circuit (a) in that it contains many fewer logic gates. Thus, as part of an integrated circuit, it would take less space and require less power.

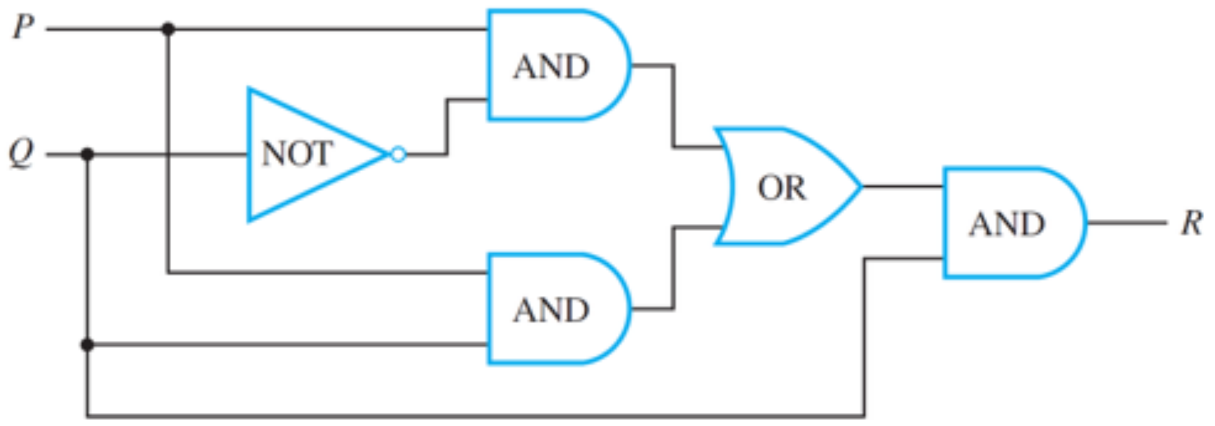


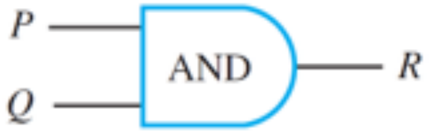
- **Definition**

Two digital logic circuits are **equivalent** if, and only if, their input/output tables are identical.

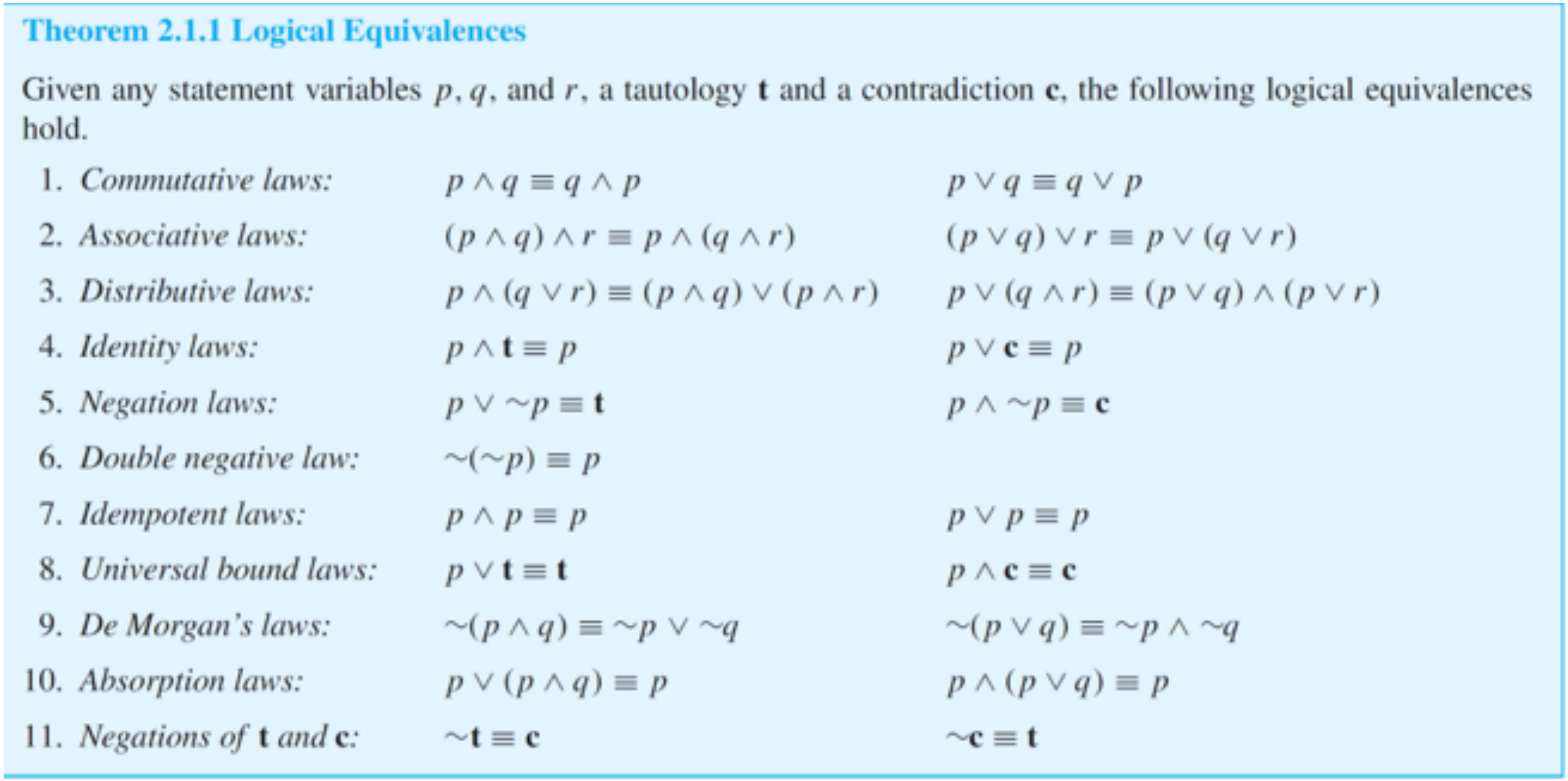
# Showing Two Circuits Are Equivalent

Find the Boolean expressions for each circuit below. Use Theorem 2.1.1 (logical equivalence) to show that these expressions are logically equivalent.





# Example 6 – *Showing That Two Circuits Are Equivalent* cont’d



**Theorem 2.1.1 Logical Equivalences**

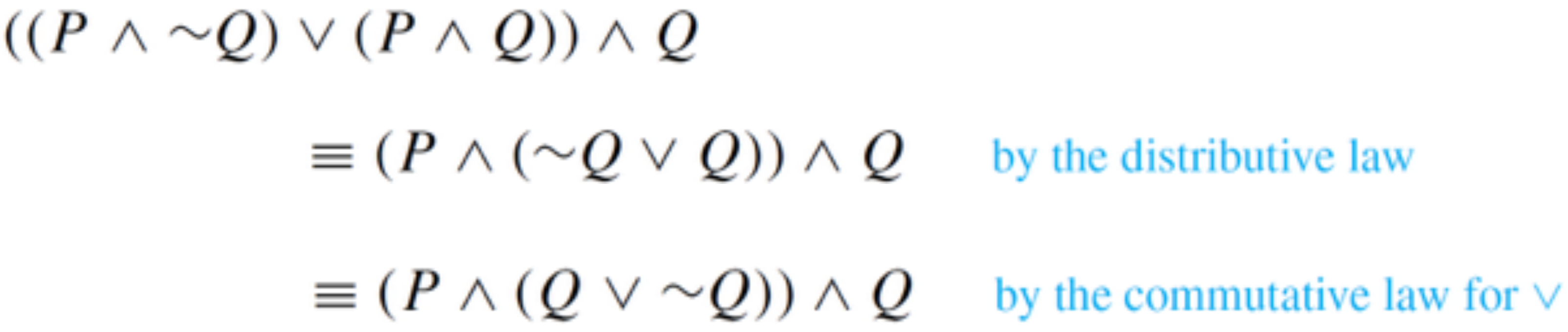
Given any statement variables $p$, $q$, and $r$, a tautology $\mathbf{t}$ and a contradiction $\mathbf{c}$, the following logical equivalences hold.

| | | |
|---|---|---|
| 1. *Commutative laws:* | $p \wedge q \equiv q \wedge p$ | $p \vee q \equiv q \vee p$ |
| 2. *Associative laws:* | $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ | $(p \vee q) \vee r \equiv p \vee (q \vee r)$ |
| 3. *Distributive laws:* | $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ | $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ |
| 4. *Identity laws:* | $p \wedge \mathbf{t} \equiv p$ | $p \vee \mathbf{c} \equiv p$ |
| 5. *Negation laws:* | $p \vee {\sim}p \equiv \mathbf{t}$ | $p \wedge {\sim}p \equiv \mathbf{c}$ |
| 6. *Double negative law:* | ${\sim}({\sim}p) \equiv p$ | |
| 7. *Idempotent laws:* | $p \wedge p \equiv p$ | $p \vee p \equiv p$ |
| 8. *Universal bound laws:* | $p \vee \mathbf{t} \equiv \mathbf{t}$ | $p \wedge \mathbf{c} \equiv \mathbf{c}$ |
| 9. *De Morgan’s laws:* | ${\sim}(p \wedge q) \equiv {\sim}p \vee {\sim}q$ | ${\sim}(p \vee q) \equiv {\sim}p \wedge {\sim}q$ |
| 10. *Absorption laws:* | $p \vee (p \wedge q) \equiv p$ | $p \wedge (p \vee q) \equiv p$ |
| 11. *Negations of* **t** *and* **c**: | ${\sim}\mathbf{t} \equiv \mathbf{c}$ | ${\sim}\mathbf{c} \equiv \mathbf{t}$ |

# Example 6 – *Solution*

The Boolean expressions that correspond to circuits (a) and (b) are $((P \wedge {\sim}Q) \vee (P \wedge Q)) \wedge Q$ and $P \wedge Q$, respectively.

By Theorem 2.1.1,



$$((P \wedge {\sim}Q) \vee (P \wedge Q)) \wedge Q$$

$$\equiv (P \wedge ({\sim}Q \vee Q)) \wedge Q \qquad \text{by the distributive law}$$

$$\equiv (P \wedge (Q \vee {\sim}Q)) \wedge Q \qquad \text{by the commutative law for } \vee$$

# Example 6 – *Solution*

cont’d

$$\equiv (P \wedge \mathbf{t}) \wedge Q \qquad \text{by the negation law}$$

$$\equiv P \wedge Q \qquad \text{by the identity law.}$$

It follows that the truth tables for $((P \wedge {\sim}Q) \vee (P \wedge Q)) \wedge Q$ and $P \wedge Q$ are the same.

