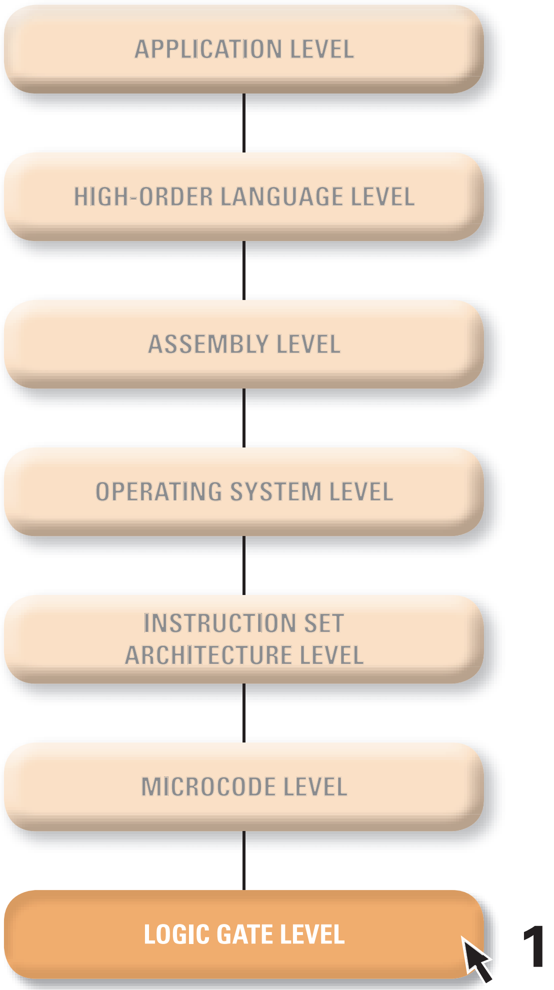
LEVEL 1

Logic Gate



CHAPTER 10

Combinational Circuits

TABLE OF CONTENTS

Finally we come to the lowest level in our description of the typical computer system. Each level of abstraction hides the details that are unnecessary for the user at the next-higher level. The details at Level LG1 are hidden from the user at Level ISA3, the instruction set architecture level. Remember that the user at Level ISA3 sees a von Neumann machine whose language is machine language. The job of the designer at Level LG1 is to construct the Level ISA3 machine. These last three chapters describe the language and design principles at Level LG1 that are required to construct a von Neumann machine.

Omitting Level Mc2

The figures in this text consistently show the microcode level between the instruction set architecture level and the logic gate level. Some designers choose to omit the microcode level in their machines and construct the Level ISA3 machine directly from Level LG1. Others choose to design their systems with a microcode level.

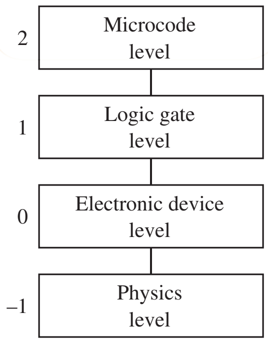
What are the advantages and disadvantages of each design approach? The same as we encountered at Levels 7, 6, and 5. Suppose you need to design an application for a user at Level App7. Would you rather write it in C at Level HOL6 and compile it to a lower level or write it directly in Pep/9 assembly language at Level Asmb5? Because C is at a higher level of abstraction, one C statement can do the work of many Pep/9 statements. The C program would be much shorter than the equivalent Pep/9 program. It would, therefore, be easier to design and debug. But it would require a compiler for translation to a lower level. Furthermore, a good assembly language programmer can usually produce shorter, faster code than the object code from even an optimizing compiler. Though the program would execute faster, it would be difficult to design and debug; it would thus be more costly to develop.

The tradeoff at Levels 7, 6, and 5 is development cost versus execution speed. The same tradeoff applies at Levels 3, 2, and 1. Generally, systems that include Level Mc2 are simpler and less costly than those that omit it. But they usually execute more slowly than if they were built directly from Level LG1. A recent design trend is to build simple but fast von Neumann machines with small instruction sets, called *reduced instruction set computers* (RISCs). An important characteristic of a RISC machine is its omission of Level Mc2.

Two levels that are interesting but whose descriptions are not given in this text are the levels below the logic gate level, as FIGURE 10.1 shows. At the electronic device level (Level 0), designers connect transistors, resistors, and capacitors to make an individual logic gate at Level LG1. At the physics level (Level –1), applied physicists construct the transistors that the electrical engineer can use to construct the gates that the computer architect can use to construct the von Neumann machine. There is no level below physics, the most fundamental of all the sciences.

The languages at Levels 0 and –1 are the set of mathematical equations that model the behavior of the objects at that level. You may be familiar with some of them. At Level 0 they include Ohm's law, Kirchoff's rules, and the voltage versus current characteristics of electronic devices. At Level –1 they include Coulomb's law, Newton's laws, and some laws from quantum mechanics. At all the levels, from the calculus for relational databases at Level App7 to Newton's laws at Level –1, formal mathematics is the tool for modeling the behavior of the system.

FIGURE 10.1
The levels below the logic gate level.



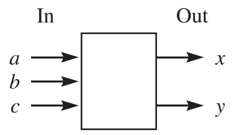
10.1 Boolean Algebra and Logic Gates

A *circuit* is a collection of devices that are physically connected by wires. The two basic types of circuits at Level LG1 are *combinational* and *sequential*. You can visualize either type of circuit as a rectangular block called a *black box* with a fixed number of input lines and a fixed number of output lines. FIGURE 10.2 shows a three-input, two-output circuit.

Each line can carry a signal whose value is either 1 or 0. Electrically, a 1 signal is a small voltage, usually about 3 volts, and a 0 signal is 0 volts. The circuit is designed to detect and produce only those binary values.

You should recognize Figure 10.2 as one more manifestation of the input-processing-output structure that is present at all levels of the computer system. The circuit performs the processing that transforms the input to the output.

FIGURE 10.2
The black box representation of a circuit.



Combinational Circuits

With a *combinational circuit*, the input determines the output. For example, in Figure 10.2 if you put in $a = 1$, $b = 0$, $c = 1$ (abbreviated $abc = 101$) today and get out $xy = 01$, then if you put in $abc = 101$ tomorrow you will get out $xy = 01$ again. Mathematically, x and y are functions of a, b, and c. That is, $x = x(a, b, c)$ and $y = y(a, b, c)$.

This behavior is not characteristic of a sequential circuit. It may be possible for you to put $abc = 101$ into a sequential circuit and get out $xy = 01$ at one moment, but $xy = 11$ a few microseconds later. Chapter 11 shows how this seemingly useless behavior comes about and how it is, in fact, indispensable for building computers.

The three most common methods for describing the behavior of a combinational circuit are

- Truth tables
- Boolean algebraic expressions
- Logic diagrams

The remainder of this section describes those representations.

FIGURE 10.3
The truth table for a three-input, two-output combinational circuit.

a	b	c	x	y
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

FIGURE 10.4
The truth table for a four-input, two-output combinational circuit.

a	b	c	d	x	y
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

Truth Tables

Of these three methods of representing a combinational circuit, truth tables are at a higher level of abstraction than algebraic expressions or logic diagrams. A truth table specifies what the combinational circuit does, not how it does it. A truth table simply lists the output for every possible combination of input values (hence the name *combinational circuit*).

Example 10.1 FIGURE 10.3 is the truth table for a three-input, two-output combinational circuit. Because there are three inputs and each input can have one of two possible values, the table has $2^3 = 8$ entries. In general, the truth table for an n-input combinational circuit will have 2^n entries. ■

Example 10.2 Another example of a combinational circuit specified by a truth table is FIGURE 10.4. It is a four-input circuit with 16 entries in its truth table. ■

The black box schematic of Figure 10.2 is particularly appropriate for the truth table representation of a combinational circuit. You cannot see inside a box that is painted black. Similarly, you cannot see how a circuit produces a function that is defined by a truth table.

Boolean Algebra

An algebraic expression written according to the laws of Boolean algebra specifies not only what a combinational circuit does, but how it does it. Boolean algebra is similar in some respects to the algebra for real numbers that you are familiar with, but it is different in other respects. The four basic operations for real algebra are addition, subtraction, multiplication, and division. Boolean algebra has three basic operations: OR (denoted +), AND (denoted $\cdot$), and complement (denoted $'$). AND and OR are binary operations, and complement is a unary operation.

The 10 fundamental properties of Boolean algebra are

$x + y = y + x$	$x \cdot y = y \cdot x$	commutative
$(x + y) + z = x + (y + z)$	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	associative
$x + (y \cdot z) = (x + y) \cdot (x + z)$	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	distributive
$x + 0 = x$	$x \cdot 1 = x$	identity
$x + (x') = 1$	$x \cdot (x') = 0$	complement

where x, y, and z are Boolean variables. As with real algebra, the notation is infix with parentheses to denote which of several operations to perform first. To simplify expressions with many parentheses, the Boolean operations have the precedence structure shown in FIGURE 10.5. Using the precedence rules, the distributive properties are

$$x + y \cdot z = (x + y) \cdot (x + z) \qquad x \cdot (y + z) = x \cdot y + x \cdot z$$

and the complement properties are

$$x + x' = 1 \qquad x \cdot x' = 0$$

The distributive law

A striking difference between the properties of real algebra and Boolean algebra is the distributive law. With real numbers, multiplication distributes over addition. For example,

$$2 \cdot (3 + 4) = 2 \cdot 3 + 2 \cdot 4$$

But addition does not distribute over multiplication. It is not true that

$$2 + 3 \cdot 4 = (2 + 3) \cdot (2 + 4)$$

In Boolean algebra, however, where + represents OR and · represents AND, OR does distribute over AND.

FIGURE 10.5
Precedence of the Boolean operators.

Precedence	Operator
Highest	Complement
	AND
Lowest	OR

Duality

The laws of Boolean algebra have a symmetry that the laws of real algebra do not have. Each Boolean property has a *dual* property. To obtain the dual expression,

- Exchange + and ·
- Exchange 1 and 0

The two forms of the distributive law are an example of dual expressions. In the distributive property

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

if you exchange the + and · operators, you get

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

which is the other distributive property. Each fundamental property of Boolean algebra has a corresponding dual property.

The associative law

The associative properties also permit simplification of expressions. Because the order in which you perform two OR operations is immaterial, you can write

$$(x + y) + z$$

without parentheses as

$$x + y + z$$

The same is true for the AND operation.

Boolean Algebra Theorems

Because Boolean algebra has a different mathematical structure from the real algebra with which you are familiar, the theorems of Boolean algebra may appear unusual at first. Some of the following theorems proved from the 10 basic properties of Boolean algebra are useful in the analysis and design of combinational circuits.

The *idempotent property* states that

The idempotent property

$$x + x = x$$

Proving this theorem requires a sequence of substitution steps, each of which is based on one of the 10 basic properties of Boolean algebra:

$$\begin{aligned}
& x + x \\
= \quad & \langle \text{identity of AND} \rangle \\
& (x + x) \cdot 1 \\
= \quad & \langle \text{complement of OR} \rangle \\
& (x + x) \cdot (x + x') \\
= \quad & \langle \text{distributive of OR over AND} \rangle \\
& x + (x \cdot x') \\
= \quad & \langle \text{complement of AND} \rangle \\
& x + 0 \\
= \quad & \langle \text{identity of OR} \rangle \\
& x
\end{aligned}$$

The dual property is

$$x \cdot x = x$$

The proof of the dual theorem requires exactly the same sequence of steps, with each substitution based on the dual of the corresponding step in the original proof:

$$\begin{aligned}
& x \cdot x \\
= \quad & \langle \text{identity of OR} \rangle \\
& (x \cdot x) + 0 \\
= \quad & \langle \text{complement of AND} \rangle \\
& (x \cdot x) + (x \cdot x') \\
= \quad & \langle \text{distributive of AND over OR} \rangle \\
& x \cdot (x + x') \\
= \quad & \langle \text{complement of OR} \rangle \\
& x \cdot 1 \\
= \quad & \langle \text{identity of AND} \rangle \\
& x
\end{aligned}$$

The proofs of the idempotent properties illustrate an important application of duality in Boolean algebra. Once you prove a theorem, you can assert immediately that its dual must also be true. Because each of the 10 basic properties has a dual, the corresponding proof will be identical in structure to the original proof, but with each step based on the dual of the original step.

Using duality to assert a theorem

Here are three more useful theorems with their duals. The mathematical rule for proving theorems is that you may use any axiom or previously proved theorem in your proof. So to prove the first theorem below, you may use any of the fundamental properties or the idempotent property. To prove the second theorem, you may use any of the fundamental properties, or the idempotent property, or the first theorem, and so on. The first theorem

The zero theorem

$$x + 1 = 1 \qquad\qquad x \cdot 0 = 0$$

is called the *zero theorem*. 0 is the zero for the AND operator, and 1 is the "zero" for the OR operator. The second theorem

The absorption property

$$x + x \cdot y = x \qquad\qquad x \cdot (x + y) = x$$

is called the *absorption property* because y is absorbed into x. The third theorem

The consensus theorem

$$x \cdot y + x' \cdot z + y \cdot z = x \cdot y + x' \cdot z$$
$$(x + y) \cdot (x' + z) \cdot (y + z) = (x + y) \cdot (x' + z)$$

is called the *consensus theorem*. Proofs of these theorems are exercises at the end of the chapter.

Proving Complements

The complement of x is x'. To prove that some expression, y, is the complement of some other expression, z, you must show that y and z obey the same complement properties,

$$y + z = 1 \qquad\qquad y \cdot z = 0$$

that x and x' obey.

An example of proving complements is *De Morgan's law*, which states that

De Morgan's law

$$(a \cdot b)' = a' + b'$$

To show that the complement of $a \cdot b$ is $a' + b'$, you must show that

$$(a \cdot b) + (a' + b') = 1 \qquad\qquad (a \cdot b) \cdot (a' + b') = 0$$

The first part of the proof is

$$\begin{array}{ll} & (a \cdot b) + (a' + b') \\ = & \quad \langle \text{commutative of OR} \rangle \\ & (a' + b') + a \cdot b \\ = & \quad \langle \text{distributive of OR over AND} \rangle \\ & ((a' + b') + a) \cdot ((a' + b') + b) \\ = & \quad \langle \text{commutative and associative of OR} \rangle \\ & (b' + (a + a')) \cdot (a' + (b + b')) \\ = & \quad \langle \text{complement of OR} \rangle \\ & (b' + 1) \cdot (a' + 1) \\ = & \quad \langle \text{the zero theorem of OR, } x + 1 = 1 \rangle \\ & 1 \cdot 1 \\ = & \quad \langle \text{identity of AND, } (x \cdot 1 = 1)\ [x := 1] \rangle \\ & 1 \end{array}$$

and the second part of the proof is

$$\begin{array}{ll} & (a \cdot b) \cdot (a' + b') \\ = & \quad \langle \text{distributive of AND over OR} \rangle \\ & (a \cdot b) \cdot a' + (a \cdot b) \cdot b' \\ = & \quad \langle \text{commutative and associative of AND} \rangle \\ & b \cdot (a \cdot a') + a \cdot (b \cdot b') \\ = & \quad \langle \text{complement of AND} \rangle \\ & b \cdot 0 + a \cdot 0 \\ = & \quad \langle \text{the zero theorem of AND, } x \cdot 0 = 0 \rangle \\ & 0 + 0 \\ = & \quad \langle \text{identity of OR, } (x + 0 = x)[x := 0] \rangle \\ & 0 \end{array}$$

De Morgan's second law,

$$(a + b)' = a' \cdot b'$$

follows immediately from duality.

De Morgan's laws generalize to more than one variable. For three variables, the laws are

De Morgan's law for three variables

$$(a \cdot b \cdot c)' = a' + b' + c' \qquad (a + b + c)' = a' \cdot b' \cdot c'$$

Proofs of the general theorems for more than two variables are an exercise at the end of the chapter.

The complement of x′

Another complement theorem is $(x')' = x$. The complement of x' is x because $x' + x = 1$ by the following proof:

$$\begin{aligned} & x' + x \\ = \quad & \langle \text{commutative of OR} \rangle \\ & x + x' \\ = \quad & \langle \text{complement of OR} \rangle \\ & 1 \end{aligned}$$

and $x' \cdot x = 0$ by the following proof:

$$\begin{aligned} & x' \cdot x \\ = \quad & \langle \text{commutative of AND} \rangle \\ & x \cdot x' \\ = \quad & \langle \text{complement of AND} \rangle \\ & 0 \end{aligned}$$

Yet another complement theorem is $1' = 0$. 1 is the complement of 0 because $1 + 0 = 1$ by the following proof:

$$\begin{aligned} & 1 + 0 \\ = \quad & \langle \text{identity of OR, } (x + 0 = x)\ [x := 1] \rangle \\ & 1 \end{aligned}$$

and $1 \cdot 0 = 0$ by the following proof:

$$\begin{aligned} & 1 \cdot 0 \\ = \quad & \langle \text{commutative of AND} \rangle \\ & 0 \cdot 1 \\ = \quad & \langle \text{identity of AND, } (x \cdot 1 = x)\ [x := 0] \rangle \\ & 0 \end{aligned}$$

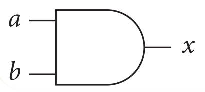
The dual theorem, $0' = 1$, follows immediately.

Logic Diagrams

The third representation of a combinational circuit is an interconnection of logic gates. This representation corresponds most closely to the hardware because the lines that connect the gates in a logic diagram represent physical wires that connect physical devices on a circuit board or in an integrated circuit.

