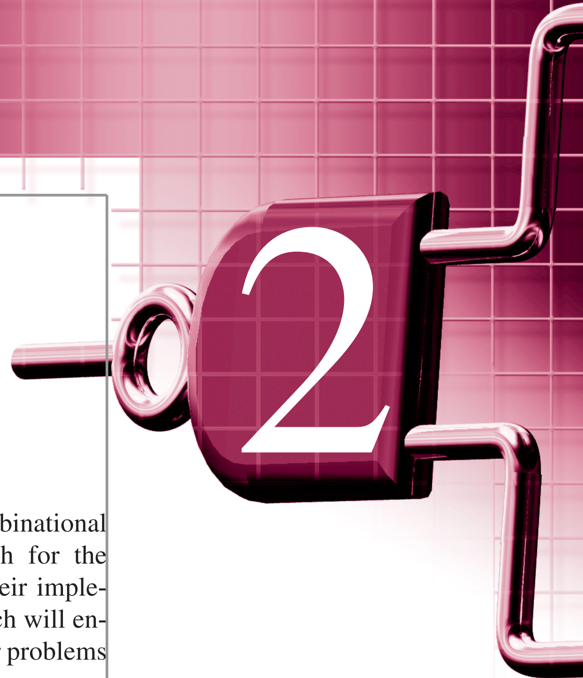


Combinational Systems

In this chapter, we will develop the tools to specify combinational systems. Then, we will develop an algebraic approach for the description of these systems, their simplification, and their implementation. We will concentrate on rather small systems, which will enable us to better understand the process. We will look at larger problems in Chapter 4.

2.1 THE DESIGN PROCESS FOR COMBINATIONAL SYSTEMS

In this section, we will outline the process to be used to design combinational systems. (A similar process will be developed in Chapter 6 for sequential systems.) The design process typically starts with a problem statement, a verbal description of the intended system. The goal is to develop a block diagram of that system, utilizing available components and meeting the design objectives and constraints.

We will use the following five examples to illustrate the steps in the design process and, indeed, continue to follow some of them in subsequent chapters, as we develop the tools necessary to do that design.

Continuing Examples (CE)

CE1. A system with four inputs, *A, B, C,* and *D,* and one output, *Z,* such that $Z = 1$ iff three of the inputs are 1.

CE2. A single light (that can be on or off) that can be controlled by any one of three switches. One switch is the master on/off switch. If it is down, the lights are off. When the master switch is up, a change in the position of one of the other switches (from up to down or from down to up) will cause the light to change state.

CE3. A system to do 1 bit of binary addition. It has three inputs (the 2 bits to be added plus the carry from the next lower order bit) and produces two outputs: a sum bit and a carry to the next higher order position.

FIRST PAGES

CE4. A display driver; a system that has as its input the code for a decimal digit and produces as its output the signals to drive a seven-segment display, such as those on most digital watches and numeric displays (more later).

CE5. A system with nine inputs, representing two 4-bit binary numbers and a carry input, and one 5-bit output, representing the sum. (Each input number can range from 0 to 15; the output can range from 0 to 31.)

Step 1: Represent each of the inputs and outputs in binary.

Sometimes, as in CE1, 3, and 5, the problem statement is already given in terms of binary inputs and outputs. Other times, it is up to the designer. In CE2, we need to create a numeric equivalence for each of the inputs and outputs. We might code the light on as a 1 output and off as 0. (We could just as well have used the opposite definition, as long as we are coordinated with the light designer.) Similarly, we will define a switch in the up position as a 1 input and down as 0. For CE4, the input is a decimal digit. We must determine what BCD code is to be used. That might be provided for us by whoever is providing the input, or we may have the ability to specify it in such a way as to make our system simplest. We must also code the output; we need to know the details of the display and whether a 1 or a 0 lights each segment. (We will discuss those details in Section 2.1.1.) In general, the different input and output representations may result in a significant difference in the amount of logic required.

Step 2: Formalize the design specification either in the form of a *truth table* or of an *algebraic expression*.