Hence the input/output tables for the circuits corresponding to these expressions are also the same, and so the circuits are equivalent.

# NAND and NOR Gates

# NAND and NOR Gates

Another way to simplify a circuit is to use different gates:

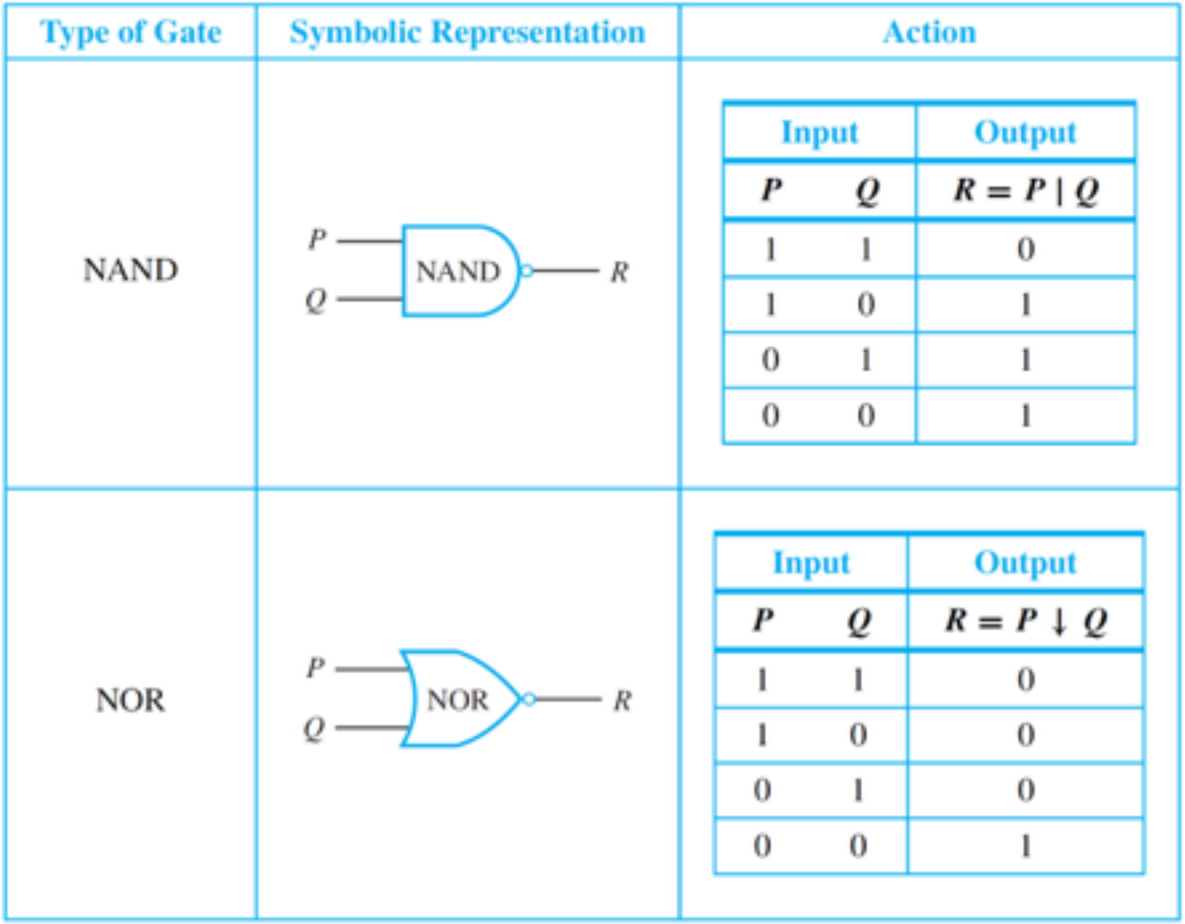
- A NAND-gate is a single gate that acts like an AND-gate followed by a NOT-gate. The logical symbol is |, called a **Sheffer stroke** (after H. M. Sheffer, 1882–1964)

- A NOR-gate acts like an OR-gate followed by a NOT-gate. The logical symbol is ↓ (for NOR), which is called a **Peirce arrow** (after C. S. Peirce, 1839–1914)

$$P \mid Q \equiv {\sim}(P \wedge Q) \quad \text{and} \quad P \downarrow Q \equiv {\sim}(P \vee Q).$$

# NAND and NOR Gates

The table below summarizes the actions of NAND and NOR gates.



<table>
<tr><th>Type of Gate</th><th>Symbolic Representation</th><th>Action</th></tr>
<tr><td>NAND</td><td>P<br>Q<br>NAND<br>R</td><td>
<table>
<tr><th colspan="2">Input</th><th>Output</th></tr>
<tr><th>$P$</th><th>$Q$</th><th>$R = P \mid Q$</th></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
</table>
</td></tr>
<tr><td>NOR</td><td>P<br>Q<br>NOR<br>R</td><td>
<table>
<tr><th colspan="2">Input</th><th>Output</th></tr>
<tr><th>$P$</th><th>$Q$</th><th>$R = P \downarrow Q$</th></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
</table>
</td></tr>
</table>

# NAND and NOR Gates

It can be shown that any Boolean expression is equivalent to one written entirely with Sheffer strokes or entirely with Peirce arrows.

Thus any digital logic circuit is equivalent to one that uses only NAND-gates or only NOR-gates.

## Example 7 – *Rewriting Expressions Using the Sheffer Stroke*

Use Theorem 2.1.1 and the definition of Sheffer stroke to show that

**a.** $\sim P \equiv P \mid P$ and **b.** $P \vee Q \equiv (P \mid P) \mid (Q \mid Q)$.

Solution:

**a.**

$$
\begin{aligned}
\sim P &\equiv \sim(P \wedge P) && \text{by the idempotent law for } \wedge \\
&\equiv P \mid P && \text{by definition of } \mid .
\end{aligned}
$$

**b.**

$$
\begin{aligned}
P \vee Q &\equiv \sim(\sim(P \vee Q)) && \text{by the double negative law} \\
&\equiv \sim(\sim P \wedge \sim Q) && \text{by De Morgan's laws}
\end{aligned}
$$

# Example 7 – *Solution*

cont’d

$$\equiv \sim((P \mid P) \wedge (Q \mid Q)) \qquad \text{by part (a)}$$

$$\equiv (P \mid P) \mid (Q \mid Q) \qquad \text{by definition of } \mid.$$

## SECTION 2.5

# Application: Number Systems and Circuits for Addition

# Application: Number Systems and Circuits for Addition

Meaning of decimal notation:

to interpret a string of decimal digits as a number, you multiply each digit by its place value.

e.g., 5,049 has a 5 in thousands place, a 0 in hundreds place, a 4 in tens place, and a 9 in ones place

$$5{,}049 = 5 \bullet (1{,}000) + 0 \bullet (100) + 4 \bullet (10) + 9 \bullet (1).$$

# Application: Number Systems and Circuits for Addition

Using exponential notation:

$$5{,}049 = 5 \bullet 10^3 + 0 \bullet 10^2 + 4 \bullet 10^1 + 9 \bullet 10^0.$$

Decimal notation is based on the fact that any positive integer can be written uniquely as a sum of products of the form

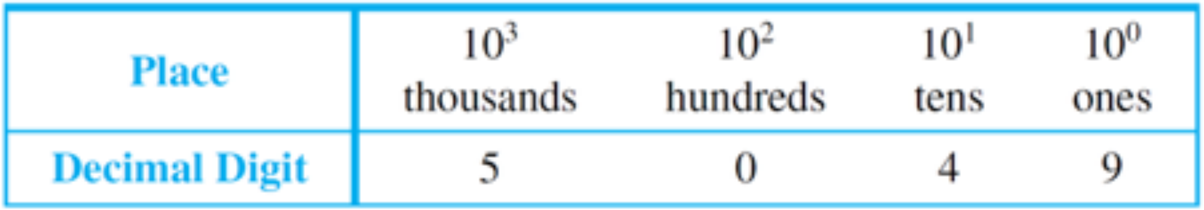
$$d \bullet 10^n,$$

where each $n$ is a nonnegative integer and each $d$ is one of the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

# Application: Number Systems and Circuits for Addition

The word *decimal* comes from Latin root *deci*, meaning “ten.” Decimal (or base 10) notation expresses a number as a string of digits in which each digit’s position indicates the power of 10 by which it is multiplied.

right-most position is the ones place (or $10^0$ place), to the left of that is the tens place (or $10^1$ place), to the left of that is the hundreds place (or $10^2$ place), and so forth, as illustrated below.



| **Place** | $10^3$<br>thousands | $10^2$<br>hundreds | $10^1$<br>tens | $10^0$<br>ones |
|---|---|---|---|---|
| **Decimal Digit** | 5 | 0 | 4 | 9 |

# Binary Representation of Numbers

**base 2 notation,** or **binary notation,** is of special importance because signals used in modern electronics are always in one of only two states. (Latin root *bi* means “two.”)

We can show that any integer can be represented uniquely as a sum of products of the form

$$d \bullet 2^n,$$

where each $n$ is an integer and each $d$ is one of the binary digits (or bits) 0 or 1.

# Binary Representation of Numbers

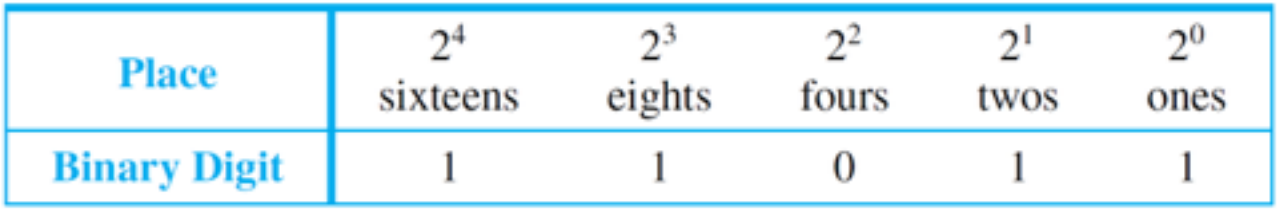
$$\begin{aligned} 27 &= 16 + 8 + 2 + 1 \\ &= 1 \bullet 2^4 + 1 \bullet 2^3 + 0 \bullet 2^2 + 1 \bullet 2^1 + 1 \bullet 2^0. \end{aligned}$$

places in binary notation correspond to various powers of 2.