Each Boolean operation is represented by a gate symbol, shown in FIGURE 10.6. The AND and OR gates have two input lines, labeled *a* and *b*. The inverter has one input line, corresponding to the fact that the complement is a unary operation. The output is *x*. Also shown in the figure are the corresponding Boolean expression and truth table for each gate.

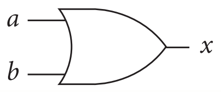
FIGURE 10.6
The three basic logic gates.



$x = a \cdot b$

a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

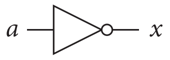
(a) AND gate.



$x = a + b$

a	b	x
0	0	0
0	1	1
1	0	1
1	1	1

(b) OR gate.

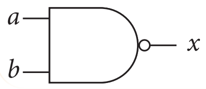


$x = a'$

a	x
0	1
1	0

(c) Inverter.

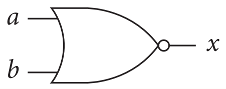
FIGURE 10.7
Three common logic gates.



$x = (a \cdot b)'$

a	b	x
0	0	1
0	1	1
1	0	1
1	1	0

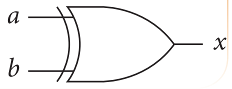
(a) NAND gate.



$x = (a + b)'$

a	b	x
0	0	1
0	1	0
1	0	0
1	1	0

(b) NOR gate.

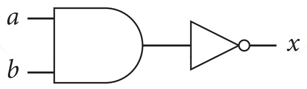


$x = a \oplus b$

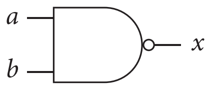
a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

(c) XOR gate.

FIGURE 10.8
Two equivalent combinational circuits.



(a) AND inverter.



(b) NAND.

Any Boolean function can be written with only the AND, OR, and complement operations. It follows that to construct any combinational circuit, you need only the three basic gates of Figure 10.6. In practice, several other gates are common. FIGURE 10.7 shows three of them.

The NAND gate (not AND) is equivalent to an AND gate followed by an inverter, as shown in FIGURE 10.8. Similarly, a NOR gate (not OR) is equivalent to an OR gate followed by an inverter. Electronically, it is frequently easier to build a NAND gate than to build an AND gate. In fact, an AND gate is often built as a NAND gate followed by an inverter. NOR gates are also more common than OR gates.

XOR stands for *exclusive* OR, in contrast to OR, which is sometimes called *inclusive* OR. The output of an OR gate is 1 if either or both of its inputs are 1. The output of an XOR gate is 1 if either of its inputs is 1 exclusive of the other input. Its output is 0 if both inputs are 1. The algebraic symbol for the XOR operation is $\oplus$. The algebraic definition of $a \oplus b$ is

$$a \oplus b = a \cdot b' + a' \cdot b$$

The precedence for the XOR operator is greater than OR but less than AND, as FIGURE 10.9 shows.

FIGURE 10.9 Precedence of the XOR operator.

Precedence	Operator
Highest	Complement
	AND
	XOR
Lowest	OR

Example 10.3 The expression

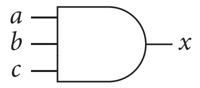
$$a + b \oplus c \cdot d$$

fully parenthesized is $a + (b \oplus (c \cdot d))$. Expanded according to the definition of XOR, the expression becomes

$$a + b \cdot (c \cdot d)' + b' \cdot (c \cdot d)$$

The AND and OR gates are also manufactured with more than two inputs. FIGURE 10.10 shows a three-input AND gate and its truth table. The output of an AND gate is 1 only if all of its inputs are 1. The output of an OR gate is 0 only if all of its inputs are 0.

FIGURE 10.10 The three-input AND gate.



$$x = a \cdot b \cdot c$$

a	b	c	x
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Alternate Representations

You may have recognized the similarity of the truth tables for the AND, OR, and inverter gates and the truth tables for the AND, OR, and NOT operations in C's Boolean expressions. The truth tables are identical, with NOT corresponding to the inverter and C's true and false values corresponding to Boolean algebra's 1 and 0, respectively.

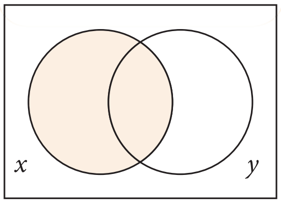
Statement logic interpretation

The mathematical structure of Boolean algebra is important because it applies not only to combinational circuits, but also to statement logic. C uses statement logic to determine the truth of a condition contained in `if` and loop statements. A group of programming languages important in artificial intelligence makes even more extensive use of statement logic. Programs written in these languages simulate human reasoning with a technique called *logic programming*. Boolean algebra is a major component of that discipline.

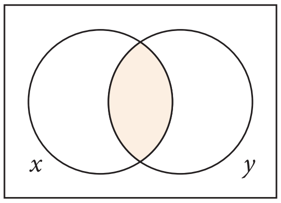
Set theory interpretation

Another interpretation of Boolean algebra is a description of operations on sets. If you interpret a Boolean variable as a set, the OR operation as set union, the AND operation as set intersection, the complement operation as set complement, 0 as the empty set, and 1 as the universal set, then all the properties and theorems of Boolean algebra hold for sets.

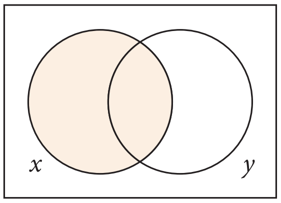
FIGURE 10.11 The set theory interpretation of an absorption property.



(a) x



(b) $x \cdot y$



(c) $x + x \cdot y$

Example 10.4 The theorem

$$x + 1 = 1$$

states that the union of the universal set with any other set is the universal set. ▮

Example 10.5 FIGURE 10.11 shows the set theory interpretation of an absorption property

$$x + x \cdot y = x$$

with a Venn diagram. Figure 10.11(a) shows set x. The intersection of x and y, shown in (b), is the set of elements in both x and y. The union of that set with x is shown in (c). The fact that the region in (a) is the same as the region in (c) illustrates the absorption property. ▮

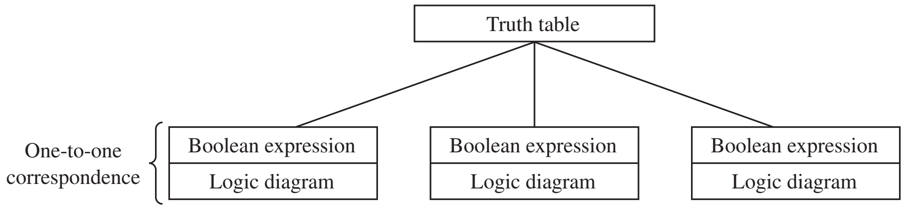
The interpretation of Boolean algebra as a description of combinational circuits and as the basis of statement logic illustrates that it is the mathematical basis of a large part of computer science. The fact that it also describes set theory shows its importance in other areas of mathematics as well.

10.2 Combinational Analysis

Every Boolean expression has a corresponding logic diagram, and every logic diagram has a corresponding Boolean expression. In mathematical terminology, there is a one-to-one correspondence between the two. A given truth table, however, can have several corresponding implementations. FIGURE 10.12 shows a truth table with several corresponding Boolean expressions and logic diagrams.

This section describes the correspondence among the three representations of a combinational circuit.

FIGURE 10.12 Several implementations of a given truth table.



Boolean Expressions and Logic Diagrams

A Boolean expression consists of one or more variables combined with the AND, OR, and invert operations. The number of inputs to the circuit equals the number of variables. This section and the next concentrate on circuits with one output. The last section of this chapter considers circuits with more than one output.

Constructing a logic diagram from a Boolean expression

To draw the logic diagram from a given Boolean expression, draw an AND gate for each AND operation, an OR gate for each OR operation, and an inverter for each complement operation. Connect the output of one gate to the input of another according to the expression. The output of the combinational circuit is the output of the one gate that is not connected to the input of another.

Example 10.6 FIGURE 10.13 shows the logic diagram corresponding to the Boolean expression $a + b' \cdot c$.

From now on, we will omit the AND operator symbol and write the Boolean expression as

$$a + b'c$$

The output of each gate is labeled with its corresponding expression. ■

When the expression has parentheses, you must construct the subdiagram within the parentheses first.

Example 10.7 FIGURE 10.14 is the logic diagram for the three-variable expression

$$((ab + bc')a)'$$

You first form ab with one AND gate, then bc' with another AND gate. The output of those two are ORed and then ANDed with a. Because the entire expression is complemented, an inverter is the last gate. ■

FIGURE 10.13
The logic diagram for the Boolean expression $a + b' \cdot c$.

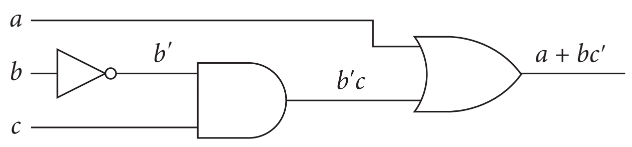


FIGURE 10.14
The logic diagram for the Boolean expression $((ab + bc')a)'$.

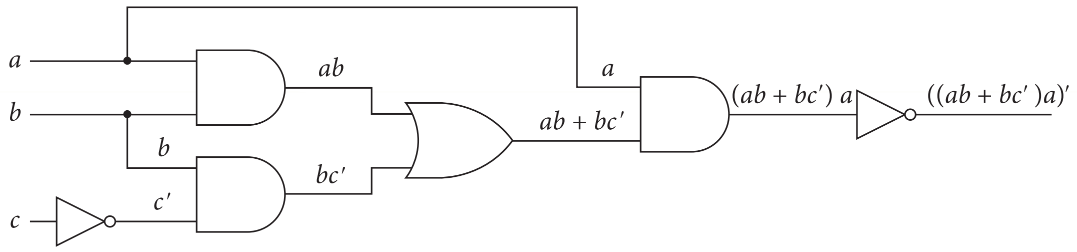


Figure 10.14 shows two junctions as small black dots where two wires are physically connected. Recall that the physical signal supplied by variable a is a voltage. When the signal from input a reaches the junction, it does not act like water in a river that encounters a fork in its path. In the river analogy, some of the water takes one path and some takes the other. In a logic diagram, it does not happen that part of the signal goes to the input of one AND gate and part goes to the other gate. The full signal from a is duplicated at the inputs at both gates.

For those who know some physics, the reason for this behavior is that voltage is a measure of electric potential. The wires have low resistance, which from Ohm's law means there is negligible potential change along a wire. So, the voltage along any wires that are physically connected is constant. The full voltage signal is, therefore, available at any point, regardless of the junction. (For those who do not know some physics, this may be an incentive to learn!)

The signal from any variable can be duplicated with a junction. The complement of any variable can be produced by an inverter, which can, in turn, also be duplicated by a junction. Rather than show variable-duplicating junctions and variable inverters, they are often omitted from the logic diagram. It is assumed that any variable or its complement is available as input to any gate.

Example 10.8 FIGURE 10.15 shows an abbreviated version of Figure 10.14 that takes advantage of this assumption. It also recognizes that an AND gate followed by an inverter is equivalent to a NAND gate.

A disadvantage of this abbreviated diagram is that the three-input nature of the network is not as evident as it is in Figure 10.14.

Example 10.9 FIGURE 10.16 shows the logic diagram for the four-input Boolean expression

$$(a'bc \oplus c + a + d)'$$

FIGURE 10.15
An abbreviated version of Figure 10.14.

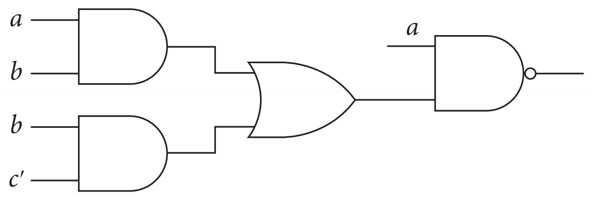
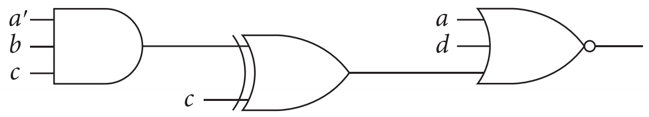


FIGURE 10.16
The logic diagram for the Boolean expression $(a'bc \oplus c + a + d)'$.



Note that the precedence of the exclusive OR operator is less than that of AND and greater than that of OR. ■

Constructing a Boolean expression from a logic diagram

To write the Boolean expression from a given logic diagram, simply label the output of each gate with the appropriate subexpression. If you were given the logic diagram of Figure 10.16 without the Boolean expression, you would start by labeling the output of the AND gate as $a'bc$. The output of the XOR gate would be labeled $a'bc \oplus c$, which when passed through the NOR gate produces the full Boolean expression.

Truth Tables and Boolean Expressions

One method for constructing a Boolean expression from a truth table is to write the expression without parentheses as an OR of several AND terms. Each AND term corresponds to a 1 in the truth table.

Example 10.10 The truth table for $a \oplus b$ has two 1's. The corresponding Boolean expression is

$$a \oplus b = a'b + ab'$$

If a is 0 and b is 1, the first AND term will be 1. If a is 1 and b is 0, the second AND term will be 1. In either case, the OR of the two terms will be 1. Furthermore, any other combination of values for a and b will make both AND terms 0, and the Boolean expression 0. ■

Example 10.11 Figure 10.3 shows x is 1 when $abc = 001$ and $abc = 011$; x is 0 for all other combinations of abc. A corresponding Boolean expression is

$$x = a'b'c + a'bc$$

The first AND term, $a'b'c$, is 1 if and only if $abc = 001$. The second is 1 if and only if $abc = 011$. So the OR of the two terms will be 0 except under either of those conditions, duplicating the truth table. ■

Example 10.12 An example with four variables is the truth table for x in Figure 10.4. A corresponding expression is

$$x = a'bc'd + a'bcd + abc'd + abcd$$

which gives 1 for the four combinations of a, b, c, and d that have 1 in the truth table. ■

The dual technique is to write an expression as the AND of several OR terms. Each OR term corresponds to a 0 in the truth table.

Example 10.13 The expression from FIGURE 10.17 is

$$x = (a + b' + c')(a' + b' + c)$$

If $abc = 011$, the first OR term is 0. If $abc = 110$, the second OR term is 0. Under either of these conditions, the AND of the OR terms is 0. All other combinations of abc will make both OR terms 1 and the expression 1. ■

FIGURE 10.17
A three-variable truth table.

a	b	c	x
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Given a Boolean expression, the most straightforward way to construct the corresponding truth table is to evaluate the expression for all possible combinations of the variables.

Example 10.14 To construct the truth table for

$$x(a, b) = (a \oplus b)' + a'$$

requires the evaluation of

$$x(0, 0) = (0 \oplus 0)' + 0' = 1$$
$$x(0, 1) = (0 \oplus 1)' + 0' = 1$$
$$x(1, 0) = (1 \oplus 0)' + 1' = 0$$
$$x(1, 1) = (1 \oplus 1)' + 1' = 1$$

This example requires the evaluation of all four possible combinations of the two variables a and b. ■

If the expression contains more than two variables, sometimes it is easier to convert the Boolean expression into an OR of AND terms using the properties and theorems of Boolean algebra. The truth table can then be written by inspection.

Example 10.15 The expression in Figure 10.16 reduces to

$$
\begin{aligned}
& (a'bc \oplus c + a + d)' \\
= \quad & \langle \text{definition of } \oplus \rangle \\
& (a'bcc' + (a'bc)'c + a + d)' \\
= \quad & \langle \text{complement, } cc' = 0\text{, and zero theorem } x \cdot 0 = 0 \rangle \\
& ((a'bc)'c + a + d)' \\
= \quad & \langle \text{De Morgan} \rangle \\
& ((a + b' + c')c + a + d)' \\
= \quad & \langle \text{distributive, complement, and identity} \rangle \\
& (ac + b'c + a + d)' \\
= \quad & \langle \text{absorption, } a + ac = a \rangle \\
& (a + b'c + d)' \\
= \quad & \langle \text{De Morgan} \rangle \\
& a'(b'c)'d' \\
= \quad & \langle \text{De Morgan} \rangle \\
& a'(b + c')d' \\
= \quad & \langle \text{distributive} \rangle \\
& a'bd' + a'c'd'
\end{aligned}
$$

The truth table has 16 entries. By inspection, insert a 1 where $abd = 010$ (two places) and $acd = 000$ (two places). All other entries are 0. The result is FIGURE 10.18. It has three 1's instead of four because one of the places where $abd = 010$ is also one of the places where $acd = 000$.

This technique saves you from evaluating the original expression 16 times. Actually, that task may not be as difficult as it first appears. With a little thought, you can reason from the original expression that when d is 1, the expression inside the parentheses must be 1, and its inverse must be 0 regardless of the values of a, b, and c. Similarly, the expression must be 0 when a is 1. That leaves you with only the four evaluations where $ad = 00$. ■

FIGURE 10.18 The truth table for the expression in Figure 10.16.

a	*b*	*c*	*d*	*x*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Two-Level Circuits

The fact that every Boolean expression can be transformed to an AND-OR expression has an important practical effect on the processing speed of the combinational circuit. When you change a signal at the input of a gate, the output does not respond instantly. Instead, there is a time delay during which the signal works its way through the internal electronic components of the gate. The time it takes for the output of a gate to respond to a change in its input is called the *gate delay*. Different manufacturing processes produce gates with different gate delays. To produce gates with short gate delays is

Gate delays

more expensive, and the gates require more power to operate than gates with longer delays. A typical gate delay is 2 ns (nanoseconds), although the delay varies widely depending on the device technology.