We will concentrate on the idea of a truth table here and leave the development of algebraic expressions for later in the chapter. The truth table format is the most common result of step 2 of the design process. We can do this in a digital system because each of the inputs only takes on one of two values (0 or 1). Thus, if we have n inputs, there are 2^n input combinations and thus the truth table has 2^n rows. These rows are normally written in binary order (if, for no other reason, than to make sure that we do not leave any out). The truth table has two sets of columns: n input columns, one for each input variable, and m output columns, one for each of the m outputs.

An example of a truth table with two inputs, A and B, and one output, Y, is shown as Table 2.1, where there are two input columns, one output column, and $2^2 = 4$ rows (not including the title row). We will look at truth tables for some of the continuing examples shortly, after presenting the other steps of the design process.

Table 2.1 A two-input truth table.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

FIRST PAGES

Step 1.5: If necessary, break the problem into smaller subproblems.

This step is listed here because sometimes it is possible to do this after having developed the truth table and sometimes, we must really break up the problem before we can even begin to do such a table.

It is not possible to apply most of the design techniques that we will develop to very large problems. Even CE5, the 4-bit adder, has nine inputs and would thus require a truth table of $2^9 = 512$ rows with nine input columns and five output columns. Although we can easily produce the entries for any line of that table, the table would spread over several pages and be very cumbersome. Furthermore, the minimization techniques of this chapter and Chapter 3 would be strained. The problem becomes completely unmanageable if we go to a realistic adder for a computer—say one that adds 32-bit numbers. There the table would be 2^{64} lines long, even without a carry input (approximately 1.84×10^{19}). (That means that if we were to write 1 million lines on each page and put 1 million pages in a book, we would still need over 18 million volumes to list the entire truth table. Or, if we had a computer that could process 1 billion lines of the truth table per second (requiring a supercomputer), it would still take over 584 years to process the whole table.)

Obviously, we have been able to solve such problems. In the case of the adder, we can imitate how we do it by hand, namely, add 1 bit at a time, producing 1 bit of the sum and the carry to the next bit. That is the problem proposed in CE3; it only requires an eight-line truth table. We can build 32 such systems and connect them together.

Also, it is often most economical to take advantage of subsystems that already have been implemented. For example, we can buy the 4-bit adder described in CE5 (on a single integrated circuit chip). We might want to use that as a component in our design. We will examine this part of the design process further in Chapter 4.

Step 3: Simplify the description.

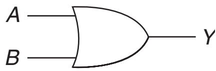
The truth table will lead directly to an implementation in some technologies (see, for example, the ROM in Chapter 4). More often, we must convert that to an algebraic form to implement it. But the algebraic form we get from the truth table tends to lead to rather complex systems. Thus, we will develop techniques for reducing the complexity of algebraic expressions in this chapter and the next.

Step 4: Implement the system with the available components, subject to the design objectives and constraints.

FIRST PAGES

A *gate* is a network with one output. Most of the implementations of this chapter and the next use gates as the components. The truth table used to illustrate step 2 (see Table 2.1) describes the behavior of one type of gate—a two-input OR gate. The final form of the solution may be a block diagram of the gate implementation, where the OR gate is usually depicted by the symbol of Figure 2.1. We may build the system in the laboratory using integrated circuit packages that contain a few such gates or we may simulate it on a computer. We will discuss each of these in more detail later.

Figure 2.1 OR gate symbol.



As mentioned earlier, more complex components, such as adders and decoders, may be available as building blocks, in addition to (or in place of) gates. (Of course, when we get to sequential systems, we will introduce storage devices and other larger building blocks.)

The design objective is often to build the least expensive circuit. That usually corresponds to the simplest algebraic expression, although not always. Since gates are usually obtained in packages (say 4 two-input OR gates in a package), the cost may be measured in terms of the number of packages. Thus, whether we need one of the four gates in a package, or all four, the cost would be the same. Sometimes, one of the objectives is speed, that is, to build as fast a circuit as possible. As we will see later, each time a signal passes through a gate, there is a small delay, slowing down the system. Thus, if speed is a factor, we may have a limit on the number of gates any one signal must pass through.