27 in binary is 11011

# Binary Representation of Numbers

The right-most position is the ones place (or $2^0$ place), to the left of that is the twos place (or $2^1$ place), to the left of that is the fours place (or $2^2$ place), and so forth, as illustrated below.



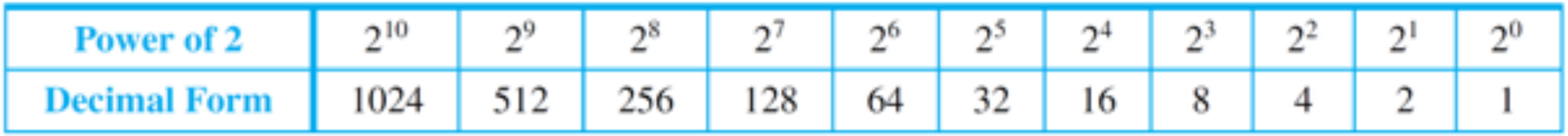
| Place | $2^4$<br>sixteens | $2^3$<br>eights | $2^2$<br>fours | $2^1$<br>twos | $2^0$<br>ones |
|---|---|---|---|---|---|
| Binary Digit | 1 | 1 | 0 | 1 | 1 |

leading zeros may be added or dropped as desired.

$$003_{10} = 3_{10} = 1 \cdot 2^1 + 1 \cdot 2^0 = 11_2 = 011_2.$$

# Binary Representation of Numbers

A list of powers of 2 is useful for doing binary-to-decimal and decimal-to-binary conversions.



| **Power of 2** | $2^{10}$ | $2^9$ | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Decimal Form** | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

Powers of 2

# *Converting a Binary to a Decimal Number*

Represent $110101_2$ in decimal notation.

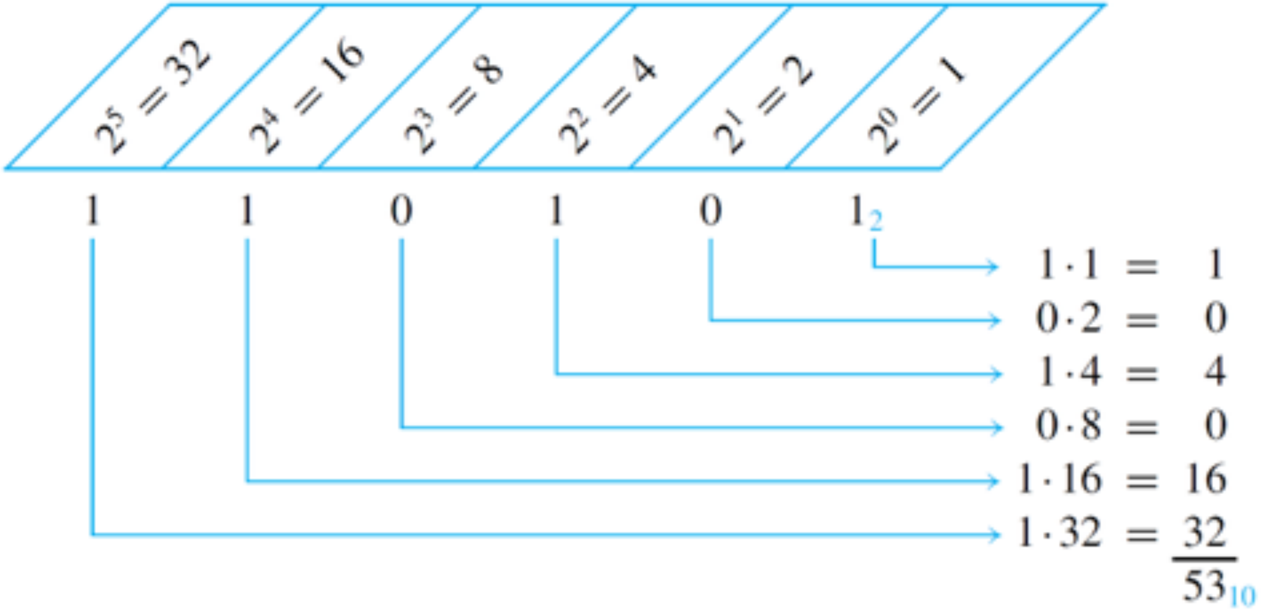
Solution:

$$
\begin{aligned}
110101_2 &= 1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 \\
&= 32 + 16 + 4 + 1 \\
&= 53_{10}
\end{aligned}
$$

# Example 2 – *Solution*

cont'd

Alternatively, the schema below may be used.

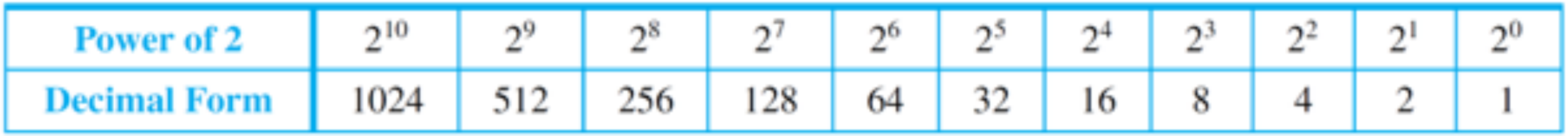


# Example 3 – *Converting a Decimal to a Binary Number*

Represent 209 in binary notation.

Solution:

Use Table 2.5.1 to write 209 as a sum of powers of 2, starting with the highest power of 2 that is less than 209 and continuing to lower powers.



| Power of 2 | $2^{10}$ | $2^9$ | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Decimal Form | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

Powers of 2

**Table 2.5.1**

# Example 3 – *Solution*

cont'd

Since 209 is between 128 and 256, the highest power of 2 that is less than 209 is 128. Hence

$$209_{10} = 128 + \text{a smaller number.}$$

Now 209 − 128 = 81, and 81 is between 64 and 128, so the highest power of 2 that is less than 81 is 64. Hence

$$209_{10} = 128 + 64 + \text{a smaller number.}$$

# Example 3 – *Solution*

cont'd

Continuing in this way, you obtain

$$209_{10} = 128 + 64 + 16 + 1$$

$$= 1\cdot 2^7 + 1\cdot 2^6 + 0\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0.$$

For each power of 2 that occurs in the sum, there is a 1 in the corresponding position of the binary number.

# Example 3 – *Solution*

cont'd

For each power of 2 that is missing from the sum, there is a 0 in the corresponding position of the binary number.

Thus

$$209_{10} = 11010001_2$$

# Binary Addition and Subtraction

# Example 4 – *Addition in Binary Notation*

Add $1101_2$ and $111_2$ using binary notation.

Solution:

Because $2_{10} = 10_2$ and $1_{10} = 1_2$, the translation of $1_{10} + 1_{10} = 2_{10}$ to binary notation is

$$\begin{array}{rr} & 1_2 \\ + & 1_2 \\ \hline & 10_2 \end{array}$$

It follows that adding two 1's together results in a carry of 1 when binary notation is used.

# Example 4 – *Solution*

cont’d

Adding three 1’s together also results in a carry of 1 since $3_{10} = 11_2$ (“one one base two”).

$$\begin{array}{cr} & 1_2 \\ + & 1_2 \\ + & 1_2 \\ \hline & 11_2 \end{array}$$

Thus the addition can be performed as follows:

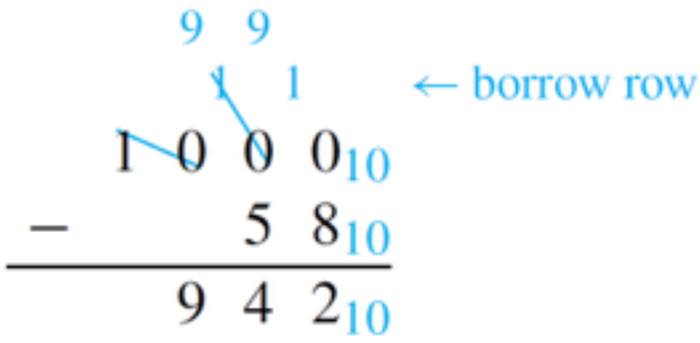
$$\begin{array}{cccccrl} & & 1 & 1 & 1 & & \leftarrow \text{carry row} \\ & & 1 & 1 & 0 & 1_2 & \\ + & & & 1 & 1 & 1_2 & \\ \hline & 1 & 0 & 1 & 0 & 0_2 & \end{array}$$

# Example 5 – *Subtraction in Binary Notation*

Subtract $1011_2$ from $11000_2$ using binary notation.

Solution:

In decimal subtraction the fact that $10_{10} - 1_{10} = 9_{10}$ is used to borrow across several columns. For example, consider the following:



$$\begin{array}{cccccl} & & 9 & 9 & & \\ & & & \cancel{1} & 1 & \leftarrow \text{borrow row} \\ & \cancel{1} & \cancel{0} & \cancel{0} & 0_{10} & \\ - & & & 5 & 8_{10} & \\ \hline & & 9 & 4 & 2_{10} & \end{array}$$

# Example 5 – *Solution*

cont'd

In binary subtraction it may also be necessary to borrow across more than one column. But when you borrow a $1_2$ from $10_2$, what remains is $1_2$.

$$\begin{array}{rr} & 10_2 \\ - & 1_2 \\ \hline & 1_2 \end{array}$$

Thus the subtraction can be performed as follows:

$$\begin{array}{cccccc} & & 0 & 1 & 1 & \\ & & & \cancel{1} & \cancel{1} & 1 \quad \leftarrow \text{borrow row} \\ & 1 & \cancel{1} & \cancel{0} & \cancel{0} & 0_2 \\ - & & 1 & 0 & 1 & 1_2 \\ \hline & & 1 & 1 & 0 & 1_2 \end{array}$$

# Circuits for Computer Addition

# Circuits for Computer Addition

Consider the question of designing a circuit to produce the sum of two binary digits *P* and *Q*. Both *P* and *Q* can be either 0 or 1. And the following facts are known:

$$\begin{aligned}
1_2 + 1_2 & & &= 10_2, \\
1_2 + 0_2 &= 1_2 & &= 01_2, \\
0_2 + 1_2 &= 1_2 & &= 01_2, \\
0_2 + 0_2 &= 0_2 & &= 00_2.
\end{aligned}$$

# Circuits for Computer Addition

It follows that the circuit to be designed must have two outputs—one for the left binary digit (this is called the **carry**) and one for the right binary digit (this is called the **sum**).