Physical limits on processing speed

Two billionths of a second may not seem like a long time to wait for the output, but in a circuit with a long string of gates that must do its processing in a loop, the time can be significant. By way of comparison, consider the fact that the signal travels through the wires at approximately the speed of light, which is 3.0×10^8 m/s (meters per second). That is 30 cm, or about a foot, in 1 ns. In 2 ns, the time of a typical gate delay, the signal can travel through 60 cm of wire. This is such a long distance compared to the size of an integrated circuit or circuit board that the gate delay is, for all practical purposes, responsible for the limit on a network's processing speed.

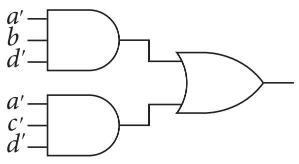
Example 10.16 Consider the circuit of Figure 10.16. If the gate delay is 2 ns, a change in b requires 2 ns to propagate through the AND gate, 2 ns to propagate through the XOR gate, and another 2 ns to propagate through the NOR gate. That is a total of 6 ns of propagation time. (We will ignore the propagation delay through any inverters.)

Now consider that we used Boolean algebra to write the expression for this circuit as an AND-OR expression:

$$\begin{aligned} x &= (a'bc \oplus c + a + d)' \\ &= a'bd' + a'c'd' \end{aligned}$$

FIGURE 10.19 shows the corresponding circuit. It is called a *two-level circuit* because a change in the input requires only two gate delays to propagate to the output. ■

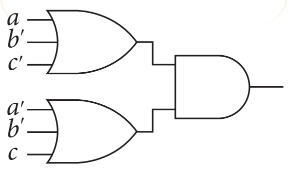
FIGURE 10.19 The two-level AND-OR circuit equivalent to the circuit of Figure 10.16.



Reducing the processing time from 6 ns to 4 ns is a 33% improvement in speed, which is significant. Because any Boolean expression can be transformed to an AND-OR expression, which corresponds to a two-level AND-OR circuit, it follows that any function can be implemented with a combinational circuit with a processing time of two gate delays at most.

The same principle applies to the dual. It is always possible to transform a Boolean expression to an OR-AND expression, which corresponds to a two-level OR-AND circuit. Such a circuit has a processing time of two gate delays at most. To obtain the Boolean expression as an OR-AND expression, you can first obtain the complement as an AND-OR expression, and then use De Morgan's law.

FIGURE 10.20 The two-level OR-AND circuit of Figure 10.17.



Example 10.17 FIGURE 10.20 is the two-level OR-AND circuit of Figure 10.17 for the expression

$$x = (a + b' + c')(a' + b' + c)$$

Recall that each OR term corresponds to a 0 in the truth table. ■

Example 10.18 The expression from Figure 10.13 is

$$x = a + b'c$$

To transform this expression into an OR-AND expression, first write its complement as

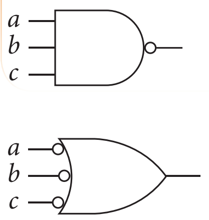
$$\begin{aligned} x' &= (a + b'c)' \\ &= a'(b'c)' \\ &= a'(b + c') \\ &= a'b + a'c' \end{aligned}$$

which is an AND-OR expression. Now use De Morgan's law to write x as

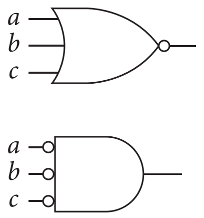
$$\begin{aligned} x &= (x')' \\ &= (a'b + a'c')' \\ &= (a'b)'(a'c')' \\ &= (a + b')(a + c) \end{aligned}$$

which is an OR-AND expression. ■

FIGURE 10.21
Equivalent gates.



(a) A NAND gate as an inverted input OR gate.



(b) A NOR gate as an inverted input AND gate.

It usually happens that a circuit with three or more levels requires fewer gates than the equivalent two-level circuit. Because a gate occupies physical space in an integrated circuit, the two-level circuit achieves its faster processing time at the expense of the extra space required for the additional gates.

This is yet another example of the space/time tradeoff in computer science. It is remarkable that the same space/time principle is manifest from software at the highest level of abstraction to hardware at the lowest level. It is truly a fundamental principle.

The fundamental space/time trade-off

The Ubiquitous NAND

The expression $(abc)'$ represents a three-input NAND gate. De Morgan's law states that

$$(abc)' = a' + b' + c'$$

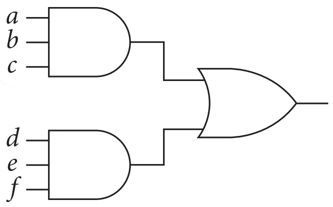
You can visualize the second expression as the output of an OR gate that inverts each input before performing the OR operation. Logic diagrams occasionally render the NAND gate as an inverted input NOR, as in FIGURE 10.21(a).

The dual concept follows from the dual expression

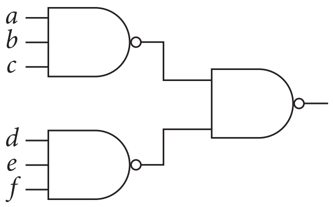
$$(a + b + c)' = a'b'c'$$

A NOR gate is equivalent to an AND gate that inverts its inputs as in Figure 10.21(b).

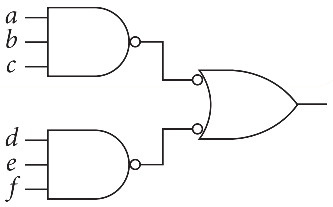
FIGURE 10.22
An AND-OR circuit and its equivalent NAND-NAND circuit.



(a) An AND-OR circuit.



(b) The equivalent NAND-NAND circuit.



(c) The same NAND-NAND circuit as in part (b).

Carrying this idea one step further to two-level circuits, consider the equivalence of

$$abc + def = ((abc)'(def)')'$$

which again follows from De Morgan's law. The first expression represents a two-level AND-OR circuit, whereas the second represents a two-level NAND-NAND circuit. Figure 10.22 shows the equivalent circuits.

FIGURE 10.22(a) shows an AND-OR circuit with two AND gates and one OR. You can make the equivalent circuit entirely out of NAND gates, as shown in (b). Part (c) shows the same circuit as (b) but with the last NAND drawn as an inverted-input OR. This drawing style makes it apparent that the complement following the AND operation cancels the complement preceding the OR operation. The shape of the gate symbols becomes similar to those in the AND-OR circuit, which helps to convey the meaning of the circuit.

Not only can you replace an arbitrary AND-OR circuit entirely with NAND gates, you can also construct an inverter from a NAND gate by connecting the NAND inputs together, as shown in FIGURE 10.23. Because the NAND produces $(ab)'$ with input a and b, if you force $b = a$, the gate will produce $(a \cdot a)' = a'$, the complement of a.

Conceptually, you can construct any combinational circuit from only NAND gates. Furthermore, NAND gates are usually easier to manufacture than either AND or OR gates. Consequently, the NAND gate is by far the most common gate found in integrated circuits.

Of course, the same principle applies to the dual circuit. De Morgan's law for two-level circuits is

$$(a + b + c)(d + e + f) = ((a + b + c)' + (d + e + f)')'$$

which shows that an OR-AND circuit is equivalent to a NOR-NOR circuit. FIGURE 10.24 is the dual circuit of Figure 10.22.

FIGURE 10.23
Three equivalent circuits.

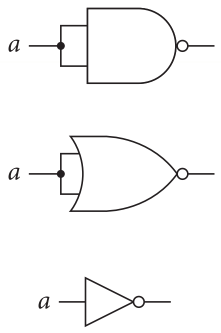
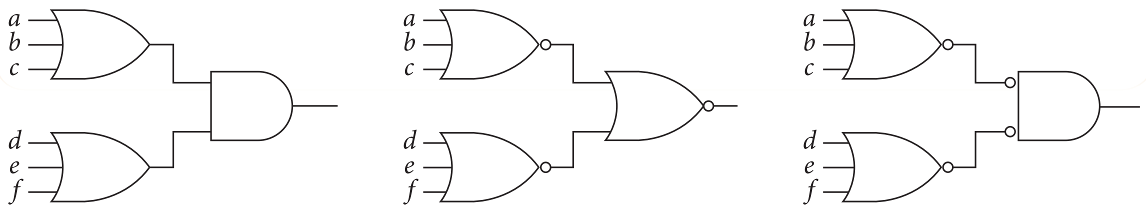


FIGURE 10.24
An OR-AND circuit and its equivalent NOR-NOR circuit.



(a) An OR-AND circuit.

(b) The equivalent NOR-NOR circuit.

(c) The same NOR-NOR circuit as in part (b).

The same reasoning that applies to NAND circuits also applies to NOR circuits. Any combinational circuit can be written as a two-level OR-AND circuit, which can be written as NOR-NOR. Connecting the inputs of a NOR makes an inverter. You can conceptually construct any combinational circuit with only NOR gates.

10.3 Combinational Design

The high speed of two-level circuits gives them an advantage over circuits with more than two levels of gates. Sometimes it is possible to reduce the number of gates in a two-level circuit and retain the processing speed of two gate delays.

Example 10.19 The Boolean expression

$$x(a, b, c, d) = a'bd' + a'c'd' + a'bc'd'$$

can be simplified using the absorption property to

$$\begin{aligned} x(a, b, c, d) &= a'bd' + (a'c'd') + (a'c'd')b \\ &= a'bd' + a'c'd' \end{aligned}$$

This expression also corresponds to a two-level circuit, but it requires only two three-input AND gates and a two-input OR gate, compared to three AND gates (one of which has four inputs) and a three-input OR gate. ■

Minimizing the number of gates in a two-level circuit is not always straightforward with Boolean algebra. This section presents a graphical method for designing two-level circuits with three or four variables that contain the minimum possible number of gates.

Canonical Expressions

Minterms

The previous section shows that any Boolean expression can be transformed to a two-level AND-OR expression. To minimize the two-level circuit, it is desirable to first make each AND term contain all input variables exactly once. Such an AND term is called a *minterm*. It is always possible to transform an AND-OR expression into an OR of minterms.

Example 10.20 Consider the Boolean expression

$$x(a, b, c) = abc + a'bc + ab$$

The first two AND terms are minterms because they contain all three variables, but the last is not. The transformation is

$$\begin{aligned} x &= abc + a'bc + ab \\ &= abc + a'bc + ab(c + c') \\ &= abc + a'bc + abc + abc' \\ &= abc + a'bc + abc' \end{aligned}$$

The definition of a canonical expression

The last expression is called a *canonical expression* because it is an OR of minterms in which no two identical minterms appear. ■

A canonical expression is directly related to the truth table because each minterm in the expression represents a 1 in the truth table. A convenient shorthand notation for a canonical expression and its corresponding truth table is called *sigma notation*, which consists of the uppercase Greek letter sigma (Σ) followed by a list of decimal numbers that specify the rows in the truth table that contain 1's. The uppercase sigma represents the OR operation. It is understood that all the rows not listed contain 0's.

FIGURE 10.25 The truth table for a canonical expression.

Row (dec)	*a*	*b*	*c*	*x*
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Example 10.21 In Example 10.20, because the canonical expression for x has three minterms, its truth table has three 1's. FIGURE 10.25 shows the truth table for this function. It labels each row with the decimal number equivalent of the binary number *abc*. The corresponding sigma notation for this function is

$$x(a, b, c) = \Sigma(3, 6, 7)$$

because rows 3, 6, and 7 contain 1's. ■

The dual canonical expression is an OR-AND expression, each term of which contains all variables once, with no OR terms duplicated. The corresponding notation for this canonical expression contains the list of 0's in the truth table. The uppercase Greek letter pi (Π), which represents the AND operation, is used instead of sigma.

Example 10.22 The dual canonical expression for the previous example is

$$x(a, b, c) = (a + b + c)\,(a + b + c')\,(a + b' + c)\,(a' + b + c)\,(a' + b + c')$$

which is written in pi notation as

$$x(a, b, c) = \Pi(0, 1, 2, 4, 5)$$

because these are the five rows that contain 0's in the truth table. ■

Example 10.23 Using the sigma notation, x and y from Figure 10.3 are

$$x(a, b, c) = \Sigma(1, 3)$$
$$y(a, b, c) = \Sigma(3, 4)$$

Functions x and y from Figure 10.4 are

$$x(a, b, c, d) = \Sigma(5, 7, 13, 15)$$
$$y(a, b, c, d) = \Sigma(4, 5, 12, 13)$$

■

Sigma and pi notation are more compact than the canonical Boolean expressions or the truth tables. The remainder of this section assumes that the function to be minimized has been transformed to its unique canonical expression, or that its truth table has been given or determined.

Three-Variable Karnaugh Maps

Minimization of two-level circuits is based on the concept of distance. The *distance* between two minterms is the number of places in which they differ.

Distance between minterms

Example 10.24 Consider the canonical expression for this function of three variables:

$$x(a, b, c) = a'bc + abc + abc'$$

The distance between minterms $a'bc$ and abc is one, because a' and a are the only variables that differ. Variables b and c are the same in both. The distance between minterms $a'bc$ and abc' is two, because a' and c in $a'bc$ differ from a and c' in abc'. ■

Recognizing *adjacent minterms*—that is, minterms a distance of 1 from each other—is key to the minimization of an AND-OR expression. Once you identify two adjacent minterms, you can factor out the common terms with the distributive property and simplify with the complement and identity properties.

Adjacent minterms

Example 10.25 You can minimize the expression in Example 10.24 by combining the first two minterms as follows:

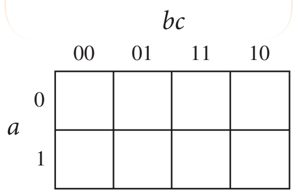
$$\begin{aligned} x(a, b, c) &= a'bc + abc + abc' \\ &= (a' + a)bc + abc' \\ &= bc + abc' \end{aligned}$$

Alternatively, you can minimize by combining the second and third minterms, as they are also adjacent.

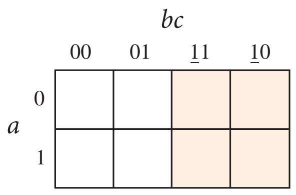
$$\begin{aligned} x(a, b, c) &= a'bc + abc + abc' \\ &= a'bc + ab(c + c') \\ &= a'bc + ab \end{aligned}$$

Either way, you have improved the circuit. The original expression is for a circuit with three three-input AND gates and one three-input OR gate. Either of the simplified expressions is for a circuit with only two AND gates, one of which has only two inputs, and an OR gate with only two inputs.

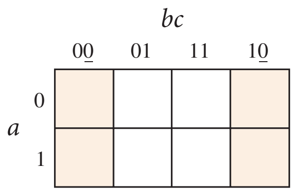
FIGURE 10.26 The Karnaugh map for a function of three variables.



(a) The Karnaugh map.



(b) The $b = 1$ region.



(c) The $c = 0$ region.

Recognizing adjacent minterms is the easy part. Sometimes it is helpful to make the expression temporarily more complicated to get a smaller final circuit. That happens when one minterm is adjacent to two other minterms. You can use the idempotent property to duplicate the minterm, then combine it with both of its adjacent minterms.

Example 10.26 In Example 10.25, you can duplicate *abc* with the idempotent property first, then combine it with both of the remaining minterms.

$$\begin{aligned} x(a, b, c) &= a'bc + abc + abc' \\ &= a'bc + abc + abc + abc' \\ &= (a' + a)bc + ab(c + c') \\ &= bc + ab \end{aligned}$$

This is better than the result in Example 10.25, because both AND gates require only two inputs.

Karnaugh maps

Performing the minimization with Boolean algebra is tedious and error-prone. The Karnaugh map is a tool to minimize a two-level circuit that makes it easy to spot adjacent minterms and to determine which ones need to be duplicated with the idempotent property. A *Karnaugh map* is simply a truth table arranged so that adjacent entries represent minterms that differ by 1.

FIGURE 10.26(a) shows the Karnaugh map for three variables. The upper left cell is for $abc = 000$. To its right is the cell for $abc = 001$. To the right of that is the cell for $abc = 011$, and then $abc = 010$. The sequence

000, 001, 011, 010

guarantees that adjacent cells differ by 1. That would not be the case if the cells were in numeric order

000, 001, 010, 011

because 001 is a distance 2 from 010.

The top row contains entries in the truth table where $a = 0$, and the bottom row contains entries where $a = 1$. Each column gives the values for bc. For example, the first column is for $bc = 00$ and the second for $bc = 01$. The two leftmost columns are for $b = 0$, and the two rightmost columns, Figure 10.26(b), are for $b = 1$. The two outside columns, Figure 10.26(c), are for $c = 0$, and the two middle columns are for $c = 1$.

The Karnaugh map equivalent of the distributive property

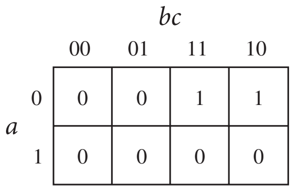
Factoring out a common term from adjacent minterms with Boolean algebra corresponds to grouping adjacent cells on a Karnaugh map. After you group the cells, you write the simplified term by inspection of the region on the Karnaugh map.

Example 10.27 FIGURE 10.27(a) shows the Karnaugh map for the canonical expression

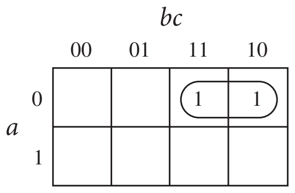
$$x(a, b, c) = a'bc + a'bc'$$

The 1 in the cell for $abc = 011$ is the truth table cell for the minterm $a'bc$. The 1 in the cell for $abc = 010$ is the truth table cell for the minterm $a'bc'$. Figure 10.27(b) is the same Karnaugh map with the zeros omitted for clarity. Because the two ones are adjacent, you can group them with an oval. The cells covered by the oval are in the row for $a = 0$ and the columns for $b = 1$. Therefore, they are the regions for $ab = 01$, which corresponds to the term $a'b$. So, $x(a, b, c) = a'b$. You can write down the result by inspecting the Karnaugh map without doing the Boolean algebra. ■

FIGURE 10.27 The Karnaugh map for the AND-OR expression of Example 10.27.