2.1.1 Don't Care Conditions

Before we can develop the truth table for the display driver example (CE4), we must understand the concept of the *don't care*. In some systems, the value of the output is specified for only some of the input conditions. (Such functions are sometimes referred to as *incompletely specified functions*.) For the remaining input combinations, it does not matter what the output is, that is, we don't care. In a truth table, don't cares are indicated by an X. (Some of the literature uses d, ϕ, or φ.) Table 2.2 is such a truth table.

Table 2.2 A truth table with a don't care.

a	b	f
0	0	0
0	1	1
1	0	1
1	1	X

This table states that the f must be 0 when a and b are 0, that it must be 1 when $a = 0$ and $b = 1$ or when $a = 1$ and $b = 0$, and that it does not matter what f is when a and b are both 1. In other words, either f_1 or f_2 of Table 2.3 are acceptable.

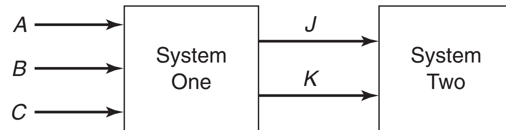
Table 2.3 Acceptable truth tables.

a	b	f_1	f_2
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

When we design a system with don't cares, we may make the output either 0 or 1 for each don't care input combination. In the example of Table 2.3, that means that we can implement either f_1 or f_2. One of these might be much less costly to implement. If there are several don't cares, the number of acceptable solutions greatly increases, since each don't care can be either 0 or 1, independently. The techniques we develop in Chapter 3 handle don't cares very easily; they do not require solving separate problems.

FIRST PAGES

Figure 2.2 Design example with don't cares.



In real systems, don't cares occur in several ways. First, there may be some input combinations that never occur. That is the case in CE4, where the input is the code for a decimal digit; there are only 10 possible input combinations. If a 4-bit code is used, then six of the input combinations never occur. When we build a system, we can design it such that the outputs would be either 0 or 1 for each of these don't care combinations, since that input never happens.

A second place where don't cares occur is in the design of one system to drive a second system. Consider the block diagram of Figure 2.2. We are designing System One to make System Two behave in a certain way. On some occasions, for certain values of *A, B,* and *C,* System Two will behave the same way whether *J* is 0 or 1. In that case, the output *J* of System One is a don't care for that input combination. We will see this behavior arise in Chapter 6, where System Two is a flip flop (a binary storage device).

We will see a third kind of don't care in CE4; we may really not care what one output is.

2.1.2 The Development of Truth Tables

Given a word problem, the first step is to decide how to code the inputs. Then, the development of a truth table is usually rather straightforward. The number of inputs determines the number of rows, and the major problem generally revolves about the ambiguity of English (or any natural language).

For CE1, a 16-row truth table is required. There are four input columns and one output column. (In Table 2.4, three output columns are shown Z_1, Z_2, and Z_3 to account for the three interpretations of the problem statement.) There is little room for controversy on the behavior of the system for the first 15 rows of the table. If there are fewer than three 1's on the input lines, the output is 0. If three of the inputs are 1 and the other is 0, then the output is 1. The only question in completing the table is in relation to the last row. Does "three of the inputs are 1" mean *exactly* three or does it mean at *least* three? If the former is true, then the last line of the truth table is 0, as shown for Z_1. If the latter is true, then the last line of the table is 1, as shown in Z_2. Two other options, both shown as Z_3, are that we know that all four inputs will not be 1 simultaneously, and that we do not care what the output is if all four inputs are 1. In those cases, the last entry is don't care, X.

FIRST PAGES

Table 2.4 Truth table for CE1.

A	*B*	*C*	*D*	Z_1	Z_2	Z_3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	1	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	1	1	1
1	1	0	0	0	0	0
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	0	1	X

For CE2, even after coding the inputs and outputs, we do not have a unique solution to the problem. We will label the switches *a, b,* and *c* (where *a* is the master switch) and use a 1 to represent up (and a 0 for down). The light output is labeled *f* (where a 1 on *f* means that the light is on). When $a = 0$, the light is off (0), no matter what the value of *b* and *c*. The problem statement does not specify the output when $a = 1$; it only specifies what effect a change in the other inputs will have. We still have two possible solutions to this problem. If we assume that switches *b* and *c* in the down position cause the light to be off, then the fifth row of the table (100) will have an output of 0, as shown in Table 2.5a. When one of these switches is up (101, 110), then the light must be on. From either of these states, changing *b* or *c* will either return the system to the 100 input state or move it to state 111; for this, the output is 0.