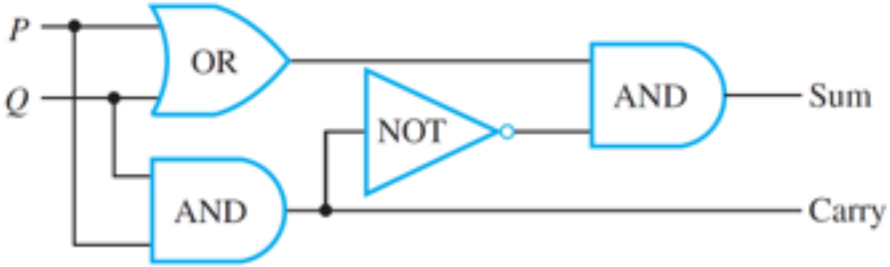
The carry output is 1 if both *P* and *Q* are 1; it is 0 otherwise. Thus the carry can be produced using the AND-gate circuit that corresponds to the Boolean expression $P \wedge Q$. The sum output is 1 if either *P* or *Q*, but not both, is 1.

# Circuits for Computer Addition

The sum can, therefore, be produced using a circuit that corresponds to the Boolean expression for *exclusive or*: $(P \vee Q) \wedge \sim(P \wedge Q)$. Hence, a circuit to add two binary digits $P$ and $Q$ can be constructed as in Figure 2.5.1. This circuit is called a **half-adder.**

**HALF-ADDER**

**Circuit**



**Input/Output Table**

| $P$ | $Q$ | Carry | Sum |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

Circuit to Add $P + Q$, Where $P$ and $Q$ Are Binary Digits

**Figure 2.5.1**

# Circuits for Computer Addition

In order to construct a circuit that will add multidigit binary numbers, it is necessary to incorporate a circuit that will compute the sum of three binary digits. Such a circuit is called a **full-adder.**

Consider a general addition of three binary digits *P*, *Q*, and *R* that results in a carry (or left-most digit) *C* and a sum (or right-most digit) *S*.

$$\begin{array}{rr} & P \\ + & Q \\ + & R \\ \hline & CS \end{array}$$

# Circuits for Computer Addition

The operation of the full-adder is based on the fact that addition is a binary operation: Only two numbers can be added at one time. Thus *P* is first added to *Q* and then the result is added to *R*. For instance, consider the following addition:

$$\left.\begin{array}{r}
\left.\begin{array}{r} 1_2 \\ +\ 0_2 \end{array}\right\} 1_2 + 0_2 = 01_2 \\
+\ 1_2 \hphantom{\} 1_2 + 0_2 = 01_2}
\end{array}\right\} 1_2 + 1_2 = 10_2$$

$$\begin{array}{r} \hline 10_2 \end{array}$$

# Circuits for Computer Addition

The process illustrated here can be broken down into steps that use half-adder circuits.

**Step 1:** Add *P* and *Q* using a half-adder to obtain a binary number with two digits.

$$\begin{array}{lr} & P \\ + & Q \\ \hline & C_1 S_1 \end{array}$$

# Circuits for Computer Addition

**Step 2:** Add *R* to the sum $C_1S_1$ of *P* and *Q*.

$$\begin{array}{r} C_1S_1 \\ +\quad R \\ \hline \end{array}$$

To do this, proceed as follows:

**Step 2a:** Add *R* to $S_1$ using a half-adder to obtain the two-digit number $C_2S$.

$$\begin{array}{r} S_1 \\ +\quad R \\ \hline C_2S \end{array}$$

Then *S* is the right-most digit of the entire sum of *P*, *Q*, and *R*.

# Circuits for Computer Addition

**Step 2b:** Determine the left-most digit, $C$, of the entire sum as follows: First note that it is impossible for both $C_1$ and $C_2$ to be 1's. For if $C_1 = 1$, then $P$ and $Q$ are both 1, and so $S_1 = 0$. Consequently, the addition of $S_1$ and $R$ gives a binary number $C_2S_1$ where $C_2 = 0$.

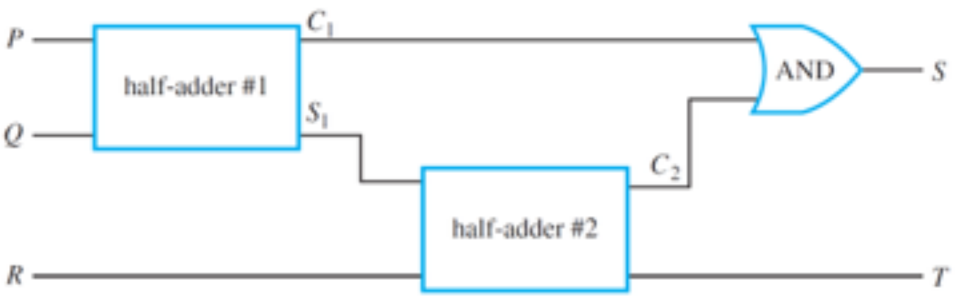
Next observe that $C$ will be a 1 in the case that the addition of $P$ and $Q$ gives a carry of 1 or in the case that the addition of $S_1$ (the right-most digit of $P + Q$) and $R$ gives a carry of 1.

# Circuits for Computer Addition

In other words, $C$ = 1 if, and only if, $C_1$ = 1 or $C_2$ = 1. It follows that the circuit shown in Figure 2.5.2 will compute the sum of three binary digits.

FULL-ADDER

Circuit



Input/Output Table

| $P$ | $Q$ | $R$ | $C$ | $S$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 |

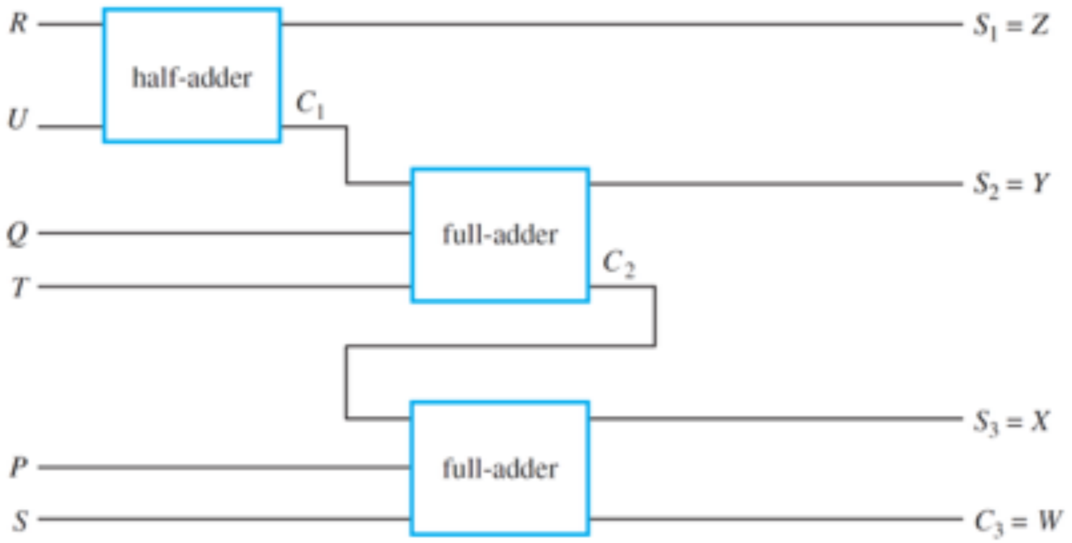
Circuit to Add $P + Q + R$, Where $P$, $Q$, and $R$ Are Binary Digits

**Figure 2.5.2**

# Circuits for Computer Addition

Two full-adders and one half-adder can be used together to build a circuit that will add two three-digit binary numbers *PQR* and *STU* to obtain the sum *WXYZ*. This is illustrated in Figure 2.5.3. Such a circuit is called a **parallel adder.**

Parallel adders can be constructed to add binary numbers of any finite length.



A Parallel Adder to Add *PQR* and *STU* to Obtain *WXYZ*

**Figure 2.5.3**

# Two’s Complements and the Computer Representation of Negative Integers

# Two’s Complements and the Computer Representation of Negative Integers

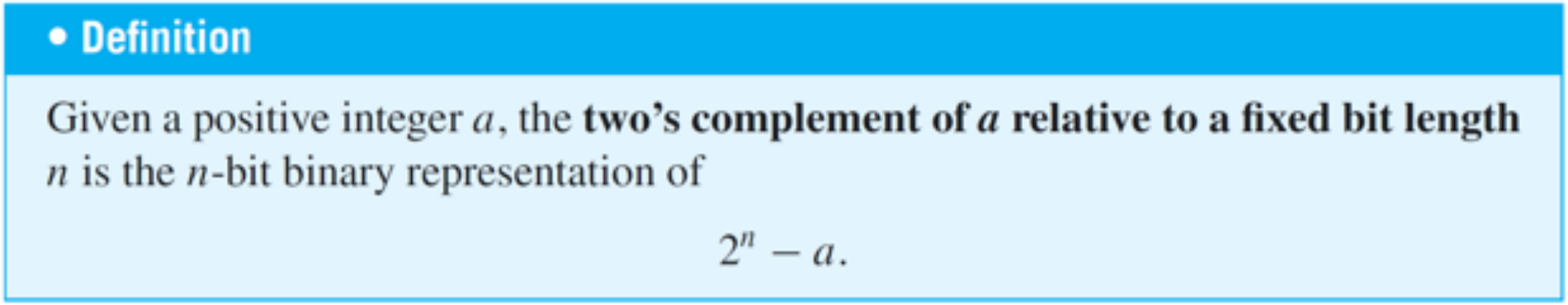
Typically, a fixed number of bits is used to represent integers on a computer, and these are required to represent negative as well as nonnegative integers.

Sometimes a particular bit, normally the left-most, is used as a sign indicator, and the remaining bits are taken to be the absolute value of the number in binary notation.

The problem with this approach is that the procedures for adding the resulting numbers are somewhat complicated and the representation of 0 is not unique.