(a) The Karnaugh map.



(b) The minimization.

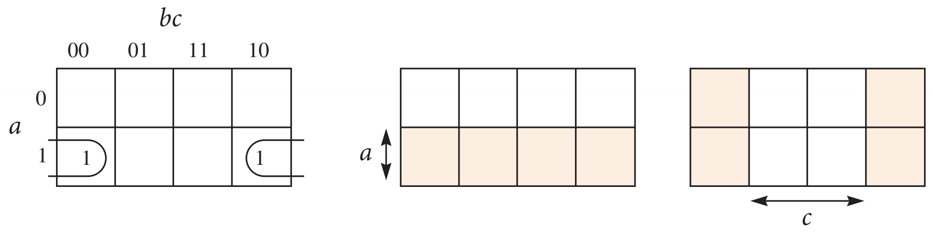
Example 10.28 FIGURE 10.28(a) shows the Karnaugh map for the canonical expression

$$x(a, b, c) = ab'c' + abc'$$

It may appear that the $ab'c'$ cell in the lower left and the abc' cell in the lower right are not adjacent, but in fact they are. You should think of the Karnaugh map as wrapping around so that its left and right sides are adjacent, the so-called *Pac-Man effect*. The single oval in the figure is drawn as two open-ended half ovals to convey this property of the Karnaugh map.

The group of two cells lies in the $a = 1$ row and the $c = 0$ columns, as parts (b) and (c) of the figure show. You can imagine the two cells as the

FIGURE 10.28
The Karnaugh map for the AND-OR expression of Example 10.28.



(a) The Karnaugh map. **(b)** Region a. **(c)** Region c'.

intersection of the shaded regions in (b) and (c). The region for the group is $ac = 10$. Therefore, the minimized function is $x(a, b, c) = ac'$. ▮

The Karnaugh map equivalent of the idempotent property

Duplicating a minterm with the idempotent property, so that it may be combined with two other minterms, corresponds to an overlap of two ovals in the Karnaugh map. If there are more than two minterms in the AND-OR expression, you are free to use a 1 in the truth table for more than one group.

Example 10.29 FIGURE 10.29 shows the Karnaugh map for

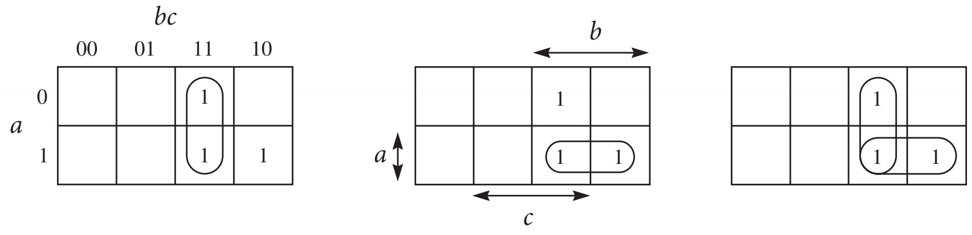
$$x(a, b, c) = a'bc + abc + abc'$$

which is the canonical expression for Example 10.26. Part (a) shows minimization of the first and second minterms. Part (b) shows minimization of the second and third minterms. Part (c) shows that using the second term in both minimizations corresponds to an overlap of the two ovals. ▮

When the original truth table is given in sigma notation, you can use the decimal labels of FIGURE 10.30 to insert 1's in the Karnaugh map.

The minimization procedure requires you to determine the best set of ovals that will cover all the 1's in the Karnaugh map. "Best" means the set

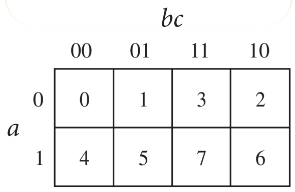
FIGURE 10.29
The Karnaugh map for the AND-OR expression of Example 10.26.



(a) $a'bc + abc = bc$ **(b)** $abc + abc' = ab$ **(c)** $x = bc + ab$

that corresponds to a two-level circuit with the least number of gates and the least number of inputs per gate. The number of ovals equals the number of AND gates. The more 1's an oval covers, the smaller the number of inputs to the corresponding AND gate. It follows that you want the smallest number of ovals, with each oval as large as possible such that the ovals cover all the 1's and no 0's. It is permissible for a 1 to be covered by several ovals. The next few examples show the general strategy.

FIGURE 10.30 Decimal labels for the minterms in the Karnaugh map.



Example 10.30 FIGURE 10.31 shows a common minimization mistake. To minimize

$$x(a, b, c) = \Sigma(0, 1, 5, 7)$$

you may be tempted to first group minterms 1 and 5 as in Figure 10.31(a). That is a bad first choice because minterm 1 is adjacent to both 0 and 5, and minterm 5 is adjacent to both 1 and 7. On the other hand, minterm 0 is adjacent only to 1. To cover 0 with the largest possible oval, you must group it with 1. Similarly, minterm 7 is adjacent only to 5. To cover 7 with the largest possible oval, you must group it with 5.

Figure 10.31(b) shows the result of these minterm groupings. It represents the expression

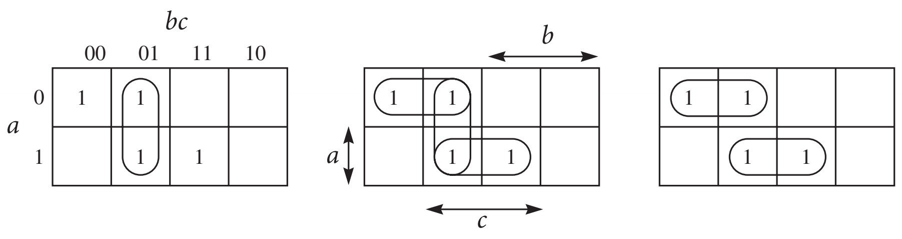
$$\begin{aligned} x(a, b, c) &= \Sigma(0, 1, 5, 7) \\ &= a'b' + b'c + ac \end{aligned}$$

which requires three two-input AND gates and a three-input OR gate. But the grouping of the first choice is not necessary. Figure 10.31(c) shows the correct minimization, which represents

$$\begin{aligned} x(a, b, c) &= \Sigma(0, 1, 5, 7) \\ &= a'b' + ac \end{aligned}$$

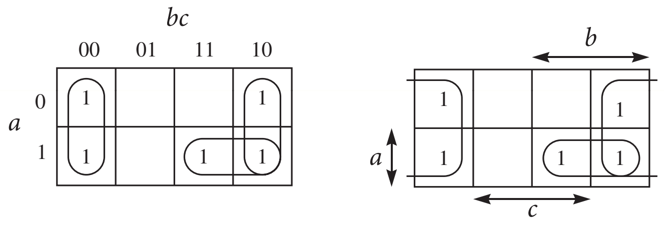
This implementation requires only two two-input AND gates and a two-input OR gate. ▮

FIGURE 10.31 The result of a bad first choice.



(a) A bad strategy. **(b)** The result of the bad strategy. **(c)** The correct minimization.

FIGURE 10.32
Failing to recognize a large grouping.



(a) An incorrect minimization. **(b)** The correct minimization.

The rule of thumb that the previous example teaches us is to start a grouping with minterms that have only one nearest neighbor. Because their neighbors must be grouped with them in any event, you may be spared an unnecessary grouping of their neighbors.

Another common mistake is failing to recognize a large grouping of 1's, as Example 10.31 illustrates.

Example 10.31 FIGURE 10.32(a) shows the minimization of a three-variable function as

$$\begin{aligned} x(a, b, c) &= \Sigma(0, 2, 4, 6, 7) \\ &= b'c' + bc' + ab \end{aligned}$$

which requires three two-input AND gates and one three-input OR gate. Figure 10.32(b) shows the correct minimization as

$$x(a, b, c) = c' + ab$$

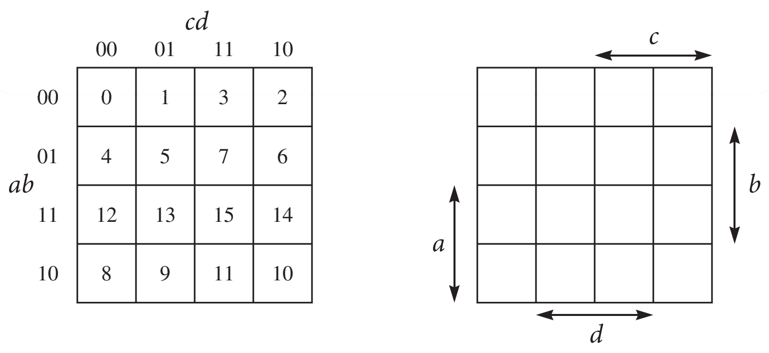
which requires only one two-input AND gate and a two-input OR gate. ■

In a three-variable problem, a grouping of four 1's corresponds to an AND term of only one variable. Because the number of 1's in a group must correspond to an intersection of regions for a, b, and c and their complements, the number of 1's in a group must be a power of 2. For example, an oval can cover one, two, or four 1's but never three or five.

Four-Variable Karnaugh Maps

Minimization of a four-variable circuit follows the same procedure as a three-variable circuit, except that the Karnaugh map has twice as many entries. FIGURE 10.33(a) shows the arrangement of cells. Not only is minterm 0 adjacent to 2, and 4 adjacent to 6, but minterm 12 is adjacent to 14, and 8 to

FIGURE 10.33
The Karnaugh map for a function of four variables.



(a) Decimal labels for the minterms in the Karnaugh map.

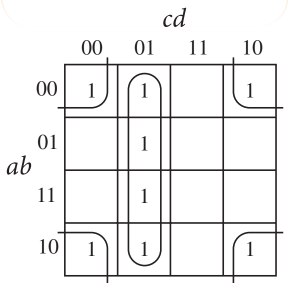
(b) The regions where the variables are 1.

10. Also, cells on the top row are adjacent to the corresponding cells on the bottom row. Minterm 0 is adjacent to 8, 1 to 9, 3 to 11, and 2 to 10.

Each cell in a three-variable Karnaugh map has three adjacent cells. In a four-variable map, each cell has four adjacent cells. For example, the cells adjacent to minterm 10 are 2, 8, 11, and 14. Those adjacent to 4 are 0, 5, 6, and 12.

Figure 10.33(b) shows the regions of the truth table where the variables are 1. Variable *a* is 1 in the two bottom rows, and *b* is 1 in the two middle rows. Variable *c* is 1 in the two right columns, and *d* is 1 in the two middle columns.

FIGURE 10.34
Minimizing a function of four variables.

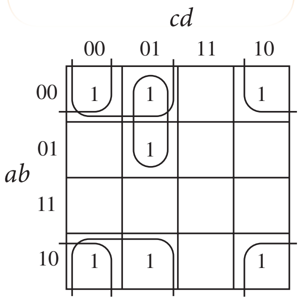


Example 10.32 FIGURE 10.34 shows the minimization

$$x(a, b, c, d) = \Sigma(0, 1, 2, 5, 8, 9, 10, 13)$$
$$= c'd + b'd'$$

Note that the four corner cells can be grouped as $b'd'$. The second column of the Karnaugh map represents $c'd$. ▮

FIGURE 10.35
The expression of Figure 10.34 with one minterm fewer.



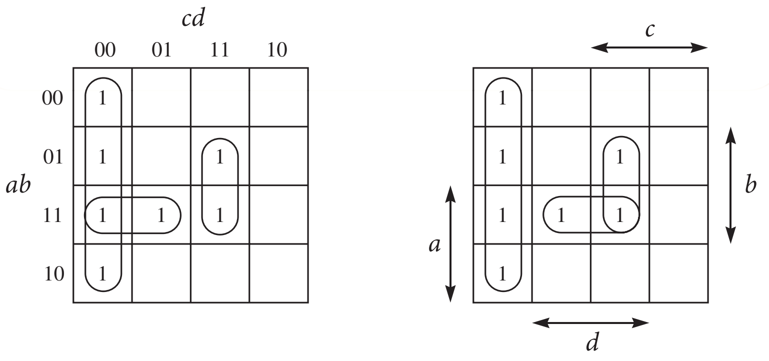
Example 10.33 FIGURE 10.35 shows the minimization

$$x(a, b, c, d) = \Sigma(0, 1, 2, 5, 8, 9, 10)$$
$$= a'c'd + b'c' + b'd'$$

Even though it differs from Example 10.32 by the omission of a single term, the minimization is much different.

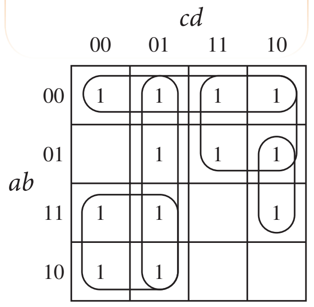
Minterm 5 has only one adjacent 1, so it is grouped first by our rule of thumb with minterm 1. The AND term for this group is $a'c'd$, which you

FIGURE 10.36
Two different correct minimizations.

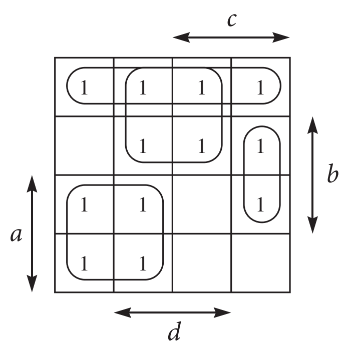


(a) One possible minimization. **(b)** A different minimization.

FIGURE 10.37
A complicated minimization problem.



(a) A plausible but incorrect minimization.



(b) A correct minimization.

can determine by visualizing the intersection of the top two rows (a'), the left two columns (c'), and the middle two columns (d).

Covering minterm 9 with the largest oval requires you to group it with minterms 0, 1, and 8, not just 8. The AND term for this group is $b'c'$, which you can determine by visualizing the intersection of the top and bottom rows (b') with the left two columns (c').

The remaining uncovered 1's are minterms 2 and 10, which are grouped with 0 and 8 as before. ■

Example 10.34 FIGURE 10.36 shows that the minimization may not be unique. Two valid minimizations of this function are

$$\begin{aligned} x(a, b, c, d) &= \Sigma(0, 4, 7, 8, 12, 13, 15) \\ &= c'd' + bcd + abc' \\ &= c'd' + bcd + abd \end{aligned}$$

The first 1 you should group is minterm 7, because it has only one adjacent 1. Minterm 0 must be grouped with 4, 8, and 12, because there is no other possible group for it. That leaves minterm 13, which can be grouped with either 12 or 15. ■

Minimizing a four-variable function is not always straightforward. Sometimes you must simply experiment with several groupings in order to determine the true minimum.

Example 10.35 FIGURE 10.37 shows such a problem. The function is

$$\Sigma(0, 1, 2, 3, 5, 6, 7, 8, 9, 12, 13, 14)$$

Figure 10.37(a) is the result of the following reasoning. Consider minterm 12. The largest group it belongs to is the group of four corresponding to ac'. Similarly, the largest group minterm 6 belongs to is the group of four, $a'c$. Given these two groupings, you can group minterm 5 in $c'd$, minterm 0 in $a'b'$, and minterm 14 in bcd'. The expression

$$ac' + a'c + c'd + a'b' + bcd'$$

is plausible because none of the groupings looks redundant. You cannot remove any oval without uncovering a 1.

Given the selection of the first two groups, the remaining three groups are the best choices possible. The problem is in the selection of the second group.

Figure 10.37(b) is the result of the following reasoning. Group minterm 12 with ac' as before. Now consider minterm 14. You must group it with either 12 or 6. Because 12 is covered, group 14 with 6. Group the remaining minterms—0, 1, 2, 3, 5, 7—most efficiently, as in Figure 10.37(b). The resulting expression,

$$ac' + a'd + a'b' + bcd'$$

requires one fewer AND gate than Figure 10.37(a).

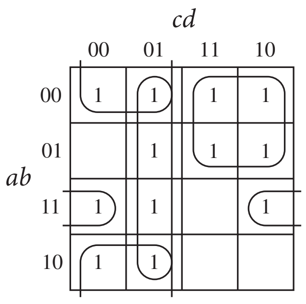
This is a tricky problem because in general you should cover a 1 with the largest possible group. That general rule does not apply in this problem, however. Once you determine the group ac', you should not place minterm 6 in the largest possible group.

FIGURE 10.38 shows that this solution is not unique. It begins by grouping minterm 6 in $a'c$, then minterm 14 with 12. The result is

$$a'c + b'c' + c'd + abd'$$ ■

How do you know which minterms and groupings to consider when confronted with a complicated Karnaugh map? It simply takes practice, reasoning, and a little experimentation.

FIGURE 10.38 Another correct minimization of the function of Figure 10.37.



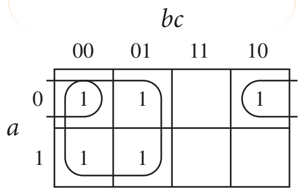
Dual Karnaugh Maps

To minimize a function in an OR-AND expression, minimize the complement of the function in the AND-OR expression, and use De Morgan's law.

Example 10.36 FIGURE 10.39 shows the minimization of the complement of the function in Figure 10.29. The original function is

$$x(a, b, c) = \Sigma(3, 6, 7)$$
$$= \Pi(0, 1, 2, 4, 5)$$

FIGURE 10.39 The complement of the function in Figure 10.29.