We could have started with some other fixed value, such as switches *b* and *c* up means that the light is on or that switches *b* and *c* down means that the light is on. Either of these would produce the truth table of Table 2.5b, which is equally acceptable.

We have already developed the truth table for CE3, the 1-bit binary full adder, in Section 1.2.2, Table 1.5 (although we did not refer to it as a truth table at that time).

Although we could easily construct a truth table for CE5, the 4-bit adder, we would need 512 rows. Furthermore, once we had done this, we would still find it nearly impossible to simplify the function by hand (that is, without the aid of a computer). We will defer further discussion of this problem to Chapter 4.

We will now examine the display driver of CE4. The first thing we must do is to choose a code for the decimal digit. That will (obviously)

FIRST PAGES

Table 2.5 Truth tables for CE2.

a	*b*	*c*	*f*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(a)

a	*b*	*c*	*f*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

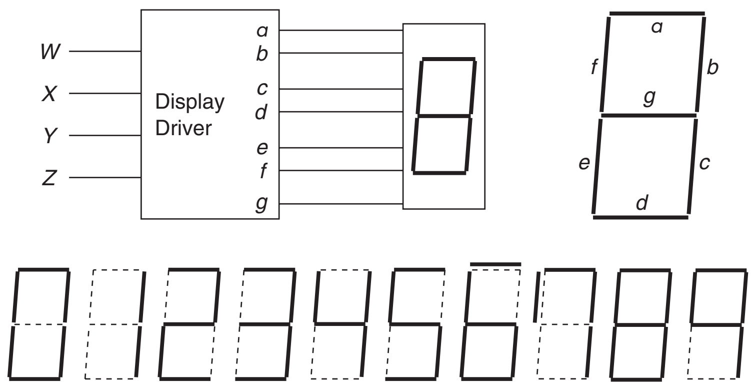
(b)

affect the table and, indeed, make a significant difference in the cost of the implementation. We will call the four binary inputs *W, X, Y,* and *Z* and the seven outputs *a, b, c, d, e, f,* and *g*. For the sake of this example, we will assume that decimal digits are stored in the 8421 code. (We will look at variations on this in Chapter 4.)

The next thing we need to know is whether the display requires a 0 or 1 on each segment input to light that segment. Both types of displays exist. In the solution presented in Table 2.6, we assume that a 1 is needed to light a segment.

The display has seven inputs, labeled *a, b, c, d, e, f, g,* one for each of the segments. A block diagram of the system is shown in Figure 2.3, along with the layout of the display and how each digit is displayed. The solid lines represent segments to be lit and the dashed ones segments that are not lit for that digit. Note that there are alternative displays for the digits 6, 7, and 9. For 6, sometimes segment *a* is lit, and sometimes it is not. The design specification might state that it must be lit or that it must not be lit or that it doesn't matter; choose whatever is easier. The latter is the choice shown in Table 2.6. We will return to this problem in Chapter 4.

Figure 2.3 A seven-segment display.



FIRST PAGES

Table 2.6 A truth table for the seven-segment display driver.

Digit	*W*	*X*	*Y*	*Z*	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	X	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	X	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	X	0	1	1
–	1	0	1	0	X	X	X	X	X	X	X
–	1	0	1	1	X	X	X	X	X	X	X
–	1	1	0	0	X	X	X	X	X	X	X
–	1	1	0	1	X	X	X	X	X	X	X
–	1	1	1	0	X	X	X	X	X	X	X
–	1	1	1	1	X	X	X	X	X	X	X

EXAMPLE 2.1

As a final example, we want to develop a truth table for a system with three inputs, *a, b,* and *c,* and four outputs, *w, x, y, z*. The output is a binary number equal to the largest integer that meets the input conditions:

a = 0: odd	*a* = 1: even
b = 0: prime	*b* = 1: not prime
c = 0: less than 8	*c* = 1: greater than or equal to 8