# Two’s Complements and the Computer Representation of Negative Integers

A more common approach, using *two’s complements*, makes it possible to add integers quite easily and results in a unique representation for 0. The two’s complement of an integer relative to a fixed bit length is defined as follows:



- **Definition**

Given a positive integer $a$, the **two’s complement of $a$ relative to a fixed bit length $n$** is the $n$-bit binary representation of

$$2^n - a.$$

# Two’s Complements and the Computer Representation of Negative Integers

There is a convenient way to compute two’s complements that involves less arithmetic than direct application of the definition. For an 8-bit representation, it is based on three facts:

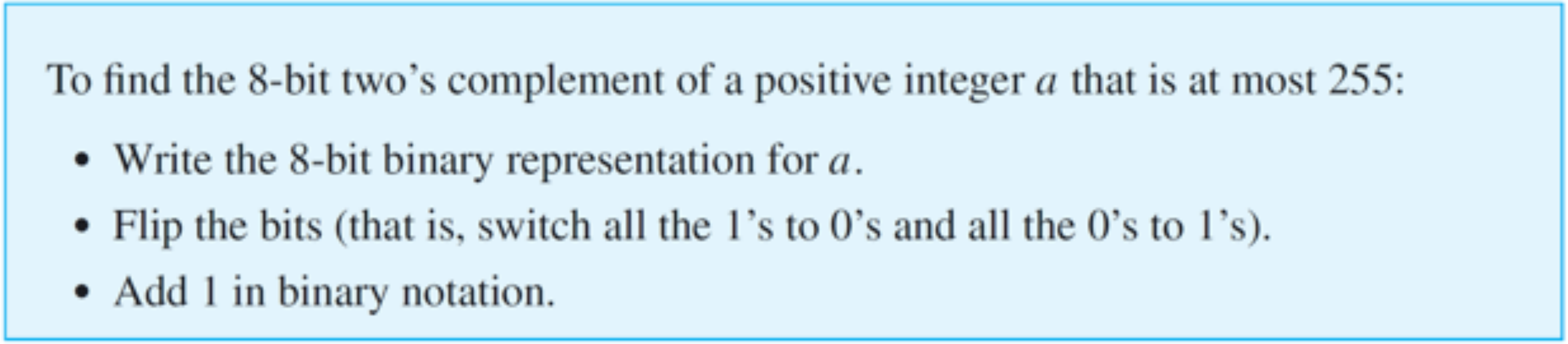
**1.** $2^8 - a = [(2^8 - 1) - a] + 1.$

**2.** The binary representation of $2^8 - 1$ is $11111111_2$.

**3.** Subtracting an 8-bit binary number *a* from $11111111_2$ just switches all the 0’s in *a* to 1’s and all the 1’s to 0’s. (The resulting number is called the **one’s complement** of the given number.)

# Two’s Complements and the Computer Representation of Negative Integers

In general,



To find the 8-bit two’s complement of a positive integer $a$ that is at most 255:

- Write the 8-bit binary representation for $a$.
- Flip the bits (that is, switch all the 1’s to 0’s and all the 0’s to 1’s).
- Add 1 in binary notation.

# Example 6 – *Finding a Two's Complement*

Find the 8-bit two's complement of 19.

Solution:

Write the 8-bit binary representation for 19, switch all the 0's to 1's and all the 1's to 0's, and add 1.

$$19_{10} = (16 + 2 + 1)_{10}$$

$$= 00010011_2 \xrightarrow{\text{flip the bits}} 11101100 \xrightarrow{\text{add 1}} 11101101$$

# Example 6 – *Solution*

cont’d

To check this result, note that

$$
\begin{aligned}
11101101_2 &= (128 + 64 + 32 + 8 + 4 + 1)_{10} \\
&= 237_{10} \\
&= (256 - 19)_{10} \\
&= (2^8 - 19)_{10},
\end{aligned}
$$

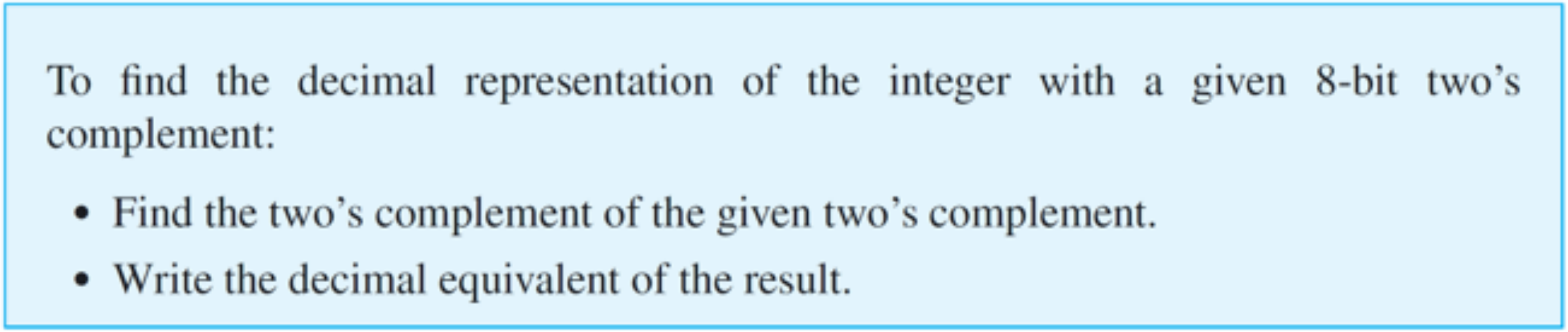
which *is* the two’s complement of 19.

# Two’s Complements and the Computer Representation of Negative Integers

Observe that because

$$2^8 - (2^8 - a) = a$$

the *two’s complement of the two’s complement of a number is the number itself*, and therefore,



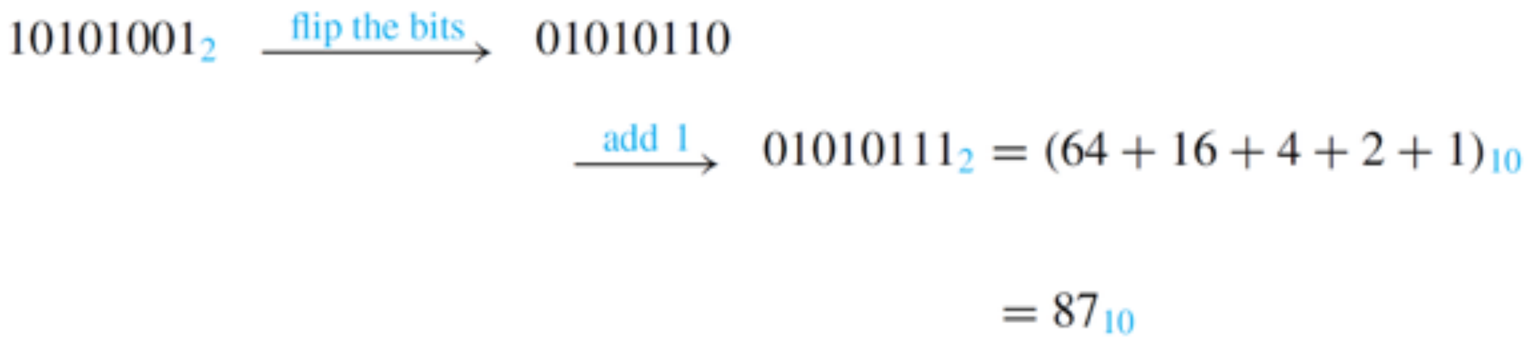
To find the decimal representation of the integer with a given 8-bit two’s complement:

- Find the two’s complement of the given two’s complement.
- Write the decimal equivalent of the result.

# Example 7 – *Finding a Number with a Given Two's Complement*

What is the decimal representation for the integer with two's complement 10101001?

Solution:



$$10101001_2 \xrightarrow{\text{flip the bits}} 01010110$$

$$\xrightarrow{\text{add 1}} 01010111_2 = (64 + 16 + 4 + 2 + 1)_{10}$$

$$= 87_{10}$$

# Example 7 – *Solution*

cont'd

To check this result, note that the given number is

$$\begin{aligned} 10101001_2 &= (128 + 32 + 8 + 1)_{10} \\ &= 169_{10} \\ &= (256 - 87)_{10} \\ &= (2^8 - 87)_{10}, \end{aligned}$$

which is the two's complement of 87.

# 8-Bit Representation of a Number

# 8-Bit Representation of a Number

Now consider the two’s complement of an integer *n* that satisfies the inequality $1 \le n \le 128$. Then

$$-1 \geq -n \geq -128$$

because multiplying by $-1$ reverses the direction of the inequality

and

$$2^8 - 1 \geq 2^8 - n \geq 2^8 - 128$$

by adding $2^8$ to all parts of the inequality.

But $2^8 - 128 = 256 - 128 = 128 = 2^7$. Hence

$$2^7 \leq \text{the two's complement of } n < 2^8.$$

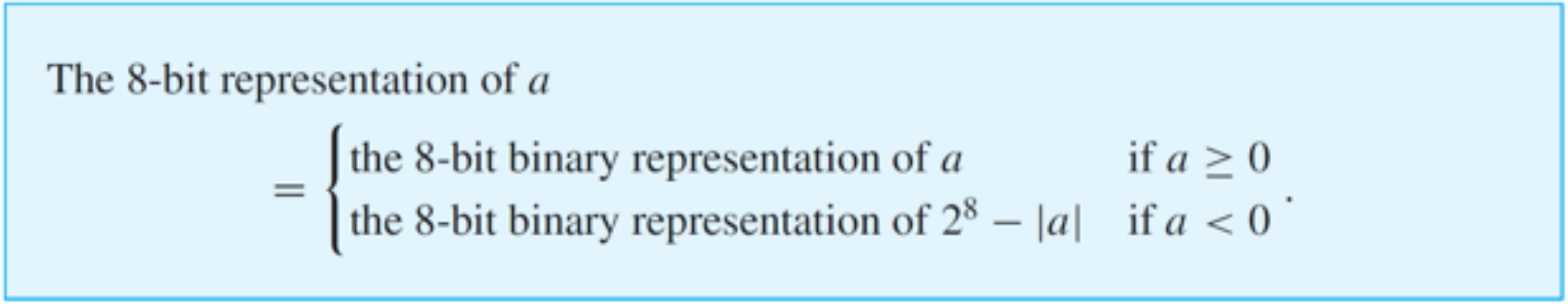
# 8-Bit Representation of a Number

It follows that the 8-bit two’s complement of an integer from 1 through 128 has a leading bit of 1. Note also that the ordinary 8-bit representation of an integer from 0 through 127 has a leading bit of 0.

Consequently, eight bits can be used to represent both nonnegative and negative integers by representing each nonnegative integer up through 127 using ordinary 8-bit binary notation and representing each negative integer from −1 through −128 as the two’s complement of its absolute value.