Its complement, minimized as shown in the figure, is

$$\begin{aligned} x'(a, b, c) &= \Sigma(0, 1, 2, 4, 5) \\ &= b' + a'c' \end{aligned}$$

The original function in the minimized OR-AND expression is

$$\begin{aligned} x(a, b, c) &= (x'(a, b, c))' \\ &= (b' + a'c')' \\ &= b(a + c) \end{aligned}$$

which requires only two gates, compared to three with the minimized AND-OR expression,

$$x(a, b, c) = bc + ab$$

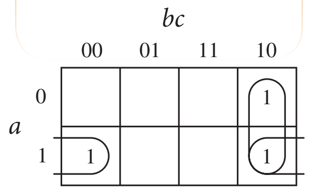
In the previous example, it pays to implement the function with a two-level NOR-NOR circuit instead of a NAND-NAND circuit. In general, you must minimize both forms to determine which requires fewer gates.

Don't-Care Conditions

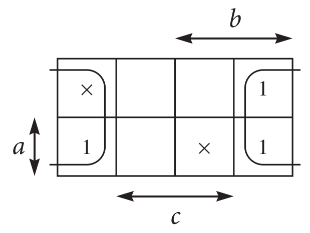
Sometimes a combinational circuit is designed to process only some of the input combinations. The other combinations are not ever expected to be present in the input. These combinations are called *don't-care conditions* because you do not care what the output is if those conditions would ever appear.

Don't-care conditions give you extra flexibility in the minimization process. You can arbitrarily design the circuit to produce either 0 or 1 when a don't-care condition is present. By selectively choosing some don't-care conditions to produce 1 and others to produce 0, you can improve the minimization.

FIGURE 10.40 Don't-care conditions.



(a) Minimizing a function without don't-care conditions.



(b) Minimizing the same function with don't-care conditions.

Example 10.37 FIGURE 10.40(a) shows minimization of

$$\begin{aligned} x(a, b, c) &= \Sigma(2, 4, 6) \\ &= bc' + ac' \end{aligned}$$

without don't-care conditions. Now suppose that instead of requiring minterms 0 and 7 to produce 0, the problem specifies that those minterms can produce either 0 or 1. The notation for this specification is

$$x(a, b, c) = \Sigma(2, 4, 6) + d(0, 7)$$

where d preceding the minterm labels stands for a don't-care condition. Figure 10.40(b) shows × in the Karnaugh map cells for don't-care conditions.

When you minimize with don't-care conditions, you are free to cover or not cover a cell with an ×. An × acts like a wildcard in that you can treat it as a 0 or a 1 as you like. In this problem, if you treat minterm 0 as a 1, and 7 as a 0, the minimization is

$$\begin{aligned} x(a, b, c) &= \Sigma(2, 4, 6) + \mathrm{d}(0, 7) \\ &= \Sigma(0, 2, 4, 6) \\ &= c' \end{aligned}$$

The function without don't-care conditions requires two AND gates and one OR gate, whereas this function requires no AND or OR gates. ▪

10.4 Combinational Devices

This section describes some combinational devices that are commonly used in computer design. Each device can be specified as a black box with a corresponding truth table to define how the outputs depend on the inputs. Because all devices in this section are combinational, they can be implemented with two-level AND-OR circuits. Some implementations shown here trade off processing time for less space—that is, fewer gates—and have more than two levels.

Viewpoints

Several of the following devices have an input line called *enable*. The enable line acts like the on/off switch of an appliance. If the enable line is 0, the output lines are all 0's regardless of the values of the input lines. The device is turned off, or disabled. If the enable line is 1, the output lines depend on the input lines according to the function that specifies the device. The device is turned on, or enabled.

An AND gate can implement the enable property as shown in FIGURE 10.41(a). Suppose line a is one of the outputs from a combinational circuit (not shown in the figure) and the circuit needs an enable line that acts like a switch to turn it on or off. You can feed line a into the AND gate and use the other input to the AND gate as the enable line.

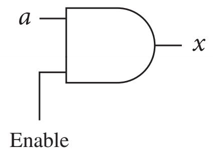
When the enable line is 1,

$$\begin{aligned} x &= a \cdot (\text{enable}) \\ &= a \cdot 1 \\ &= a \end{aligned}$$

and the output equals the input, as in Figure 10.41(b). When the enable line is 0,

$$\begin{aligned} x &= a \cdot (\text{enable}) \\ &= a \cdot 0 \\ &= 0 \end{aligned}$$

FIGURE 10.41 AND input as an enable.



(a) Logic diagram of enable gate.

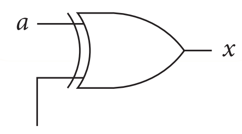
Enable = 1	
a	*x*
0	0
1	1

(b) Truth table with the device turned on.

Enable = 0	
a	*x*
0	0
1	0

(c) Truth table with the device turned off.

FIGURE 10.42
XOR input as a selective invert.



(a) Logic diagram of the selective inverter.

Invert = 1	
a	*x*
0	1
1	0

(b) Truth table with the inverter turned on.

Invert = 0	
a	*x*
0	0
1	1

(c) Truth table with the inverter turned off.

regardless of the input as in Figure 10.41(c).

Implementing the enable property does not require a new "enable gate." It requires only that you adopt a different viewpoint of the familiar AND gate. You can think of input *a* as a data line and enable as a control line. The enable controls the data by either letting it pass through the gate unchanged or preventing it from passing.

Another useful gate is the *selective inverter*. For input, it has a data line and an invert line. If the invert line is 1, the output is the complement of the data line. If the invert line is 0, the data passes through to the output unchanged.

FIGURE 10.42(a) shows that the selective inverter is an XOR gate considered with a different viewpoint than it was previously. When the invert line is 1,

$$\begin{aligned} x &= a \oplus (\text{invert}) \\ &= a' \cdot (\text{invert}) + a \cdot (\text{invert})' \\ &= a' \cdot 1 + a \cdot 1' \\ &= a' \end{aligned}$$

and the output equals the complement of the data input, as in Figure 10.42(b). When the invert line is 0,

$$\begin{aligned} x &= a \oplus (\text{invert}) \\ &= a' \cdot (\text{invert}) + a \cdot (\text{invert})' \\ &= a' \cdot 0 + a \cdot 0' \\ &= a \end{aligned}$$

and the data passes through the gate unchanged.

Multiplexer

A *multiplexer* is a device that selects one of several data inputs to be routed to a single data output. Control lines determine the particular data input to be passed through.

FIGURE 10.43(a) shows the block diagram of an eight-input multiplexer. D0 to D7 are the data input lines, and S2, S1, S0 are the select control lines. F is the single data output line.

Because this device has 11 inputs, a complete truth table would require $2^{11} = 2048$ entries. Figure 10.43(b) shows an abbreviated truth table. The second entry shows that the output is D1 when the select lines are 001. That is, if D1 is 1, F is 1, and if D1 is 0, F is 0, regardless of the other values of D0 and D2 through D7.

Because *n* select lines can select one of 2^n data lines, the number of data inputs of a multiplexer is a power of 2. FIGURE 10.44 shows the implementation of a four-input multiplexer, which contains four data lines, D0 through D3, and two select lines, S1 and S0.

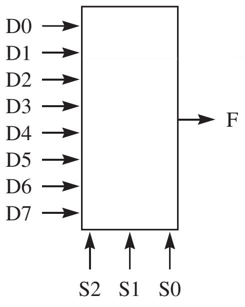
An example of where a multiplexer might be used is in the implementation of the `STWr` instruction in Pep/9. This instruction puts the contents of one of two registers from the CPU into memory via the bus. The CPU could do that with a two-input multiplexer that would make the selection. The select line would come from the register-r field, the inputs would come from the A and X registers, and the output would go to the bus.

Binary Decoder

A *decoder* is a device that takes a binary number as input and sets one of several data output lines to 1 and the rest to 0. The data line that is set to 1 depends on the value of the binary number that is input.

FIGURE 10.45(a) shows the block diagram of a 2 × 4 binary decoder. S1 S0 is the two-bit binary number input and D0 through D3 are the four outputs, one of which will be 1. Part (b) is the truth table.

FIGURE 10.43
The eight-input multiplexer.



(a) Block diagram.

S2	S1	S0	F
0	0	0	D0
0	0	1	D1
0	1	0	D2
0	1	1	D3
1	0	0	D4
1	0	1	D5
1	1	0	D6
1	1	1	D7

(b) Truth table.

FIGURE 10.44
Implementation of a four-input multiplexer.

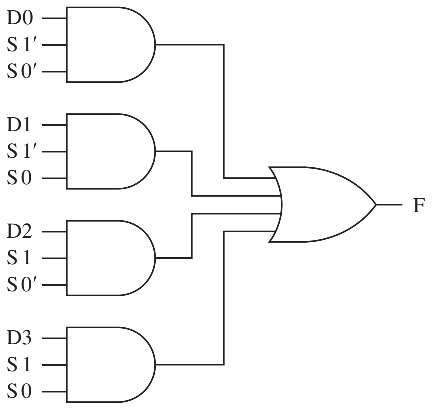
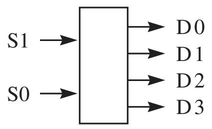


FIGURE 10.45
The 2 × 4 binary decoder.



(a) Block diagram.

S1	S0	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(b) Truth table.

FIGURE 10.46 Implementation of a 2 × 4 binary decoder.

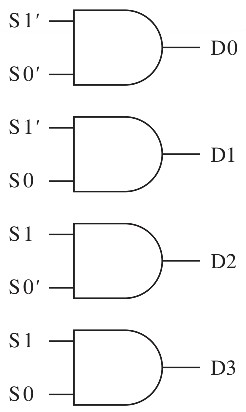
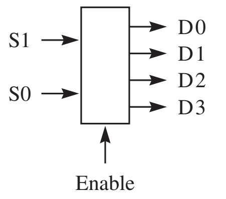


FIGURE 10.47 A 2 × 4 binary decoder with enable.



Because an n-bit number can have 2^n values, the number of data outputs of a decoder is a power of 2. FIGURE 10.46 shows the implementation of a 2 × 4 decoder. Some other possible sizes are 3 × 8 and 4 × 16.

Some decoders are designed with an enable input. FIGURE 10.47 is a block diagram of a 2 × 4 decoder with enable. When the enable line is 1, the device operates normally, as in Figure 10.45(b). When the enable line is 0, all the outputs are 0. To implement a decoder with enable requires an extra input for each AND gate. The details are an exercise at the end of the chapter.

An example of where a decoder might be used is in the CPU of Pep/9. Some instructions have a three-bit addressing-aaa field that specifies one of eight addressing modes. The hardware would have eight address computation units, one for each mode, and each unit would have an enable line. The three aaa address lines would feed into a 3 × 8 decoder. Each output line from the decoder would enable one of the address computation units.

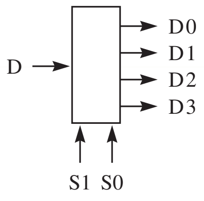
Demultiplexer

A multiplexer routes one of several data input values to a single output line. A *demultiplexer* does just the opposite. It routes a single input value to one of several output lines.

FIGURE 10.48(a) is the block diagram of a four-output demultiplexer. Part (b) is the truth table. If S1 S0 is 01, all the output lines are 0 except D1, which has the same value as the data input line.

This truth table is similar to Figure 10.45(b), the truth table for a decoder. In fact, a demultiplexer is nothing more than a decoder with enable. The data input line, D, is connected to the enable. If D is 0, the decoder is disabled, and the data output line selected by S1 S0 is 0. If D is 1, the decoder is enabled, and the data output line selected is 1. In either case, the selected output line has the same value as the data input line. This is another example of considering a combinational device from a different viewpoint to obtain a useful operation.

FIGURE 10.48 The four-output demultiplexer.

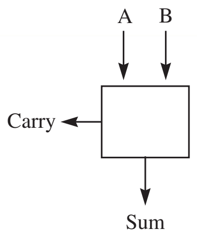


(a) Block diagram.

S1	S0	D0	D1	D2	D3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

(b) Truth table.

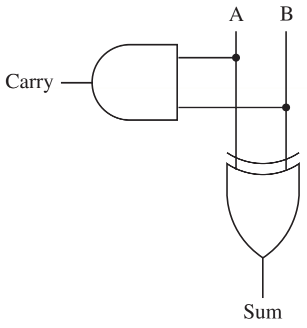
FIGURE 10.49
The half adder.



(a) Block diagram.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(b) Truth table.



(c) Implementation.

Adder

Consider the binary addition

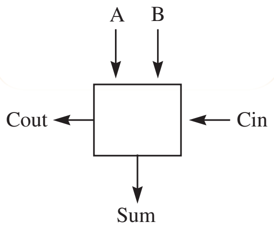
```
        1011
ADD     0011
C = 0   1110
V = 0
```

The sum of the least significant bits (LSBs) is 1 plus 1, which is 0 with a carry of 1 to the next column. To add the LSBs of two numbers requires the *half adder* of FIGURE 10.49(a). In the figure, A represents the LSB of the first number, and B the LSB of the second number. One output is Sum, 0 in this example, and the other is Carry, 1 in this example. Part (b) shows the truth table. The sum is identical to the XOR function, and the carry is identical to the AND function. Part (c) is a straightforward implementation.

To find the sum in the column next to the LSB requires a combinational circuit with three inputs: Cin, A, and B. Cin is the carry input, which comes from the carry of the LSB, and A and B are the bits from the first and second numbers. The outputs are Sum and Cout, the carry output that goes to Cin of the full adder for the next column. FIGURE 10.50(a) is the block diagram of the network, called a *full adder*. Figure 10.50(b) is the truth table. If the sum of the three inputs is odd, Sum is 1. If the sum of the three inputs is greater than 1, Cout is 1.

FIGURE 10.51 shows an implementation of the full adder that uses two half adders and an OR gate. The first half adder adds A and B. The second half adder adds the sum from the first half adder to Cin. The full adder sum

FIGURE 10.50
The full adder.

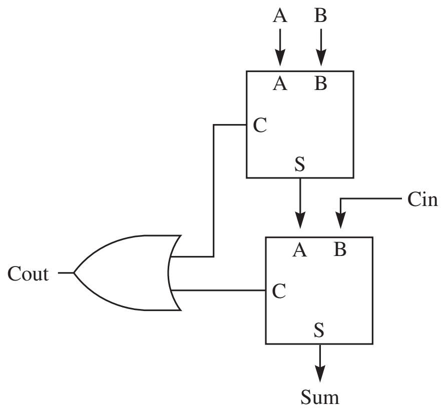


(a) Block diagram.

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(b) Truth table.

FIGURE 10.51
An implementation of the full adder with two half adders.



is the sum from the second half adder. If either the first or second half adder has a carry, the full adder has a carry out.

To add the two four-bit numbers requires an eight-input circuit, shown in FIGURE 10.52(a). A3 A2 A1 A0 are the four bits of the first number, with A0 the LSB. B3 through B0 are the same for the second number. S3 S2 S1 S0 is the four-bit sum, with Cout the carry bit. An implementation of the four-bit adder can use one half adder for the LSB and three full adders, one for each of the remaining columns in the addition. That implementation is called a *ripple-carry adder* because a carry that originates from the LSB must propagate, or ripple through, the columns to the left. Figure 10.52(b) shows the implementation.

The carry out of the ripple-carry adder is the Cout of its leftmost full adder. The carry bit indicates an overflow condition when you interpret the integer as unsigned. When you interpret the integer as signed using two's complement representation, the leftmost bit is the sign bit, and the bit next to it is the most significant bit of the magnitude. So with signed integers, the Cout signal of the penultimate full adder, S2 in this example, acts like the carry out.

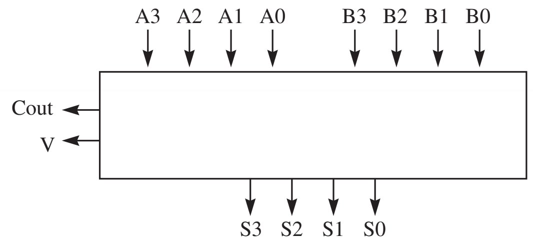
The V bit indicates whether an overflow occurs when the numbers are interpreted as signed. You can get an overflow in only one of two cases:

The two cases for an overflow with signed integers

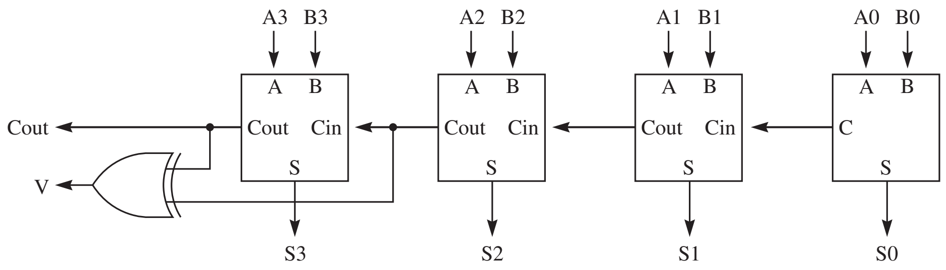
- A and B are both positive, and the result is negative.
- A and B are both negative, and the result is positive.

You cannot get an overflow by adding two integers with different signs. In the first case, A3 and B3 are both 0; there must be a carry from the penultimate full adder that makes S3 1, and Cout from the leftmost full adder is 0. In the second case, A3 and B3 are both 1, so there must be a

FIGURE 10.52
The four-bit ripple-carry adder.