Some inputs may never occur; the output is never all 0's.
(A prime is a number that is only evenly divisible by itself and 1.) The following is a truth table for this system.

a	*b*	*c*	*w*	*x*	*y*	*z*
0	0	0	0	1	1	1
0	0	1	1	1	0	1
0	1	0	X	X	X	X
0	1	1	1	1	1	1
1	0	0	0	0	1	0
1	0	1	X	X	X	X
1	1	0	0	1	1	0
1	1	1	1	1	1	0

For the first four rows, we are looking for odd numbers. The odd primes are 1, 3, 5, 7, 11, and 13. Thus, the first row is the binary for 7 (the largest odd prime less than 8) and the second row is the binary for 13. The next two rows contain nonprimes. All odd numbers less than 8 are prime; therefore, the input is never 010 and the outputs are don't cares. Finally, 9 and 15 are odd nonprimes; 15 is larger. For the second half of the table, the only even prime is 2; thus, 101 never occurs. The largest even nonprimes are 6 and 14.

[SP 1, 2; EX 1,2]

FIRST PAGES

2.2 SWITCHING ALGEBRA

In the last section, we went from a verbal description of a combinational system to a more formal and exact description—a truth table. Although the truth table is sufficient to implement a system using read-only memory (see Chapter 4), we need an algebraic description to analyze and design systems with other components. In this section, we will develop the properties of switching algebra.

We need the algebra for several reasons. Perhaps the most obvious is that if we are presented with a network of gates, we need to obtain a specification of the output in terms of the input. Since each gate is defined by an algebraic expression, we most often need to be able to manipulate that algebra. (We could try each possible input combination and follow the signals through each gate until we reached the output. That, however, is a very slow approach to creating a whole truth table for a system of gates.)

Second, in the design process, we often obtain an algebraic expression that corresponds to a much more complex network of gates than is necessary. Algebra allows us to simplify that expression, perhaps even minimize the amount of logic needed to implement it. When we move on to Chapter 3, we will see that there are other nonalgebraic ways of doing this minimization, methods that are more algorithmic. However, it is still important to understand the algebraic foundation behind them.

Third, algebra is often indispensable in the process of implementing networks of gates. The simplest algebraic expression, found by one of the techniques presented in this chapter or the next, does not always correspond to the network that satisfies the requirements of the problem. Thus, we may need the algebra to enable us to satisfy the constraints of the problem.

One approach to the development of switching algebra is to begin with a set of postulates or axioms that define the more general Boolean algebra. In Boolean algebra, each variable—inputs, outputs, and internal signals—may take on one of k values (where $k \geq 2$). Based on these postulates, we can define an algebra and eventually determine the meaning of the operators. We can then limit them to the special case of switching algebra, $k = 2$. Rather, we will define switching algebra in terms of its operators and a few basic properties.

2.2.1 Definition of Switching Algebra

Switching algebra is binary, that is, all variables and constants take on one of two values: 0 and 1. Quantities that are not naturally binary must then be coded into binary format. Physically, they may represent a light off or on, a switch up or down, a low voltage or a high one, or a magnetic field in one direction or the other. From the point of view of the algebra, the physical representation does not matter. In the laboratory, we will choose one of the physical manifestations to represent each value.

FIRST PAGES

We will first define the three operators of switching algebra and then develop a number of properties of switching algebra:

OR (written as $+$)*
$a + b$ (read a OR b) is 1 if and only if $a = 1$ **or** $b = 1$ or both
AND (written as $\cdot$ or simply two variables catenated)
$a \cdot b = ab$ (read a AND b) is 1 if and only if $a = 1$ **and** $b = 1$.
NOT (written $'$)
a' (read NOT a) is 1 if and only if $a = 0$.

The term *complement* is sometimes used instead of NOT. The operation is also referred to as inversion, and the device implementing it is called an inverter.

Because the notation for OR is the same as that for addition in ordinary algebra and that for AND is the same as multiplication, the terminology *sum* and *product* is commonly used. Thus, ab is often referred to as a product term and $a + b$ as a sum term. Many of the properties discussed in this chapter apply to ordinary algebra, as well as switching algebra, but, as we will see, there are some notable exceptions.