# 8-Bit Representation of a Number

That is, for any integer *a* from −128 through 127,

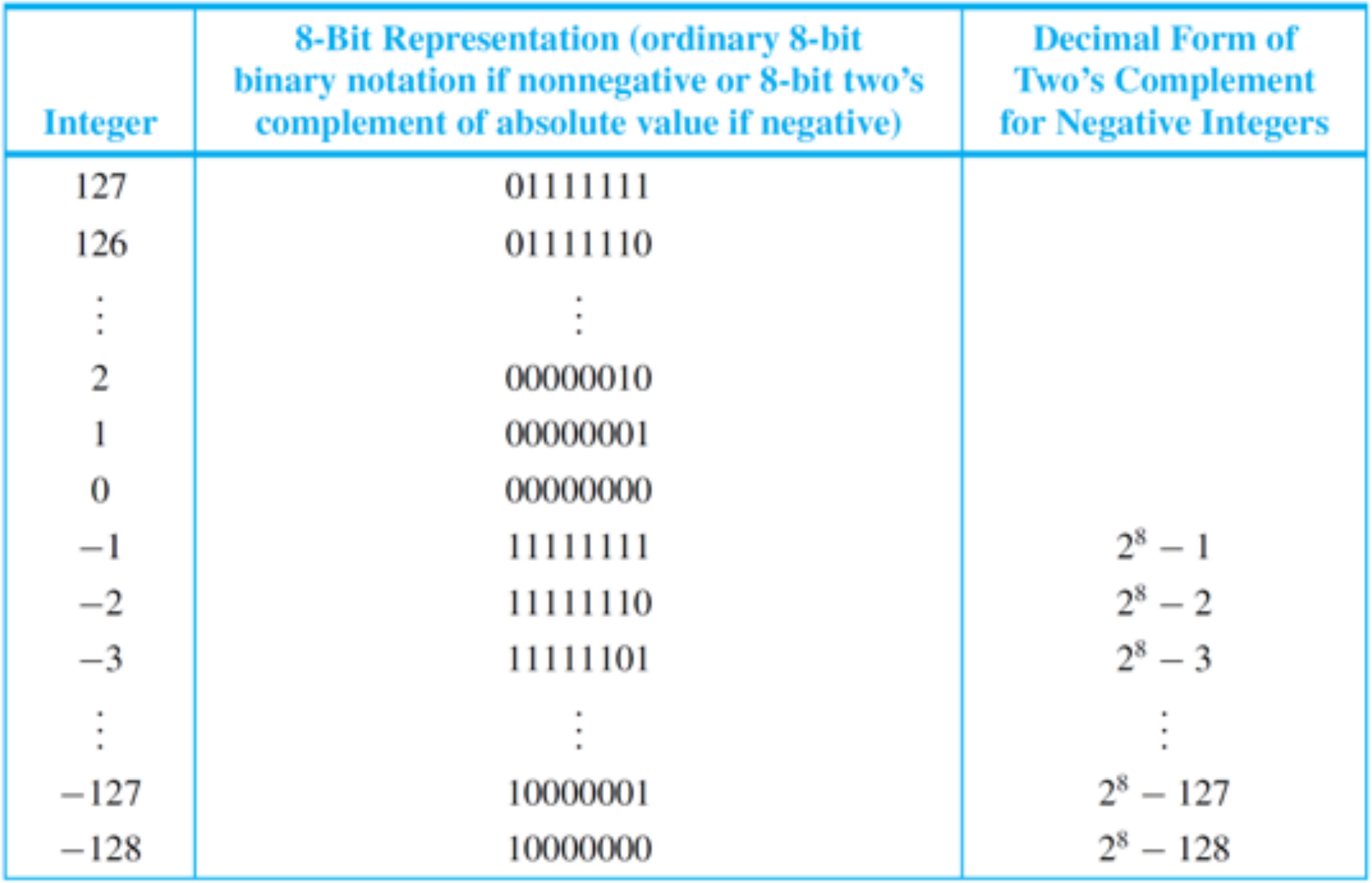


The 8-bit representation of $a$

$$= \begin{cases} \text{the 8-bit binary representation of } a & \text{if } a \geq 0 \\ \text{the 8-bit binary representation of } 2^8 - |a| & \text{if } a < 0 \end{cases}.$$

# 8-Bit Representation of a Number

The representations are illustrated in Table 2.5.2.

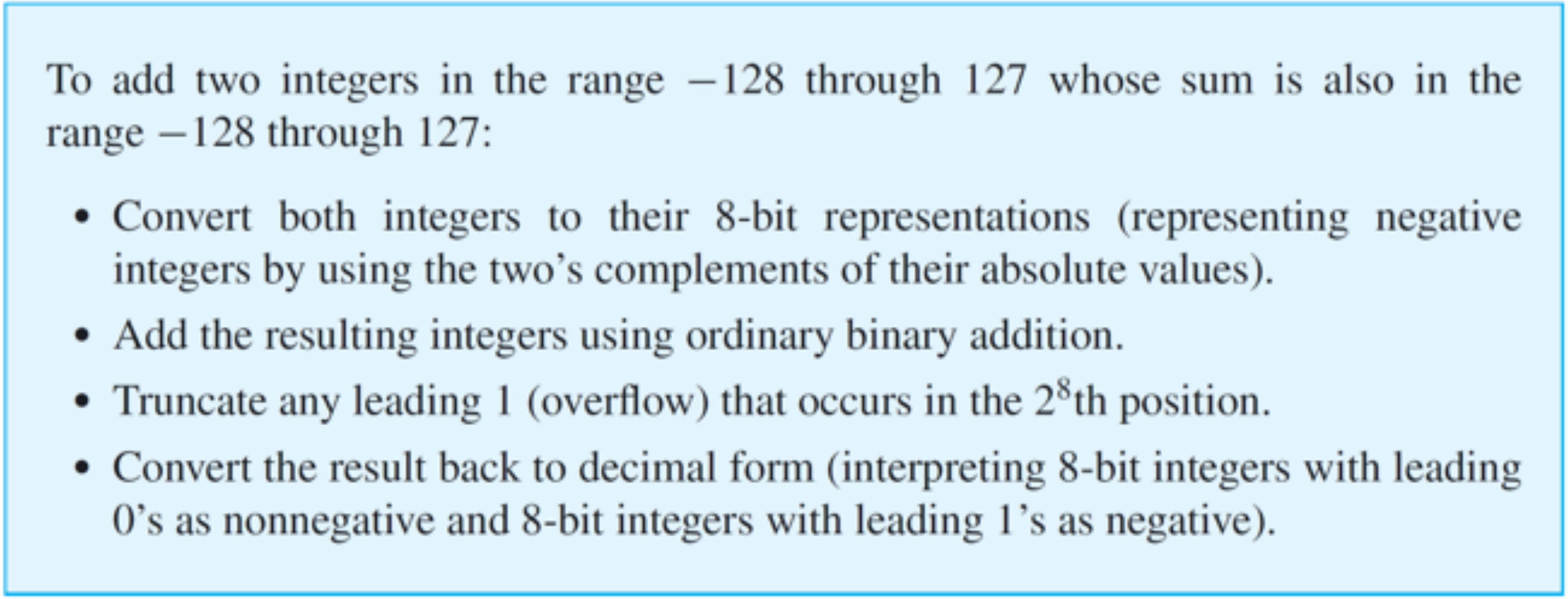


| Integer | 8-Bit Representation (ordinary 8-bit binary notation if nonnegative or 8-bit two's complement of absolute value if negative) | Decimal Form of Two's Complement for Negative Integers |
|---|---|---|
| 127 | 01111111 | |
| 126 | 01111110 | |
| ⋮ | ⋮ | |
| 2 | 00000010 | |
| 1 | 00000001 | |
| 0 | 00000000 | |
| $-1$ | 11111111 | $2^8 - 1$ |
| $-2$ | 11111110 | $2^8 - 2$ |
| $-3$ | 11111101 | $2^8 - 3$ |
| ⋮ | ⋮ | ⋮ |
| $-127$ | 10000001 | $2^8 - 127$ |
| $-128$ | 10000000 | $2^8 - 128$ |

**Table 2.5.2**

# Computer Addition with Negative Integers

# Computer Addition with Negative Integers



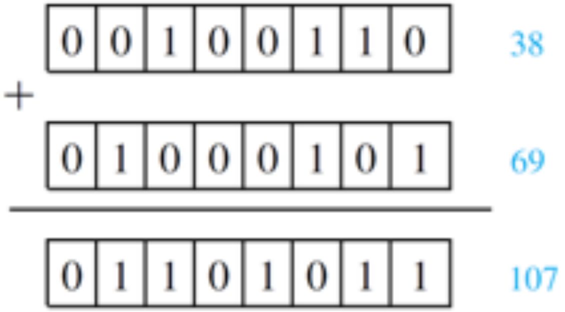
To add two integers in the range −128 through 127 whose sum is also in the range −128 through 127:

- Convert both integers to their 8-bit representations (representing negative integers by using the two's complements of their absolute values).
- Add the resulting integers using ordinary binary addition.
- Truncate any leading 1 (overflow) that occurs in the $2^8$th position.
- Convert the result back to decimal form (interpreting 8-bit integers with leading 0's as nonnegative and 8-bit integers with leading 1's as negative).

# Computer Addition with Negative Integers

***Case 1, (both integers are nonnegative):*** This case is easy because if two nonnegative integers from 0 through 127 are written in their 8-bit representations and if their sum is also in the range 0 through 127, then the 8-bit representation of their sum has a leading 0 and is therefore interpreted correctly as a nonnegative integer.

The example below illustrates what happens when 38 and 69 are added.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 38 |
| + | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 69 |
| | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 107 |

# Computer Addition with Negative Integers

To be concrete, let the nonnegative integer be *a* and the negative integer be −*b* and suppose both *a* and −*b* are in the range −128 through 127. The crucial observation is that adding the 8-bit representations of *a* and −*b* is equivalent to computing

$$a + (2^8 - b)$$

because the 8-bit representation of −*b* is the binary representation of $2^8 - b$.

# Computer Addition with Negative Integers

***Case 2 (a is nonnegative and −b is negative and |a| < |b|):*** In this case, observe that a = |a| < |b| = b and

$$a + (2^8 - b) = 2^8 - (b - a),$$

and the binary representation of this number is the 8-bit representation of −(*b* − *a*) = *a* + (−*b*). We must be careful to check that $2^8 - (b - a)$ is between $2^7$ and $2^8$. But it *is* because

$$2^7 = 2^8 - 2^7 \le 2^8 - (b - a) < 2^8 \qquad \text{since } 0 < b - a \le \mathrm{b} \le 128 = 2^7.$$

Hence in case |*a*| < |*b*|, adding the 8-bit representations of *a* and −*b* gives the 8-bit representation of *a* + (−*b*).