(a) Block diagram.



(b) Implementation.

carry out from the leftmost full adder, and there cannot be a carry out from the penultimate full adder, because S3 must be 0. In both of these cases, the carry out of the leftmost full adder is different from the carry out of the penultimate full adder. But that is precisely the XOR function. It is 1 if and only if its two inputs are different. So, the V bit is computed with the XOR gate taking its two inputs from the Cout signals of the leftmost and penultimate full adders.

The primary disadvantage of the ripple-carry adder is the time it takes the carry to ripple through all the full adders before a valid result is present in the output. Adder circuits have been extensively studied, because addition is such a basic mathematical operation. The carry-lookahead adder overcomes much of the speed disadvantage of the ripple-carry adder by incorporating a carry-lookahead unit in its design. More sophisticated adders are beyond the scope of this text.

Adder/Subtracter

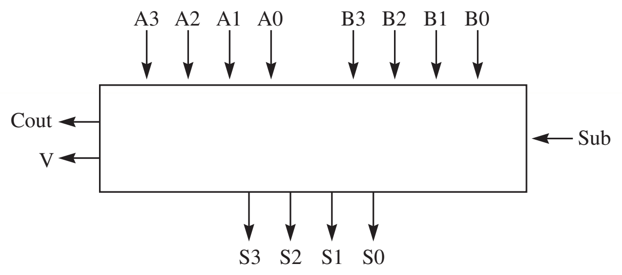
To subtract B from A you could design a subtracter circuit along the same lines as the adder, but with a borrow mechanism that corresponds to the carry mechanism in addition. Rather than build a separate subtracter circuit, however, it is easier to simply negate B and add it to A. Recall the two's complement rule from Chapter 3:

NEG x = 1 + NOT x

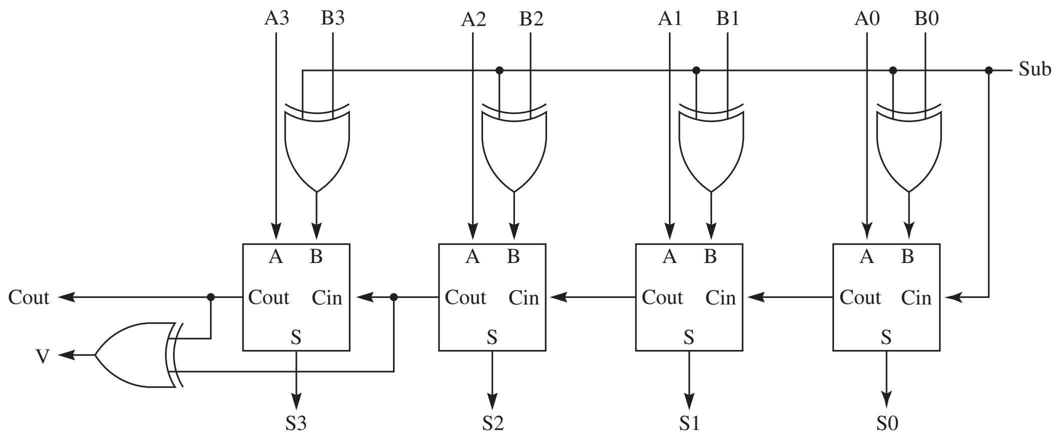
To negate a number, you invert all the bits of the number and then add 1. So, to build a circuit that will function as an adder or a subtracter, we need a way to selectively invert all the bits in B and a way to selectively add 1 to it. Fortunately, the XOR gate comes to the rescue, because you can consider the XOR gate to be a selective inverter.

FIGURE 10.53 shows an adder/subtracter circuit based on this idea. Part (a) is a block diagram that differs from the block diagram of the ripple-carry

FIGURE 10.53
The four-bit ripple-carry adder/subtracter.



(a) Block diagram.



(b) Implementation.

adder only by the addition of a single control line labeled *Sub*. When Sub = 0, the circuit acts like an adder. When Sub = 1, the circuit acts like a subtracter.

Figure 10.53(b) is the implementation. With the adder circuit, you only need a half adder for the LSB. The adder/subtracter replaces it with a full adder. Consider the situation when Sub = 0. In that case, Cin of the least significant full adder is 0 and it acts like a half adder. Furthermore, the left input of each of the top four XOR gates is also 0, which allows the B signals to pass through them unchanged. The circuit computes the sum of A and B.

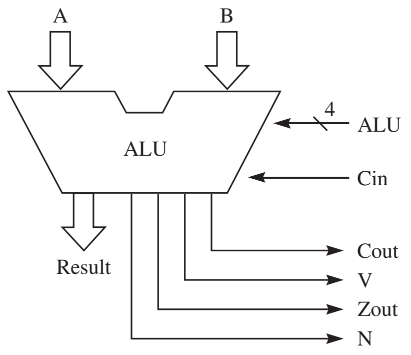
Now consider the case when Sub = 1. Because the left input of the top four XOR gates is 1, the values of all the bits in B are inverted. Furthermore, Cin of the least significant full adder is 1, adding 1 to the result. Consequently, the sum is the sum of A and the negation of B.

Arithmetic Logic Unit

The Pep/9 instructions that perform processing include `ADDr`, `ANDr`, and `ORr`. The addition is an arithmetic operation, whereas AND and OR are logical operations. The CPU typically contains a single combinational circuit called the *arithmetic logic unit (ALU)* that performs these computations.

FIGURE 10.54 shows the ALU for the Pep/9 CPU. A line with a slash represents more than one control line, with the number by the slash specifying the number of lines. The line labeled *ALU* represents four wires. The ALU has a total of 21 input lines—8 lines for the A input, 8 lines for the B input, 4 lines to specify the function that the ALU performs, and the Cin line. It has 12 output lines—8 lines for Result, plus the 4 NZVC values corresponding to Result. The carry output line is labeled *Cout* to distinguish

FIGURE 10.54
Block diagram of the Pep/9 ALU.



it from the carry input line Cin. The zero output line is labeled *Zout* to distinguish it from another Z line in the CPU, as described in Chapter 12.

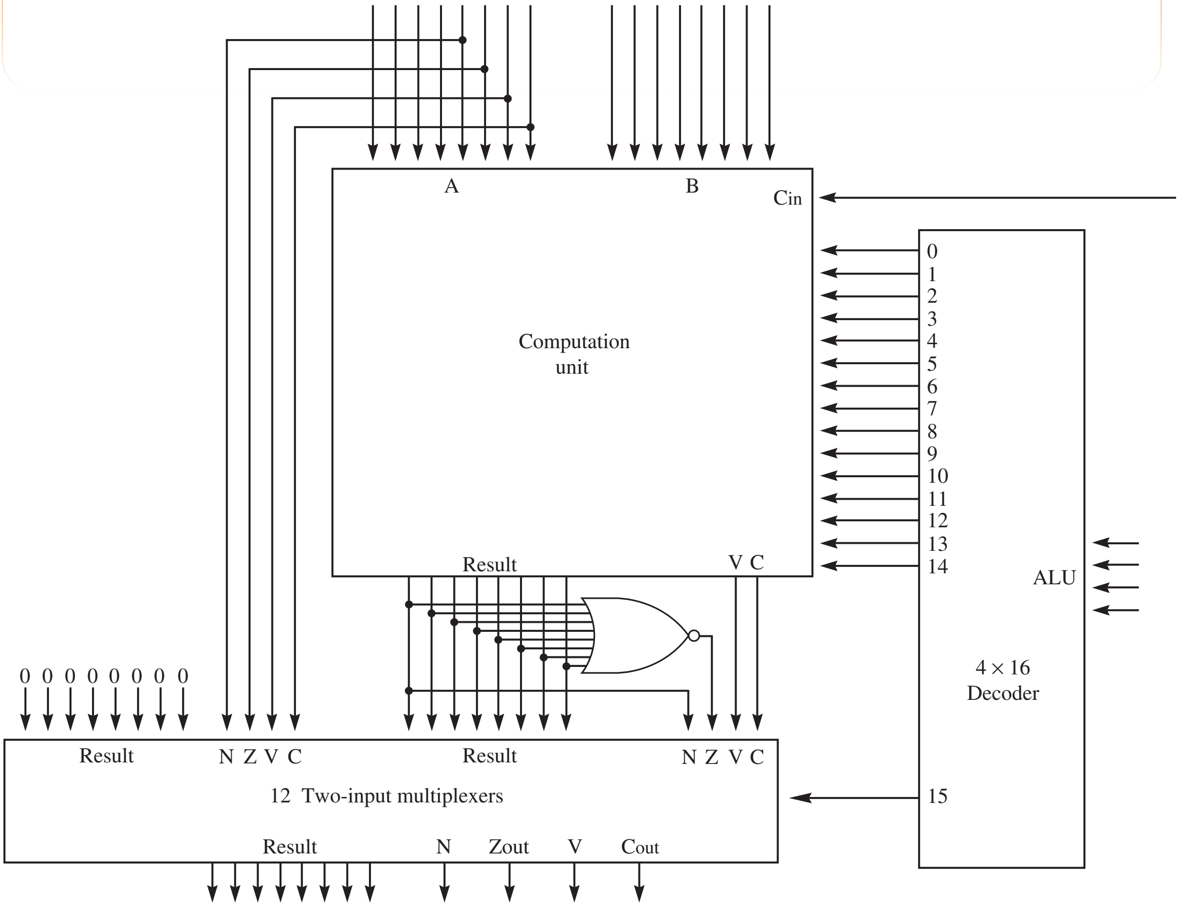
The four ALU control lines specify which of 16 functions the ALU will perform. FIGURE 10.55 lists the 16 functions, most of which correspond directly to the operations available in the Pep/9 instruction set. Because the + symbol is commonly used for the logical OR operation, the arithmetic addition operation is spelled out as "plus." Listed with each operation are the values of the corresponding NZVC bits. The overbar notation is another notation for negation. For example, the notation $\overline{A \cdot B}$ for the NAND function is the same as $(A \cdot B)'$.

FIGURE 10.56 shows the implementation of the ALU. You can see the 21 input lines coming in from the top and the right, and the 12 output lines coming out from the bottom. The four ALU lines that come in from the right drive a 4 × 16 decoder. Recall that depending on the value of the ALU input, exactly one of the output lines of the decoder will be 1 and the others will all

FIGURE 10.55
The 16 functions of the Pep/9 ALU.

ALU Control			Status Bits			
(bin)	(dec)	Result	N	Zout	V	Cout
0000	0	A	N	Z	0	0
0001	1	A plus B	N	Z	V	C
0010	2	A plus B plus Cin	N	Z	V	C
0011	3	A plus $\overline{B}$ plus 1	N	Z	V	C
0100	4	A plus $\overline{B}$ plus Cin	N	Z	V	C
0101	5	$A \cdot B$	N	Z	0	0
0110	6	$\overline{A \cdot B}$	N	Z	0	0
0111	7	A + B	N	Z	0	0
1000	8	$\overline{A + B}$	N	Z	0	0
1001	9	$A \oplus B$	N	Z	0	0
1010	10	$\overline{A}$	N	Z	0	0
1011	11	ASL A	N	Z	V	C
1100	12	ROL A	N	Z	V	C
1101	13	ASR A	N	Z	0	C
1110	14	ROR A	N	Z	0	C
1111	15	0	A<4>	A<5>	A<6>	A<7>

FIGURE 10.56
Implementation of the ALU of Figure 10.54.



be 0. The computation unit inside the ALU performs the first 15 functions of Figure 10.55. Each of the 15 lines from the decoder into the computation unit enables a combinational circuit that performs the function.

The computation unit has 32 input lines—8 lines for the A input, 8 lines for the B input, 1 line for Cin, and 15 lines from the decoder. It has 10 output lines—8 lines for the result of the computation plus 1 line each for V and C. Computation of the N and Z bits is external to the computation unit. Figure 10.56 shows that the N bit is simply a copy of the most significant bit of Result from the computation unit. The Z bit is the NOR of all eight bits of Result. If all eight bits are 0, the output of the NOR gate is 1. If one or more inputs are 1, the output of the NOR gate is 0. These are precisely the

Computation of the N and Z bits

conditions for which the Z bit should be set, depending on the result of the computation.

The bottom box on the left is a set of 12 two-input multiplexers. The control line of each multiplexer is tied to line 15 from the decoder. The control line acts as follows:

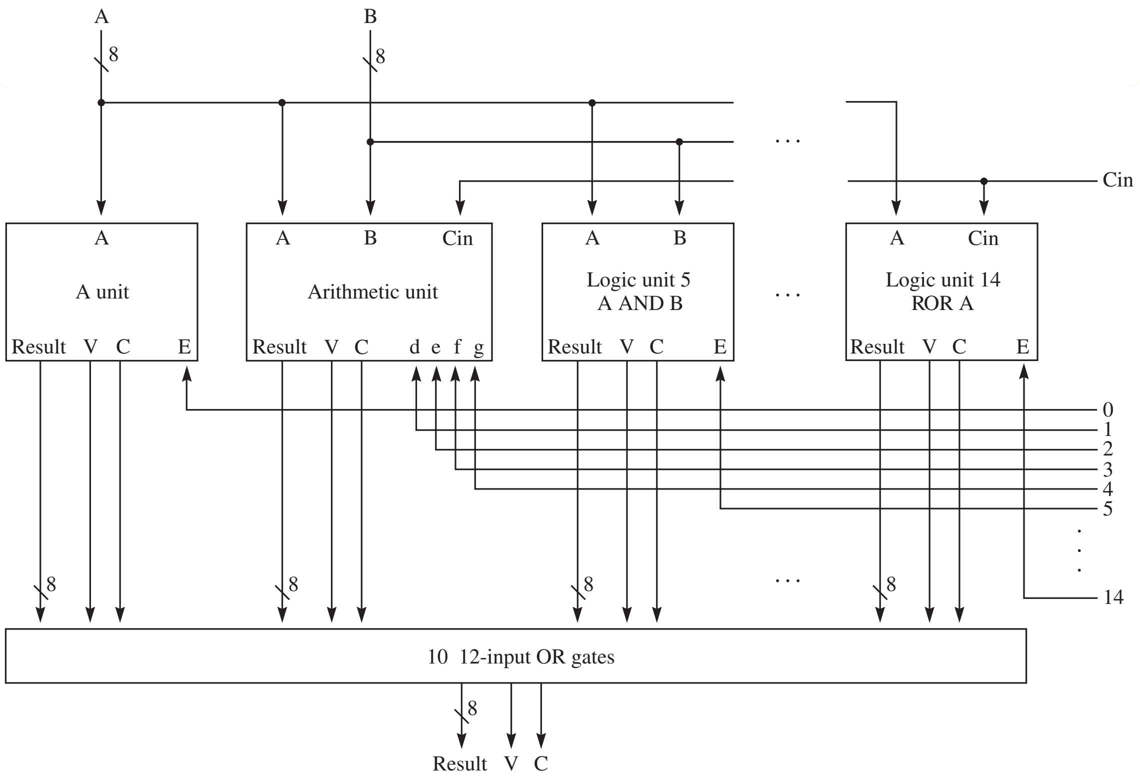
The multiplexer of Figure 10.56

- If line 15 is 1, Result and NZVC from the left are routed to the output.
- If line 15 is 0, Result and NZVC from the right are routed to the output.

You can see how Figure 10.56 computes the last function of Figure 10.55. If the ALU input is 1111 (bin), then line 15 is 1, and Result and NZVC from the left are routed to the output of the ALU. But Figure 10.56 shows that Result from the left is tied to 0 and NZVC comes from the low nybble (half byte) of A, as required.

FIGURE 10.57 is an implementation of the computation unit of Figure 10.56. It consists of 1 A Unit, 1 arithmetic unit, and 10 logic units

FIGURE 10.57
Implementation of the computation unit of Figure 10.56.



labeled *Logic unit 5* through *Logic unit 14*. The A unit and the logic units are each enabled by 1 of the 15 decoder lines. If the enable line E of any unit is 0, then all bits of Result as well as V and C are 0 regardless of any other input to the unit. The arithmetic unit is responsible for computing Result, V, and C for the arithmetic operations that correspond to functions 1, 2, 3, and 4 in Figure 10.55. The corresponding control lines for the arithmetic unit are labeled *d*, *e*, *f*, and *g*, respectively. If all four of d, e, f, and g are 0, then all bits of Result as well as V and C are 0, regardless of any other input to the arithmetic unit.

Each output line of a computation unit feeds into a 12-input OR gate. The other 11 inputs to the OR gate are the corresponding lines from the other 11 computation units. For example, the V outputs of all 12 computation units feed into one OR gate. Because 11 of the computation units are guaranteed to be disabled, exactly 11 inputs are guaranteed to be 0 for every OR gate. The one input that is not guaranteed to be 0 is the input from the unit that is enabled. Because 0 is the identity for the OR operation

$$p \text{ OR } 0 = p$$

the output from the unit that is enabled passes through the OR gate unchanged.

FIGURE 10.58 is an implementation of the A unit. It consists of eight two-input AND gates that act as enable gates for the eight bits of the A signal. Figure 10.55 specifies that V and C should be 0. Consequently, both output lines for V and C are tied to 0 in the implementation.

FIGURE 10.59 is an implementation of the arithmetic unit. It is an extension of the adder/subtracter circuit of Figure 10.53, modified to handle two additional cases for adding and subtracting 16-bit values with two 8-bit

FIGURE 10.58
Implementation of the A unit of Figure 10.57.