Truth tables for the three operators are shown in Table 2.7.

Table 2.7 Truth tables for OR, AND, and NOT.

a	b	$a + b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	ab
0	0	0
0	1	0
1	0	0
1	1	1

a	a'
0	1
1	0

We will now begin to develop a set of properties of switching algebra. (These are sometimes referred to as theorems.) A complete list of the properties that we will use may be found inside the front cover.† The first group of properties follow directly from the definitions (or the truth tables).

commutative

P1a. $a + b = b + a$ **P1b.** $ab = ba$

Note that the values for both OR and AND are the same for the second and third lines of the truth table. This is known as the *commutative* property. It seems obvious because it holds for addition and multiplication, which

*OR is sometimes written $\vee$; AND is then written as $\wedge$. NOT x is sometimes written $\sim x$ or $\bar{x}$.

†This list is somewhat arbitrary. We are including those properties that we have found useful in manipulating algebraic expressions. Any pair of expressions that are equal to each other could be included on the list. Indeed, other books have a somewhat different list.

FIRST PAGES

use the same notation. However, it needs to be stated explicitly, because it is not true for all operators in all algebras. (For example, $a - b \neq b - a$ in ordinary algebra. There is no subtraction operation in switching algebra.)

P2a. $a + (b + c) = (a + b) + c$ **P2b.** $a(bc) = (ab)c$

associative

This property, known as the *associative* law, states that the order in which one does the OR or AND operation doesn't matter, and thus we can write just $a + b + c$ and abc (without the parentheses). It also enables us to talk of the OR or AND of several things. We can thus extend the definition of OR to

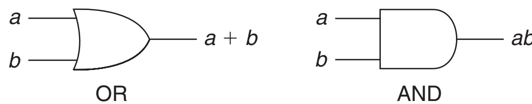
$a + b + c + d + \cdots$ is 1 if any of the operands ($a, b, c, d, \ldots$) is 1 and is 0 only if all are 0

and the definition of AND extends to

$abcd \ldots$ is 1 if all of the operands are 1 and is 0 if any is 0

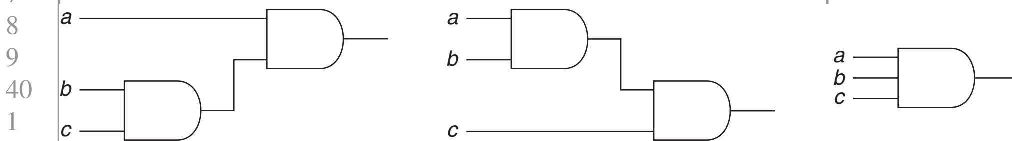
The most basic circuit element is the gate. A *gate* is a circuit with one output that implements one of the basic functions, such as the OR and AND. (We will define additional gate types later.) Gates are available with two inputs, as well as three, four, and eight inputs. (They could be built with other numbers of inputs, but these are the standard commercially available sizes.) The symbols most commonly used (and which we will use throughout this text) are shown in Figure 2.4. (Note in Figure 2.4 the rounded input for the OR and the flat input for the AND; and the pointed output on the OR and the rounded output on the AND.)

Figure 2.4 Symbols for OR and AND gates.



Property 2b states that the three circuits of Figure 2.5 all produce the same output.

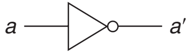
Figure 2.5 AND gate implementation of Property 2b.



The third gate we will include is the NOT, which has the symbol shown in Figure 2.6. The triangle is just the symbol for an amplifier (from electronics). The circle (sometimes referred to as a bubble) on the output is

FIRST PAGES

Figure 2.6 A NOT gate.



the symbol for inversion (NOT) and, as we will see later, is often shown attached to other gate inputs and outputs to indicate the NOT function.

Parentheses are used as in other mathematics; expressions inside the parentheses are evaluated first. When evaluating expressions without parentheses, the order of precedence is

NOT
AND
OR

Thus, for example,

$$ab' + c'd = [a(b')] + [(c')d]$$

Even without parentheses, the input b is complemented first and then ANDed with a. Input c is complemented and ANDed with d and then the two product terms are ORed. If the intent is to AND a and b and then complement them, it must be written $(ab)'$ rather than ab' and if the intent is to do the OR before the ANDs, it must be written $a(b' + c')d$.