# Example 8 – *Computing a + (−b) Where 0 ≤ a < b ≤ 128*

Use 8-bit representations to compute 39 + (−89).

Solution:

**Step 1:** Change from decimal to 8-bit representations using the two's complement to represent −89.

Since $39_{10} = (32 + 4 + 2 + 1)_{10} = 100111_2$, the 8-bit representation of 39 is 00100111.

Now the 8-bit representation of −89 is the two's complement of 89.

# Example 8 – *Solution*

cont'd

This is obtained as follows:

$$89_{10} = (64 + 16 + 8 + 1)_{10} = 01011001_2 \xrightarrow{\text{flip the bits}} 10100110 \xrightarrow{\text{add } 1} 10100111$$

So the 8-bit representation of −89 is 10100111.

# Example 8 – *Solution*

cont'd

**Step 2:** Add the 8-bit representations in binary notation and truncate the 1 in the $2^8$th position if there is one:

$$
\begin{array}{c|c|c|c|c|c|c|c|c|}
 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
+ & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ \hline
 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

There is no 1 in the $2^8$th position to truncate→

# Example 8 – *Solution*

cont’d

**Step 3:** Find the decimal equivalent of the result. Since its leading bit is 1, this number is the 8-bit representation of a negative integer.

$$11001110 \xrightarrow{\text{flip the bits}} 00110001 \xrightarrow{\text{add 1}} 00110010$$
$$\leftrightarrow -(32+16+2)_{10} = -50_{10}$$

Note that since 39 − 89 = −50, this procedure gives the correct answer.

# Computer Addition with Negative Integers

***Case 3* (*a is nonnegative and −b is negative and* |*b*| ≤ |*a*|)*:*** In this case, observe that b = |b| ≤ |a| = a and

$$a + (2^8 - b) \;=\; 2^8 + (a - b).$$

Also

$$2^8 \;\leq\; 2^8 + (a - b) \;<\; 2^8 + 2^7 \qquad \text{because } 0 \leq a - b \leq \text{a} < 128 = 2^7.$$

So the binary representation of $a + (2^8 - b) \;=\; 2^8 + (a - b)$ has a leading 1 in the ninth ($2^8$th) position. This leading 1 is often called “overflow” because it does not fit in the 8-bit integer format.

# Computer Addition with Negative Integers

Now subtracting $2^8$ from $2^8 + (a - b)$ is equivalent to truncating the leading 1 in the $2^8$th position of the binary representation of the number. But

$$[a + (2^8 - b)] - 2^8 = 2^8 + (a - b) - 2^8 = a - b = a + (-b).$$

Hence in case $|a| \geq |b|$, adding the 8-bit representations of $a$ and $-b$ and truncating the leading 1 (which is sure to be present) gives the 8-bit representation of $a + (-b)$.

# Example 9 – *Computing $a + (-b)$ Where $1 \le b < a \le 127$*

Use 8-bit representations to compute 39 + (−25).

Solution:

**Step 1:** Change from decimal to 8-bit representations using the two's complement to represent −25.

As in Example 8, the 8-bit representation of 39 is 00100111. Now the 8-bit representation of −25 is the two's complement of 25, which is obtained as follows:

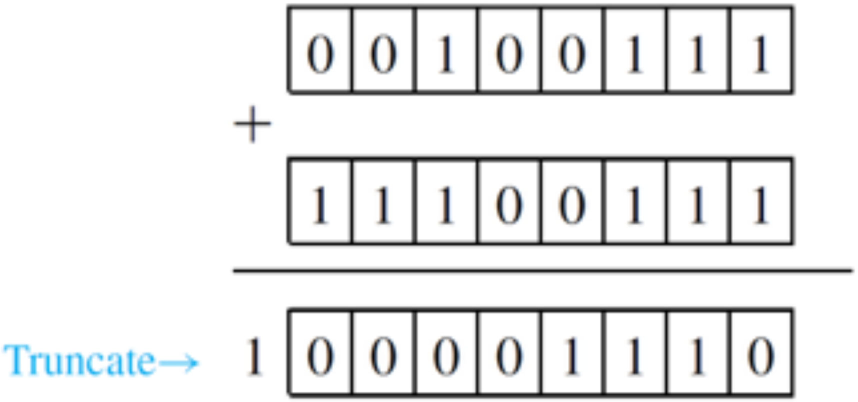
$$25_{10} = (16 + 8 + 1)_{10} = 00011001_2 \xrightarrow{\text{flip the bits}} 11100110 \xrightarrow{\text{add } 1} 11100111$$

# Example 9 – *Solution*

cont'd

So the 8-bit representation of −25 obtained as 11100111.

**Step 2:** Add the 8-bit representations in binary notation and truncate the 1 in the $2^8$th position if there is one:



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| + | | | | | | | | |
| | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| Truncate→ 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

# Example 9 – *Solution*

cont’d

**Step 3:** Find the decimal equivalent of the result:

$$00001110_2 = (8 + 4 + 2)_{10}$$

$$= 14_{10}.$$

Since 39 − 25 = 14, this is the correct answer.

# Computer Addition with Negative Integers

***Case 4 (both integers are negative):*** This case involves adding two negative integers in the range −1 through −128 whose sum is also in this range.

To be specific, consider the sum $(-a) + (-b)$ where $a$, $b$, and $a + b$ are all in the range 1 through 128. In this case, the 8-bit representations of $-a$ and $-b$ are the 8-bit representations of $2^8 - a$ and $2^8 - b$.

So if the 8-bit representations of $-a$ and $-b$ are added, the result is

$$(2^8 - a) + (2^8 - b) = [2^8 - (a + b)] + 2^8.$$

# Computer Addition with Negative Integers

We know that truncating a leading 1 in the ninth ($2^8$th) position of a binary number is equivalent to subtracting $2^8$.

So when the leading 1 is truncated from the 8-bit representation of $(2^8 - a) + (2^8 - b)$, the result is $2^8 - (a + b)$, which is the 8-bit representation of $-(a + b) = (-a) + (-b)$.

# Example 10 – *Computing (−a) + (−b) Where 1 ≤ a, b ≤ 128, and 1 ≤ a + b ≤ 128*

Use 8-bit representations to compute (−89) + (−25).

Solution:

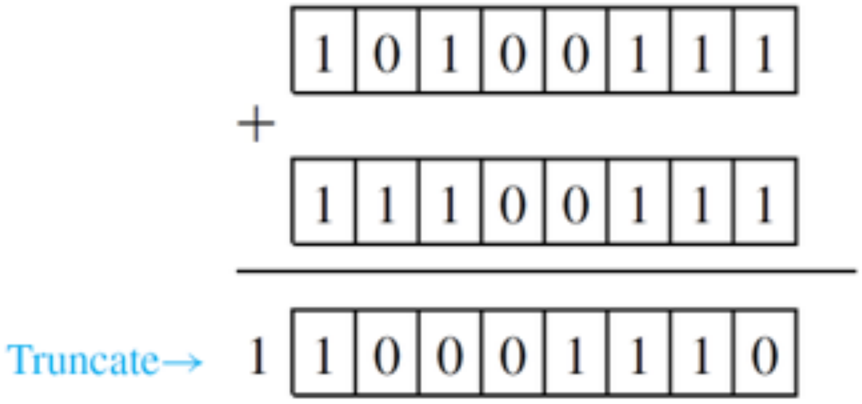
**Step 1:** Change from decimal to 8-bit representations using the two's complements to represent −89 and −25.

The 8-bit representations of −89 and −25 were shown in Examples 2.5.8 and 2.5.9 to be 10100111 and 11100111, respectively.

# Example 10 – *Solution*

cont’d

**Step 2:** Add the 8-bit representations in binary notation and truncate the 1 in the $2^8$th position if there is one:

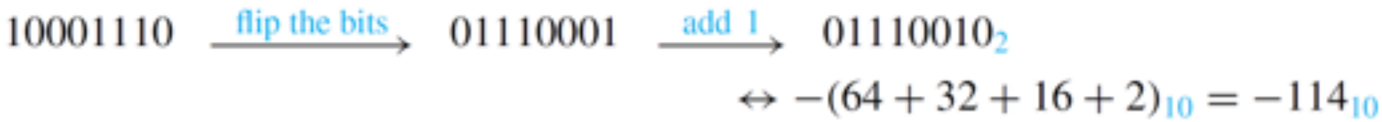


$$\begin{array}{rc|c|c|c|c|c|c|c|c|}
 & & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
+ & & & & & & & & & \\
 & & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
\hline
\text{Truncate}\rightarrow & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}$$

# Example 10 – *Solution*

cont’d

**Step 3:** Find the decimal equivalent of the result. Because its leading bit is 1, this number is the 8-bit representation of a negative integer.



$$10001110 \xrightarrow{\text{flip the bits}} 01110001 \xrightarrow{\text{add } 1} 01110010_2$$
$$\leftrightarrow -(64 + 32 + 16 + 2)_{10} = -114_{10}$$

Since (−89) + (−25) = −114, that is the correct answer.

# Hexadecimal Notation

# Hexadecimal Notation

**Hexadecimal notation** is even more compact than decimal notation, and it is much easier to convert back and forth between hexadecimal and binary notation than it is between binary and decimal notation.