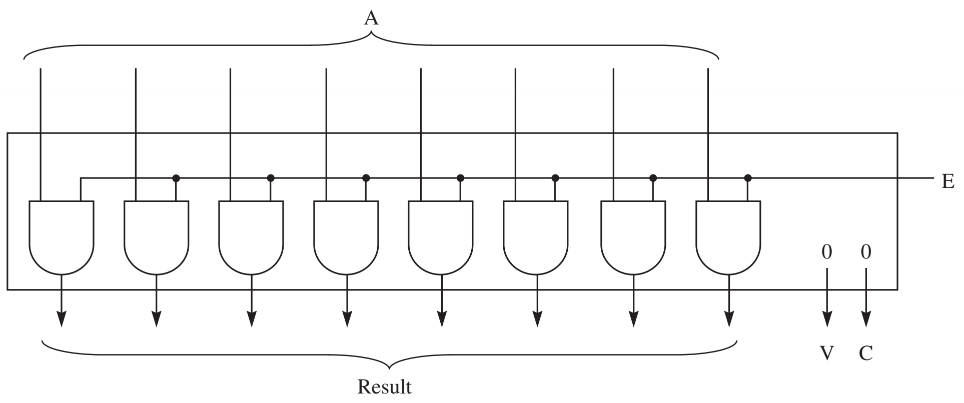
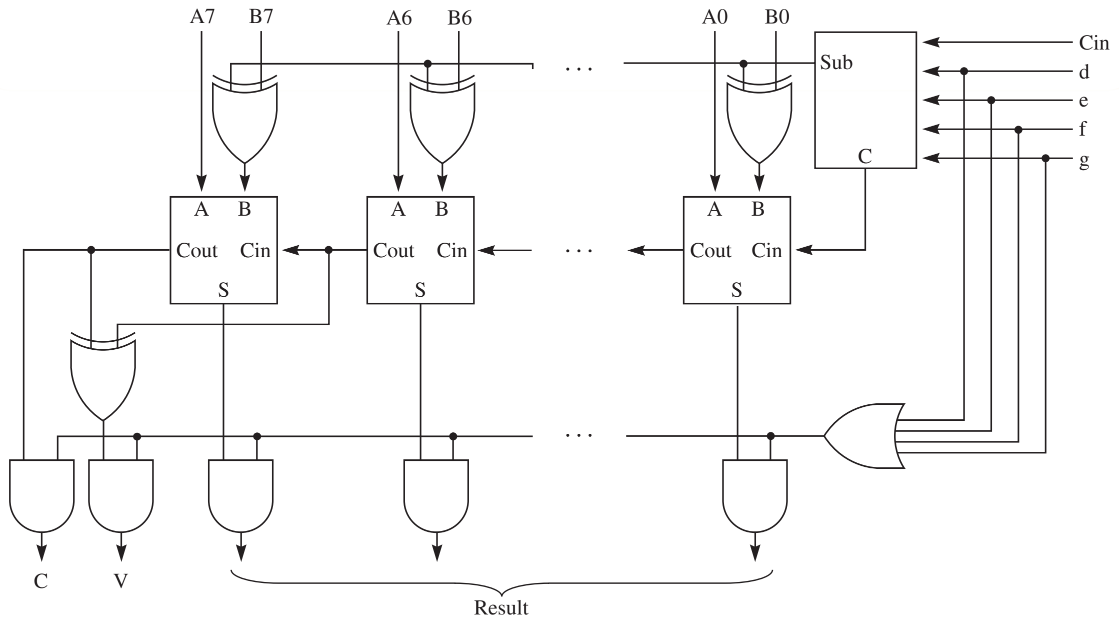


FIGURE 10.59
Implementation of the arithmetic unit of Figure 10.57.



operations. FIGURE 10.60 shows how to do a 16-bit operation with two 8-bit operations. In Figure 10.60(a), you do a 16-bit add with

A plus B

on the low-order bytes of A and B, followed by

A plus B plus Cin

on the high-order bytes, where Cin is the Cout of the low-order operation. In Figure 10.60(b), you do a 16-bit subtraction with

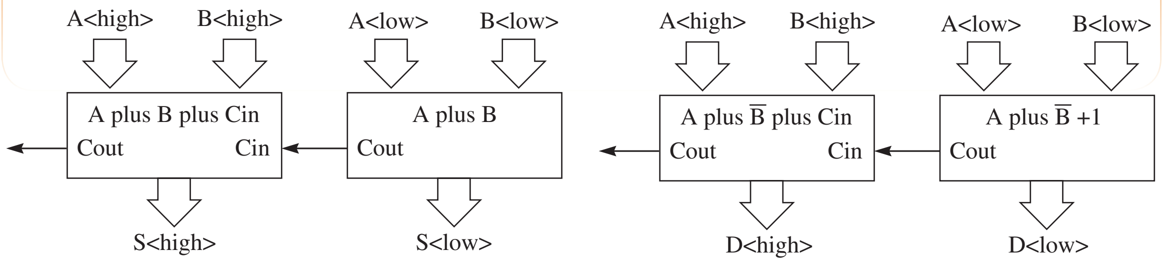
A plus $\overline{B}$ plus 1

on the low-order bytes of A and B, followed by

A plus $\overline{B}$ plus Cin

on the high-order bytes, where Cin is again the Cout of the low-order operation. This last operation follows from the fact that subtracting B from A is performed in hardware by adding the two's complement of B to A. The carry out of the low-order operation is the carry out of an addition, not a subtraction. That is why the circuit adds Cin from the low-order operation instead of subtracting it.

FIGURE 10.60
Using two 8-bit operations to produce a 16-bit operation.



(a) 16-bit addition.

(b) 16-bit subtraction.

Example 10.38 Here is how the hardware subtracts 259 from 261. As a 16-bit quantity, 259 (dec) = 0000 0001 0000 0011, so that A<high> = 0000 0001 and A<low> = 0000 0011. As a 16-bit quantity, 261 (dec) = 0000 0001 0000 0101, so that B<high> = 0000 0001 and B<low> = 0000 0101. The low-order addition is

```
           0000 0011
           1111 1010
ADD                1
--------------------
C = 0      1111 1110
```

and the high-order addition is

```
           0000 0001
           1111 1110
ADD                0
--------------------
C = 0      1111 1111
V = 0
```

The final difference is 1111 1111 1111 1110 (bin) = –2 (dec), as expected. The final V bit is computed from the exclusive OR of the final carry out with the penultimate carry out as follows:

$0 \oplus 0 = 0$ ▮

Example 10.39 Here is how the hardware subtracts 261 from 259. This time, A<high> = 0000 0001, A<low> = 0000 0101, B<high> = 0000 0001, and B<low> = 0000 0011. The low-order addition is

```
           0000 0101
           1111 1100
ADD                1
--------------------
C = 1      0000 0010
```

and the high-order addition is

```
                0000 0001
                1111 1110
ADD                     1
-------------------------
C = 1           0000 0000
V = 0
```

The final difference is 0000 0000 0000 0010 (bin) = 2 (dec), as expected. The final V bit is computed from the exclusive OR of the final carry out with the penultimate carry out as follows:

$$1 \oplus 1 = 0$$

The control circuit box on the top right part of Figure 10.59 controls the function of the circuit. FIGURE 10.61 is its truth table. Compare the box with the Sub line that controls the adder/subtracter circuit in Figure 10.53. When Sub is 0 in the adder/subtracter, B is not inverted by the XOR gates, and the carry in of the low-order bit is 0. When Sub is 1, B is inverted, and the carry in of the low-order bit is 1. The first and third rows of Figure 10.61 duplicate these two functions. The second row of Figure 10.61 is for the high-order addition, and the last row is for the high-order subtraction.

Theoretically, the Sub and C outputs of the control box are functions of d, e, f, g, and Cin. Inspection of the truth table, however, shows that Sub can be expressed as

$$\text{Sub} = f + g$$

and C can be expressed as

$$C = (\text{Cin} + f) \cdot d'$$

with neither depending on e.

Another requirement of the arithmetic unit is that if d, e, f, and g are all 0, then all the outputs must be 0 regardless of the other inputs. The output

FIGURE 10.61
The truth table for the control circuit in Figure 10.59.

Function	d	e	f	g	Sub	C
A plus B	1	0	0	0	0	0
A plus B plus Cin	0	1	0	0	0	Cin
A plus $\overline{B}$ plus 1	0	0	1	0	1	1
A plus $\overline{B}$ plus Cin	0	0	0	1	1	Cin

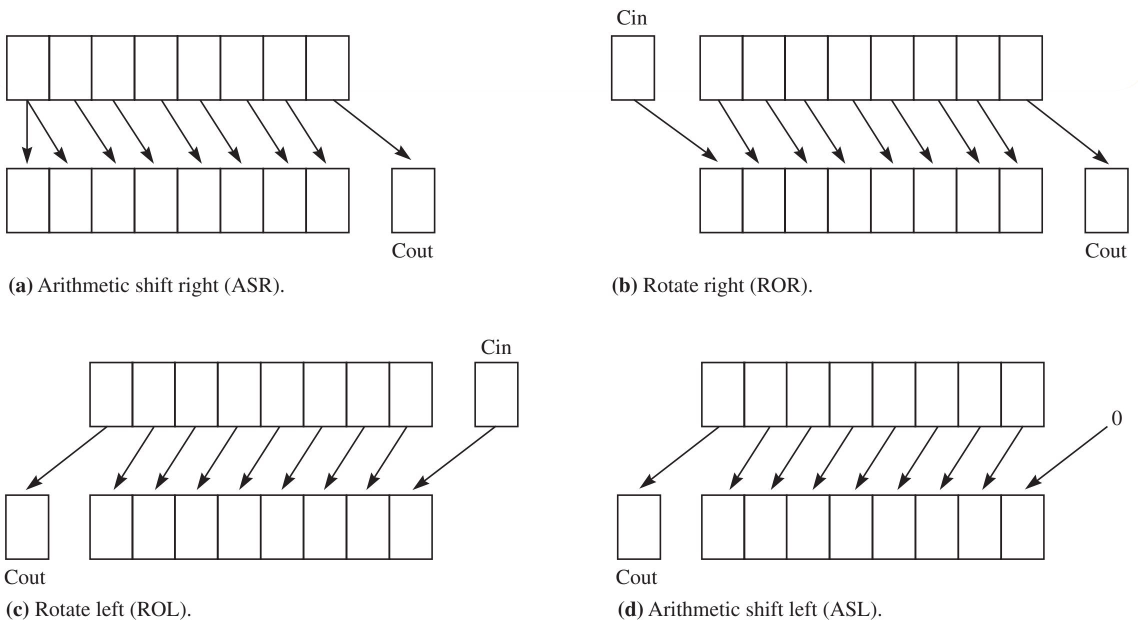
of the four-input OR gate in Figure 10.59 acts as the enable signal, which allows all 10 outputs of the unit to pass through when one of d, e, f, or g is 1.

In the same way that a 16-bit addition can be composed of an 8-bit addition of the low-order bytes followed by an 8-bit addition of the high-order bytes, the shift and rotate operations can be composed of two 8-bit operations. FIGURE 10.62 shows the specification of the shift and rotate operations for the Pep/9 ALU of Figure 10.55. Although it is not shown in Figure 10.62, the V bit is set by both the ROL (rotate left) and ASL (arithmetic shift left) operations.

Figures 10.62(a) and (b) show that a 16-bit arithmetic shift right is composed first of an 8-bit arithmetic shift right of the high-order byte. That shift duplicates the sign bit and shifts its low-order bit into Cout. Following that operation, a rotate right of the low-order byte takes the Cout from the high-order shift as its Cin and rotates its low-order bit into Cout.

Figures 10.62(c) and (d) show that a 16-bit arithmetic shift left starts with an arithmetic shift left of the low-order byte in part (d). Cout gets the most significant bit, which is used as Cin for the subsequent rotate left of the high-order byte in part (c). The second step shows why the ALU sets the V bit on the ROL operation. Figure 5.2 shows that the `ROLr` instruction at Level Asmb5 does not affect the V bit. However, at this lower level of

FIGURE 10.62
Specification of the shift and rotate operations.



(a) Arithmetic shift right (ASR).
(b) Rotate right (ROR).
(c) Rotate left (ROL).
(d) Arithmetic shift left (ASL).

abstraction, LG1, the ALU sets the V bit because the ROL function at this level is used to implement the `ASLr` instruction at the higher level.

Implementation of logic units 5 through 14 is left as an exercise for the student. They are straightforward to implement, because the logic operations are available as common logic gates.

Abstraction at Level LG1

Abstract data types (ADTs) are an important design tool at Level HOL6. The idea is that you should understand the behavior of an ADT by knowing what the functions and procedures that operate on the ADT do, not necessarily how they do it. Once an operation has been implemented, you can free your mind of the implementation details and concentrate on solving the problem at a higher level of abstraction.

The same principle operates at the hardware level. Each combinational device in this section has a block diagram and a truth table that describes its function. The block diagram is to hardware what an ADT is to software. It is an abstraction that specifies the input and output while hiding the implementation details.

The block diagram as an ADT

Higher levels of abstraction in the hardware are obtained by constructing devices defined by block diagrams whose implementation is an interconnection of blocks at a lower level of abstraction. Figure 10.50 is a perfect example. This full adder block is implemented with the half adder blocks in Figure 10.51.

The highest level of abstraction for the hardware is the block diagram of the Pep/9 computer we have seen repeatedly. The three blocks—disk, CPU, and main memory with memory-mapped I/O devices—are connected by the bus. At a slightly lower level of abstraction, you see the registers in the CPU. Each register is depicted as a block. The remaining two chapters build up successively higher levels of abstraction, culminating with the Pep/9 computer at Level ISA3.

Chapter Summary

In a combinational circuit, the input determines the output. Three representations of a combinational circuit are truth tables, Boolean algebraic expressions, and logic diagrams. Of the three representations, truth tables are at the highest level of abstraction. They specify the function of a circuit without specifying its implementation. A truth table lists the output for all possible combinations of the input, hence the name *combinational circuit*.

The three basic operations of Boolean algebra are AND, OR, and NOT. The 10 fundamental properties of Boolean algebra consist of 5 laws—commutative, associative, distributive, identity, and complement—and their duals, from which useful Boolean theorems may be proved. An important theorem is De Morgan's law, which shows how to take the NOT of the AND or OR of several terms.

A Boolean expression corresponds to a logic diagram, which in turn corresponds to a connection of electronic gates. Three common gates are NAND (AND followed by NOT), NOR (OR followed by NOT), and XOR (exclusive OR). Two-level circuits minimize processing time, but may require more gates than an equivalent multilevel circuit. This is another manifestation of the fundamental space/time tradeoff. Karnaugh maps help minimize the number of gates needed to implement a two-level combinational circuit.

Combinational devices include the multiplexer, the decoder, the demultiplexer, the adder, and the arithmetic logic unit (ALU). A multiplexer selects one of several data inputs to be routed to a single data output. A decoder takes a binary number as input and sets one of several data output lines to 1 and the rest to 0. A demultiplexer routes one of several data input values to a single output line and is logically equivalent to a decoder with an enable line. A half adder adds two bits, and a full adder adds three bits, one of which is the previous carry. A subtracter works by negating the second operand and adding it to the first. An ALU performs both arithmetic and logic functions.

Exercises

Section 10.1

1. *(**a**) Prove the zero theorem $x + 1 = 1$ with Boolean algebra. Give a reason for each step in your proof. Hint: Expand the 1 on the left with the complement property and then use the idempotent property. (**b**) Show the dual proof of part (a).

2. (**a**) Prove with Boolean algebra the absorption property, $x + x \cdot y = x$. Give a reason for each step in your proof. (**b**) Show the dual proof of part (a).

3. (**a**) Prove with Boolean algebra the consensus theorem $x \cdot y + x' \cdot z + y \cdot z = x \cdot y + x' \cdot z$. Give a reason for each step in your proof. (**b**) Show the dual proof of part (a).

***4.** Prove De Morgan's law, $(a + b)' = a' \cdot b'$, by giving the dual of the proof in the text. Give a reason for each step in your proof.

5. **(a)** Prove the general form of De Morgan's law,

$(a_1 \cdot a_2 \cdot \cdots \cdot a_n)' = a_1' + a_2' + \cdots + a_n'$ where $n \geq 2$

from De Morgan's law for two variables using mathematical induction. **(b)** Show the dual proof of part (a).

6. **(a)** Prove with Boolean algebra that $(x + y) \cdot (x' + y) = y$. Give a reason for each step in your proof. **(b)** Show the dual proof of part (a).

7. **(a)** Prove with Boolean algebra that $(x + y) + (y \cdot x') = x + y$. Give a reason for each step in your proof. **(b)** Show the dual proof of part (a).

8. ***(a)** Draw a three-input OR gate, its Boolean expression, and its truth table, as in Figure 10.10. **(b)** Do part (a) for the three-input NAND gate. **(c)** Do part (a) for the three-input NOR gate.

9. For each of the following Boolean properties or theorems, state the set theory interpretation:

***(a)** $x + 0 = x$ **(b)** $x \cdot 1 = x$ **(c)** $x + x' = 1$ **(d)** $x \cdot x' = 0$
(e) $x \cdot x = x$ **(f)** $x + x = x$ **(g)** $x \cdot 0 = 0$

10. ***(a)** Show the associative property for the OR operation using Venn diagrams with x, y, and z overlapping regions. Sketch the following regions to show that region (3) is the same as region (6):

(1) $(x + y)$ (2) z (3) $(x + y) + z$
(4) x (5) $(y + z)$ (6) $x + (y + z)$

(b) Do the dual of part (a).