In each of the properties, we use a single letter, such as $a, b, c, \ldots$ to represent any expression, not just a single variable. Thus, for example, Property 1a also states that

$$xy'z + w' = w' + xy'z$$

One other thing to note is that properties always appear in *dual* pairs. To obtain the dual of a property, interchange OR and AND, and the constants 0 and 1. The first interchange is obvious in P1 and P2; the other will be used in the next three properties. It can be shown that whenever two expressions are equal, the duals of those expressions are also equal. That could save some work later on, since we do not have to prove both halves of a pair of properties.

[SP 3; EX 3]

2.2.2 Basic Properties of Switching Algebra

We will next look at three pairs of properties associated with the constants 0 and 1.

identity	**P3a.** $a + 0 = a$	**P3b.** $a \cdot 1 = a$
null	**P4a.** $a + 1 = 1$	**P4b.** $a \cdot 0 = 0$
complement	**P5a.** $a + a' = 1$	**P5b.** $a \cdot a' = 0$

Properties 3a and 4b follow directly from the first and third lines of the truth tables; Properties 3b and 4a follow from the second and fourth lines. Property 5 follows from the definition of the NOT, namely, that either a or a' is always 1 and the other is always 0. Thus, P5a must be either $0 + 1$ or $1 + 0$, both of which are 1, and P5b must be either $0 \cdot 1$ or $1 \cdot 0$, both of which are 0. Once again, each of the properties comes in dual pairs.

FIRST PAGES

1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
20
1
2
3
4
5
6
7
8
9
30
1
2
3
4
5
6
7
8
9
40
1
2
3
4
5

Note that by combining the commutative property (P1a) with 3, 4, and 5, we also have

P3aa. $0 + a = a$ **P3bb.** $1 \cdot a = a$

P4aa. $1 + a = 1$ **P4bb.** $0 \cdot a = 0$

P5aa. $a' + a = 1$ **P5bb.** $a' \cdot a = 0$

Often, as we manipulate expressions, we will use one of these versions, rather than first interchanging the terms using the commutative law (P1).

Another property that follows directly from the first and last lines of the truth tables for OR and AND (see Table 2.7) is

idempotency

P6a. $a + a = a$ **P6b.** $a \cdot a = a$

By repeated application of Property 6a, we can see that

$$a + a + a + a = a$$

In the process of manipulating logic functions, it should be understood that each of these equalities is bidirectional. For example, $xyz + xyz$ can be replaced in an expression by $2xyz$; but, also, it is sometimes useful to replace xyz by $xyz + xyz$.

The final property that we will obtain directly from the truth tables of the operators is the only one we will include on our list that is a self-dual.

involution

P7. $(a')' = a$

If $a = 0$, then $a' = 1$. However, when that is complemented again, that is, $(a')' = 1' = 0 = a$. Similarly, if $a = 1$, $a' = 0$ and $(a')' = 1$. Because there are no ANDs, ORs, 0's, or 1's, the dual is the same property.

The next pair of properties, referred to as the *distributive* law, are most useful in algebraic manipulation.

distributive

P8a. $a(b + c) = ab + ac$ **P8b.** $a + bc = (a + b)(a + c)$

P8a looks very familiar; we use it commonly with addition and multiplication. In right to left order, it is referred to as *factoring*. On the other hand, P8b is not a property of regular algebra. (Substitute 1, 2, 3 for *a, b, c,* and the computation is $1 + 6 = 7$ on the left and $4 \times 3 = 12$ on the right.) The simplest way to prove these properties of switching algebra is to produce a truth table for both sides of the equality and show that they are equal. That is shown for Property 8b in Table 2.8. The left three columns are the input columns. The left-hand side (LHS) of the equality is constructed by first forming a column for bc. That column has a 1 in each of the rows where both b and c are 1 and 0 elsewhere. Then LHS = $a + bc$ is computed using the column for a and that for bc. LHS is 1 when either of those columns contains a 1 or both are 1 and is 0 when they are both 0. Similarly, the right-hand side (RHS) is computed by first constructing a column for $a + b$, which contains a 1 when $a = 1$ or $b = 1$.