The word *hexadecimal* comes from the Greek root *hex*-, meaning “six,” and the Latin root *deci*-, meaning “ten.” Hence *hexadecimal* refers to “sixteen,” and hexadecimal notation is also called **base 16 notation.**

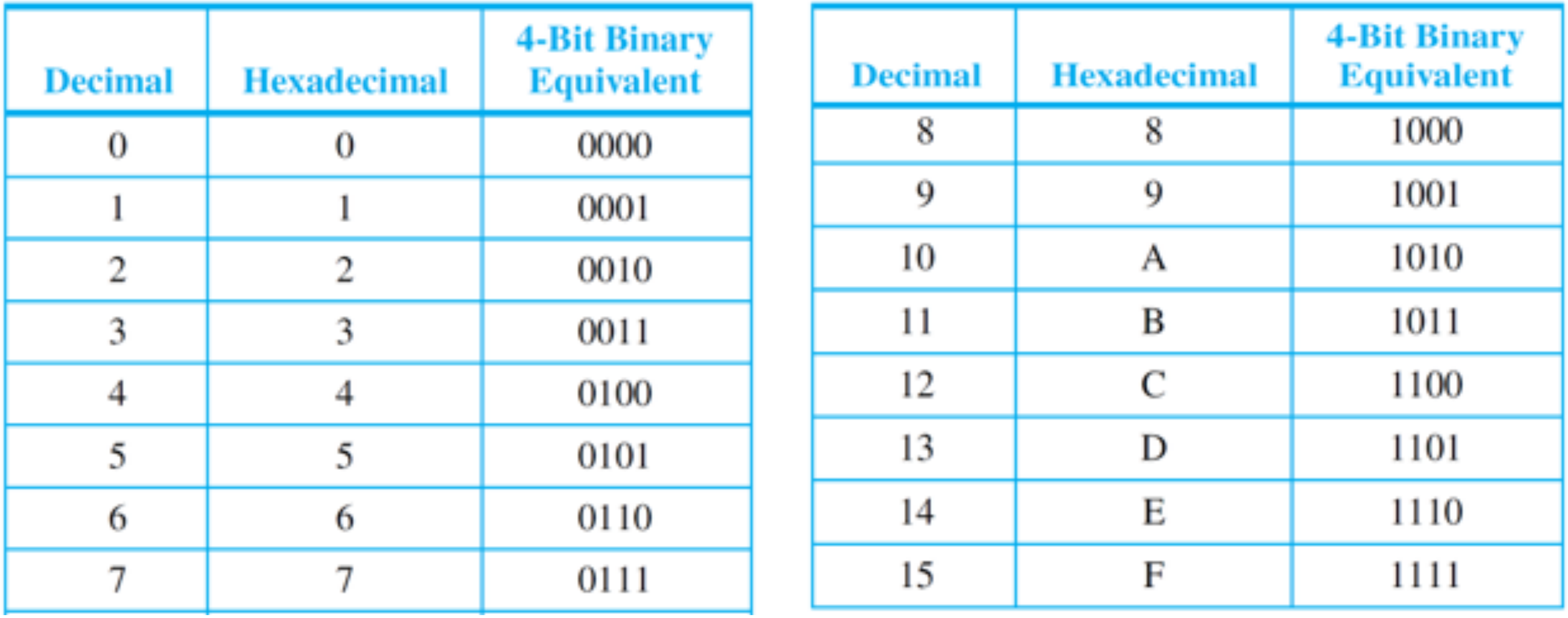
# Hexadecimal Notation

Hexadecimal notation is based on the fact that any integer can be uniquely expressed as a sum of numbers of the form

where each *n* is a nonneg $d \cdot 16^n$, teger and each *d* is one of the integers from 0 to 15. In order to avoid ambiguity, each hexadecimal digit must be represented by a single symbol. The integers 10 through 15 are represented by the symbols A, B, C, D, E, and F.

# Hexadecimal Notation

The sixteen hexadecimal digits are shown in Table 2.5.3, together with their decimal equivalents and, for future reference, their 4-bit binary equivalents.



| Decimal | Hexadecimal | 4-Bit Binary Equivalent |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

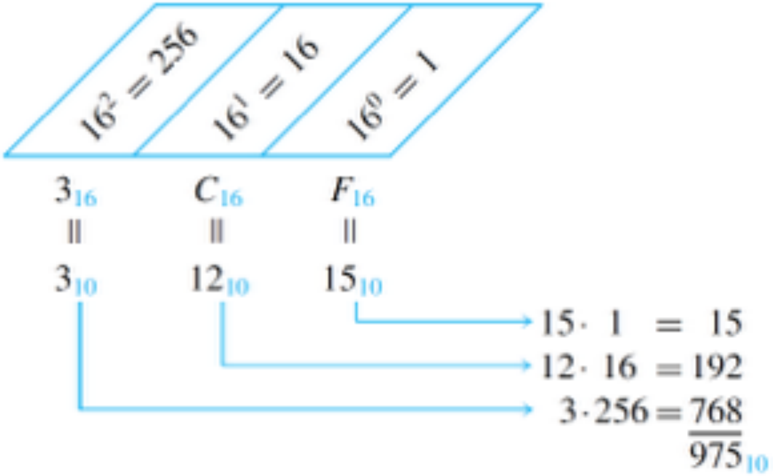
**Table 2.5.3**

# Example 11 – *Converting from Hexadecimal to Decimal Notation*

Convert $3CF_{16}$ to decimal notation.

Solution:

Consider the following schema.

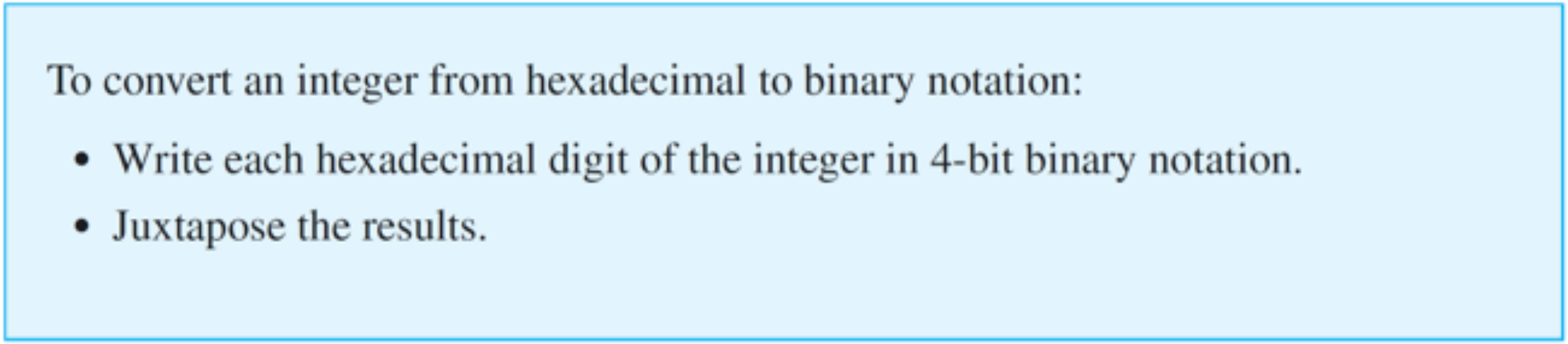


So $3CF_{16} = 975_{10}$.

# Hexadecimal Notation

Now consider how to convert from hexadecimal to binary notation.

The following sequence of steps will give the required conversion from hexadecimal to binary notation.



To convert an integer from hexadecimal to binary notation:

- Write each hexadecimal digit of the integer in 4-bit binary notation.
- Juxtapose the results.

# Example 12 – *Converting from Hexadecimal to Binary Notation*

Convert B09F$_{16}$ to binary notation.

Solution:

$$B_{16} = 11_{10} = 1011_2,$$

$$0_{16} = 0_{10} = 0000_2,$$

$$9_{16} = 9_{10} = 1001_2,$$

and

$$F_{16} = 15_{10} = 1111_2.$$

# Example 12 – *Solution*

cont’d

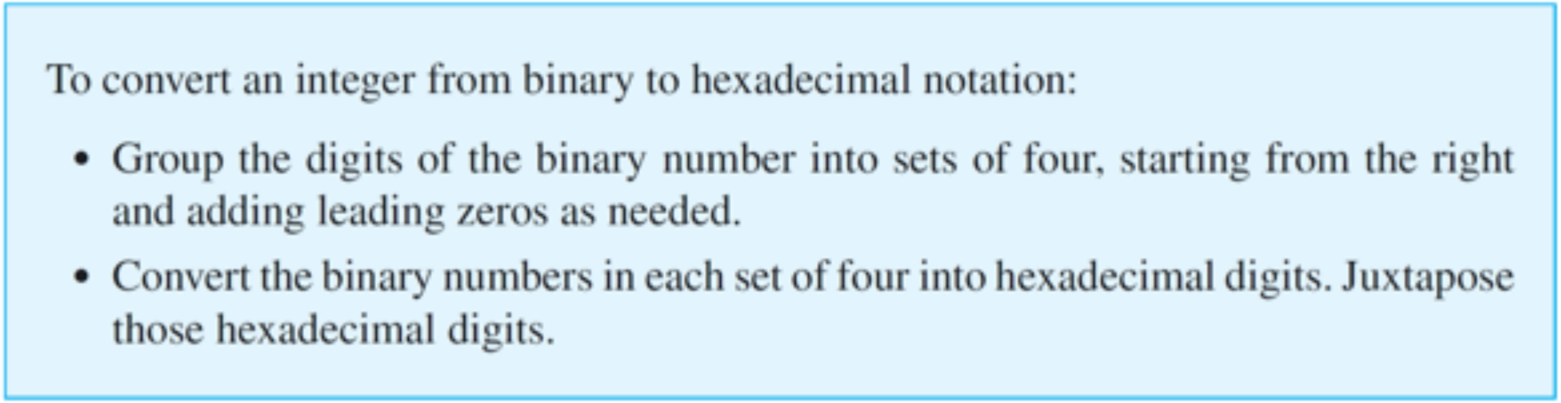
Consequently,

$$\begin{array}{cccc} \mathrm{B} & 0 & 9 & \mathrm{F} \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow \\ 1011 & 0000 & 1001 & 1111 \end{array}$$

and the answer is $1011000010011111_2$.

# Hexadecimal Notation

To convert integers written in binary notation into hexadecimal notation, reverse the steps of the previous procedure.



To convert an integer from binary to hexadecimal notation:

- Group the digits of the binary number into sets of four, starting from the right and adding leading zeros as needed.
- Convert the binary numbers in each set of four into hexadecimal digits. Juxtapose those hexadecimal digits.

# Example 13 – *Converting from Binary to Hexadecimal Notation*

Convert $100110110101001_2$ to hexadecimal notation.

Solution:

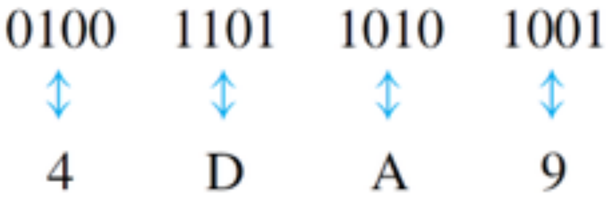
First group the binary digits in sets of four, working from right to left and adding leading 0’s if necessary.

0100 1101 1010 1001.

# Example 13 – *Solution*

cont’d

Convert each group of four binary digits into a hexadecimal digit.



| 0100 | 1101 | 1010 | 1001 |
|---|---|---|---|
| ↕ | ↕ | ↕ | ↕ |
| 4 | D | A | 9 |

Then juxtapose the hexadecimal digits.

$$4DA9_{16}$$