11. **(a)** Show the distributive property using Venn diagrams with x, y, and z overlapping regions. Sketch the following regions to show that region (3) is the same as region (6):

(1) x (2) $y \cdot z$ (3) $x + y \cdot z$
(4) $(x + y)$ (5) $(x + z)$ (6) $(x + y) \cdot (x + z)$

(b) Do the dual of part (a).

12. **(a)** Show De Morgan's law using Venn diagrams with a and b overlapping regions. Sketch the following regions to show that region (2) is the same as region (5):

(1) $a \cdot b$ (2) $(a \cdot b)'$ (3) a' (4) b' (5) $a' + b'$

(b) Do the dual of part (a).

13. Although a Boolean variable for a combinational circuit can have only two values, 1 or 0, Boolean algebra can describe a system where a variable can have one of four possible values—0, 1, A, or B. Such a system corresponds to the description of subsets of $\{a, b\}$ where $1 = \{a, b\}$ (the universal set), $A = \{a\}$, $B = \{b\}$, and $0 = \{\}$ (the empty set). The truth tables for two-input AND and OR operations have 16 entries instead of 4, and the truth table for the complement has 4 entries instead of 2. Construct the truth table for the following:

*(a) AND (b) OR (c) the complement

14. The exclusive NOR gate, written *XNOR*, is equivalent to an XOR followed by an inverter. ***(a)** Draw the symbol for a two-input XNOR gate. **(b)** Construct its truth table. **(c)** The XNOR is also called a *comparator*. Why?

Section 10.2

15. Draw the nonabbreviated logic diagram for the following Boolean expressions. You may use XOR gates.

*(a) $((a')')'$ (b) $(((a')')')'$
*(c) $a'b + ab'$ (d) $ab + a'b'$
(e) $ab + ab' + a'b$ (f) $((ab \oplus b')' + a'b)'$
(g) $(a'bc + a)b$ (h) $(ab'c)'(ac)'$
(i) $((ab)'(b'c)' + a'b'c')'$ (j) $(a \oplus b + b' \oplus c')'$
(k) $(abc)' + (a'b'c')'$ (l) $(a + b)(a' + c)(b' + c')$
(m) $(a \oplus b) \oplus c + ab'c$ (n) $(((a + b)' + c)' + d)'$
(o) $(ab' + b'c + cd)'$ (p) $((a + b')(b' + c)(c + d))'$
(q) $(((ab)'c)'d)'$ (r) $(((a \oplus b)' \oplus c)' \oplus d)'$

16. Draw the abbreviated logic diagram for the Boolean expressions of Exercise 15. You may use XOR gates.

17. Construct the truth tables for the Boolean expressions of Exercise 15.

18. Write the Boolean expressions for the logic diagrams of FIGURE 10.63.

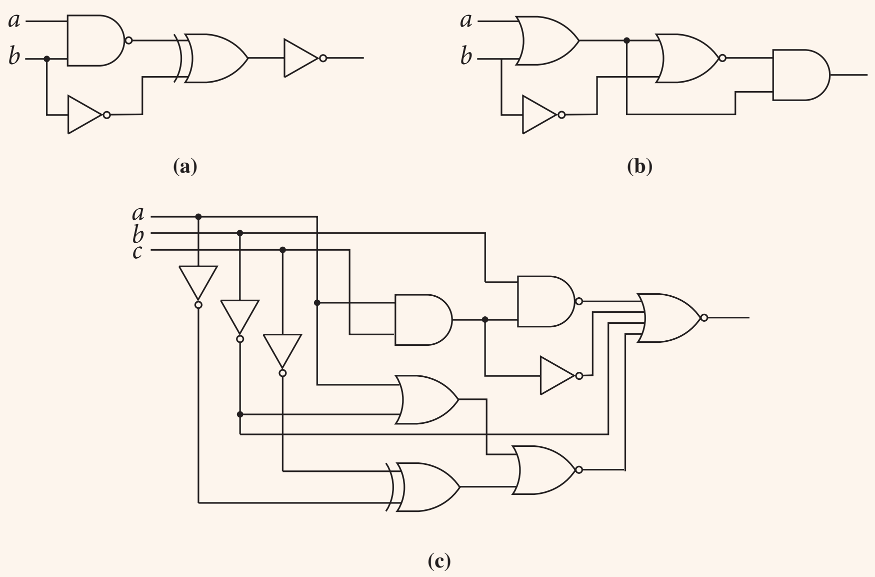
19. Write the Boolean AND-OR expression for the following:

*(a) function y in Figure 10.3
(b) function y in Figure 10.4
(c) function x in Figure 10.17
(d) the NAND gate in Figure 10.7(a)
(e) the XOR gate in Figure 10.7(c)

20. Write the Boolean OR-AND expression for the following:

*(a) function y in Figure 10.3
(b) function x in Figure 10.17
(c) the NOR gate in Figure 10.7(b)
(d) the XOR gate in Figure 10.7(c)

FIGURE 10.63
The logic diagrams for Exercise 18.



21. Use the properties and theorems of Boolean algebra to reduce the following expressions to AND-OR expressions without parentheses. The expressions may not be unique. Construct the truth table, which will be unique, by inspection of your final expression.

*__(a)__ $(a'b + ab')'$
(b) $(ab + a'b')'$
(c) $(ab + ab' + a'b)'$
*__(d)__ $(ab \oplus b')' + ab$
(e) $(a'bc + a)b$
(f) $(ab'c)'(ac)'$
(g) $(a \oplus b) \oplus c$
(h) $a \oplus (b \oplus c)$
(i) $(a + b)(a' + c)(b' + c')$
(j) $((a + b)' + c)'$

***22.** Construct two-level circuits for the expressions of Exercise 21 using only NAND gates.

23. Use the properties and theorems of Boolean algebra to reduce the following expressions to OR-AND expressions. The expressions may not be unique. Construct the truth table, which will be unique, by inspection of your final expression.

(a) $a'b + ab'$
*__(b)__ $ab + a'b'$
(c) $ab + ab' + a'b$
(d) $((ab \oplus b')' + ab)'$
(e) $(a'bc + a)b$
(f) $(ab'c)'(ac)'$
(g) $(a \oplus b) \oplus c$
(h) $a \oplus (b \oplus c)$
(i) $((a + b)(a' + c)(b' + c'))'$
(j) $(a + b)' + c$

***24.** Construct a two-level circuit for the expressions of Exercise 23 using only NOR gates.

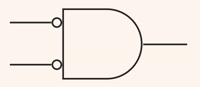
25. Draw the logic diagram of a two-level circuit that produces the XOR function using the following:

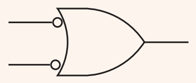
***(a)** only NAND gates **(b)** only NOR gates

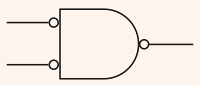
26. State whether each gate in FIGURE 10.64 is the following:

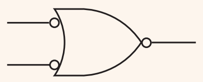
(1) an AND gate (2) an OR gate
(3) a NAND gate (4) a NOR gate

FIGURE 10.64
The gates for Exercise 26.


(a)


(b)


(c)


(d)

Section 10.3

***27.** Write each function of Exercise 21 with the sigma notation.

***28.** Write each function of Exercise 23 with the pi notation.

29. In Figure 10.3, find the minimum AND-OR expression for the following:

***(a)** $x(a, b, c)$ **(b)** $y(a, b, c)$

Draw the minimized two-level circuit for each expression with only NAND gates.

30. In Figure 10.3, find the minimum OR-AND expression for the following:

***(a)** $x(a, b, c)$ **(b)** $y(a, b, c)$

Draw the minimized two-level circuit for each expression with only NOR gates.

31. Use a Karnaugh map to find the minimum AND-OR expression for $x(a, b, c)$:

***(a)** $\Sigma(0, 4, 5, 7)$ **(b)** $\Sigma(2, 3, 4, 6, 7)$ **(c)** $\Sigma(0, 3, 5, 6)$
(d) $\Sigma(0, 1, 2, 3, 4, 6)$ **(e)** $\Sigma(1, 2, 3, 4, 5)$ **(f)** $\Sigma(1, 2, 3, 4, 5, 6, 7)$
(g) $\Sigma(0, 1, 2, 4, 6)$ **(h)** $\Sigma(1, 4, 6, 7)$ **(i)** $\Sigma(2, 3, 4, 5, 6)$
(j) $\Sigma(0, 2, 5)$

***32.** Write each expression of Exercise 31 in pi notation. Use a Karnaugh map to find its minimum OR-AND expression.

33. Use a Karnaugh map to find the minimum AND-OR expression for $x(a, b, c, d)$:

***(a)** $\Sigma(2, 3, 4, 5, 10, 12, 13)$
(b) $\Sigma(1, 5, 6, 7, 9, 12, 13, 15)$
(c) $\Sigma(0, 1, 2, 4, 6, 8, 10)$
(d) $\Sigma(7)$
(e) $\Sigma(2, 4, 5, 11, 13, 15)$
(f) $\Sigma(1, 2, 4, 5, 6, 7, 12, 15)$

(g) $\Sigma(1, 2, 4, 5, 6, 7, 8, 11, 12, 15)$
(h) $\Sigma(1, 7, 10, 12)$
(i) $\Sigma(0, 2, 3, 4, 5, 6, 8, 10, 11, 13)$
(j) $\Sigma(0, 1, 2, 3, 4, 5, 6, 10, 11, 13, 14, 15)$
(k) $\Sigma(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)$

***34.** Write each expression of Exercise 33 in pi notation. Use a Karnaugh map to find its minimum OR-AND expression.

35. Use a Karnaugh map to find the minimum AND-OR expression for $x(a, b, c)$ with don't-care conditions:

***(a)** $\Sigma(0, 6) + d(1, 3, 7)$
(b) $\Sigma(5) + d(0, 2, 4, 6)$
(c) $\Sigma(1, 3) + d(0, 2, 4, 6)$
(d) $\Sigma(0, 5, 7) + d(3, 4)$
(e) $\Sigma(1, 7) + d(2, 4)$
(f) $\Sigma(4, 5, 6) + d(1, 2, 3, 7)$

36. Use a Karnaugh map to find the minimum AND-OR expression for $x(a, b, c, d)$ with don't-care conditions:

***(a)** $\Sigma(5, 6) + d(2, 7, 9, 13, 14, 15)$
(b) $\Sigma(0, 3, 14) + d(2, 4, 7, 8, 10, 11, 13, 15)$
(c) $\Sigma(3, 4, 5, 10) + d(2, 11, 13, 15)$
(d) $\Sigma(5, 6, 12, 15) + d(0, 4, 10, 14)$
(e) $\Sigma(1, 6, 9, 12) + d(0, 2, 3, 4, 5, 7, 14, 15)$
(f) $\Sigma(0, 2, 3, 4) + d(8, 9, 10, 11, 13, 14, 15)$
(g) $\Sigma(2, 3, 10) + d(0, 4, 6, 7, 8, 9, 12, 14, 15)$

37. **(a)** A Karnaugh map for three variables has minterm 0 adjacent to 2, and 4 adjacent to 6. Copy Figure 10.30, cut out the Karnaugh map, and tape it in the shape of a cylinder so that adjacent minterms are physically adjacent. **(b)** For adjacent minterms to be physically adjacent in a four-variable Karnaugh map requires a three-dimensional *torus* (shaped like a doughnut). Construct a torus from clay or some other suitable material and inscribe or write on it the cells and their decimal labels of Figure 10.33(a). For example, the cell with 2 should be physically adjacent to the cells with 0, 3, 6, and 10.

Section 10.4

38. Using the viewpoint that one of the lines is a data line and the other is a control line, explain the operation of each of the following two-input gates:

***(a)** OR
(b) NAND
(c) NOR
(d) XNOR

See Exercise 14 for the definition of XNOR.

39. Draw a nonabbreviated logic diagram of an eight-input multiplexer.

***40.** Construct a 16-input multiplexer from five 4-input multiplexers. Draw the 16-input multiplexer as a large block with 16 data lines labeled *D0* through *D15* and 4 select lines labeled *S3* through *S0*. Inside the large block, draw each 4-input multiplexer as a small block with data lines D0 through D3 and select lines S1 and S0. Show the connections to the small blocks from the outside lines and the connections between the blocks to implement the big multiplexer. Explain the operation of your circuit.

41. Do Exercise 40 with two eight-input multiplexers without enable inputs and any other gates you need. Explain the operation of your circuit.

42. ***(a)** Draw a nonabbreviated logic diagram of a 3 × 8 binary decoder. **(b)** Draw a nonabbreviated logic diagram of a 2 × 4 binary decoder with an enable input.

43. Construct a 4 × 16 binary decoder without an enable input from five 2 × 4 binary decoders with enable inputs. You may use the constant 1 as input to a device. Use the drawing guidelines of Exercise 40 to label your external and internal lines. Explain the operation of your circuit.

44. Construct a 4 × 16 binary decoder without an enable input from two 3 × 8 binary decoders with enable inputs plus any other gates you need. Use the drawing guidelines of Exercise 40 to label your external and internal lines. Explain the operation of your circuit.

45. Implement the 2 × 4 binary decoder with an enable input, as shown in Figure 10.47. Draw a nonabbreviated diagram of your circuit.

46. ***(a)** Draw the implementation of the full adder in Figure 10.51 showing the AND and XOR gates of the half adders. ***(b)** What is the maximum number of gate delays from input to output? **(c)** Design minimized two-level networks for Sum and Cout from the truth table of Figure 10.50(b). **(d)** Compute the percentage change in the number of gates and in the processing time for the design of part (c) compared to part (a). How do your results illustrate the space/time tradeoff?

47. **(a)** Draw the circuit of Figure 10.52 with the individual XOR, AND, and OR gates of the half adders. ***(b)** What is the maximum number of gate delays from input to output? Consider an XOR gate as requiring one gate delay. This problem requires some thought. Assume that all eight inputs are presented at the same time, even though the carry will ripple through the circuit.

48. Modify Figure 10.52(b) to provide two additional outputs, one for the N bit and one for the Z bit.

49. Implement a four-bit ASL shifter with select line S. The input is A3 A2 A1 A0, which represents a four-bit number with A0 the LSB and A3 the sign bit. The output is B3 B2 B1 B0 and C, the carry bit. If S is 1, the output is the ASL of the input. If S is 0, the output is the same as the input, and C is 0.

50. Do Exercise 49 for a four-bit ASR shifter.

51. The block diagram in FIGURE 10.65 is a three-input, two-output combinational switching circuit. If *s* is 0, the input *a* is routed directly through to *x*, and *b* is routed to *y*. If *s* is 1, they are switched, with *a* being routed to *y* and *b* to *x*. Construct the circuit using only AND, OR, and inverter gates.

52. The block diagram in FIGURE 10.66 is a four-input, two-output combinational switching circuit. If *s1 s0* = 00, the *a* input is broadcast to *x* and *y*. If *s1 s0* = 01, the *b* input is broadcast to *x* and *y*. If *s1 s0* = 10, *a* and *b* pass straight through to *x* and *y*. If *s1 s0* = 11, they are switched, with *a* being routed to *y* and *b* to *x*. Construct the circuit using only AND, OR, and inverter gates. **(a)** Use Karnaugh maps to construct the minimum AND-OR circuit. **(b)** Use Karnaugh maps to construct the minimum OR-AND circuit.

53. Draw the 12 two-input multiplexers of Figure 10.56. Show all the connections to the input and output lines. You may use ellipses (. . .) for 6 of the 8 data lines.

FIGURE 10.65
The block diagram for Exercise 51.

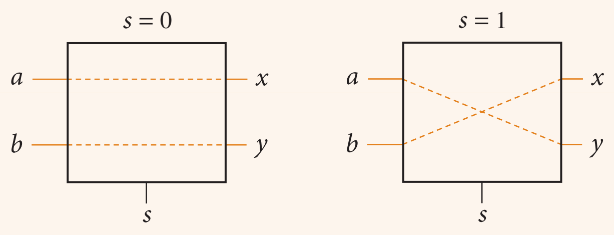
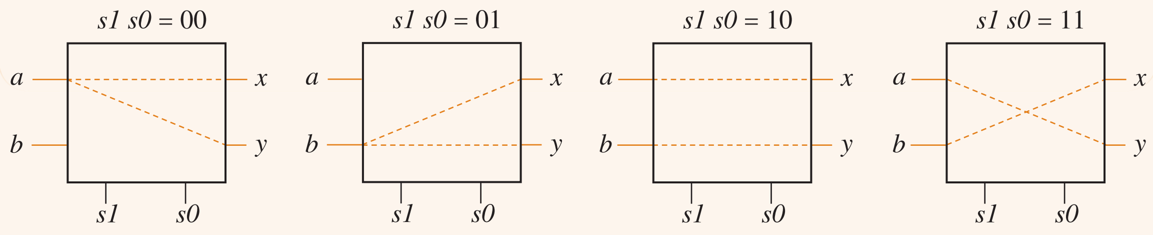


FIGURE 10.66
The block diagram for Exercise 52.



54. Implement the following logic units for the Pep/9 ALU:

(a) logic unit 5, A · B

(b) logic unit 6, $\overline{\text{A} \cdot \text{B}}$

(c) logic unit 7, A + B

(d) logic unit 8, $\overline{\text{A} + \text{B}}$

(e) logic unit 9, A ⊕ B

(f) logic unit 10, $\overline{\text{A}}$

(g) logic unit 11, ASL A

(h) logic unit 12, ROL A

(i) logic unit 13, ASR A

(j) logic unit 14, ROR A

55. Draw the nonabbreviated implementation of the five-input, two-output control box of Figure 10.59.