FIRST PAGES

Table 2.8 Truth table to prove Property 8b.

a	b	c	bc	LHS	$a + b$	$a + c$	RHS
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

The column for $a + c$ is constructed in a similar fashion and finally RHS = $(a + b)(a + c)$ is 1 wherever both of the previous columns are 1.

The table could have been constructed by evaluating each of the expressions for each row (input combination). For the first row,

$$a + bc = 0 + (0 \cdot 0) = 0 + 0 = 0$$

$$(a + b)(a + c) = (0 + 0)(0 + 0) = 0 + 0 = 0$$

and for the sixth row (101)

$$a + bc = 1 + (0 \cdot 1) = 1 + 0 = 1$$

$$(a + b)(a + c) = (1 + 0)(1 + 1) = 1 \cdot 1 = 1$$

We would need to do this for all eight rows. If we need the whole table, the first method usually requires less work.

This method can also be used to determine whether functions are equal. To be equal, the functions must have the same value for all input combinations. If they differ in any row of the truth table, they are not equal.

EXAMPLE 2.2

Construct a truth table and show which of the three functions are equal. (Be sure to state whether they are equal.)

$$f = y'z' + x'y + x'yz'$$
$$g = xy' + x'z' + x'y$$
$$h = (x' + y')(x + y + z')$$

$x\,y\,z$	$y'z'$	$x'y$	$x'yz'$	f	xy'	$x'z'$	$x'y$	g	$x' + y'$	$x + y + z'$	h
0 0 0	1	0	0	1	0	1	0	1	1	1	1
0 0 1	0	0	0	0	0	0	0	0	1	0	0
0 1 0	0	1	1	1	0	1	1	1	1	1	1
0 1 1	0	1	0	1	0	0	1	1	1	1	1
1 0 0	1	0	0	1	1	0	0	1	1	1	1
1 0 1	0	0	0	0	1	0	0	1	1	1	1
1 1 0	0	0	0	0	0	0	0	0	0	1	0
1 1 1	0	0	0	0	0	0	0	0	0	1	0

FIRST PAGES

The truth table was constructed for each of the three functions (using the same technique as we did in developing Table 2.8). For input combination 1 0 1, $f = 0$, but $g = h = 1$. Thus, f is not equal to either of the other functions. The columns for g and h are identical; thus, $g = h$.

2.2.3 Manipulation of Algebraic Functions

[SP 4, 5; EX 4, 5]

Before adding some properties that are useful in simplifying algebraic expressions, it is helpful to introduce some terminology that will make the discussion simpler.

A *literal* is the appearance of a variable or its complement. Examples are a and b'. In determining the complexity of an expression, one of the measures is the number of literals. **Each appearance** of a variable is counted. Thus, for example, the expression

$$ab' + bc'd + a'd + e'$$

contains eight literals.

A *product term* is one or more literals connected by AND operators. In the previous example, there are four product terms, ab', $bc'd$, $a'd$, and e'. Notice that a single literal is a product term.

A *standard product term,* also called a *minterm,* is a product term that includes each variable of the problem, either uncomplemented or complemented. Thus, for a function of four variables, w, x, y, and z, the terms $w'xyz'$ and $wxyz$ are standard product terms, but $wy'z$ is not.

A *sum of products* expression (often abbreviated SOP) is one or more product terms connected by OR operators. The previous expression meets this definition as do each of the following:

$w'xyz' + wx'y'z' + wx'yz + wxyz$ (4 product terms)
$x + w'y + wxy'z$ (3 product terms)
$x' + y + z$ (3 product terms)
wy' (1 product term)
z (1 product term)

It is usually possible to write several different SOP expressions for the same function.

A *canonical sum,* or *sum of standard product terms,* is just a sum of products expression where all of the terms are standard product terms. The first example is the only canonical sum (if there are four variables in all of the problems). Often, the starting point for algebraic manipulations is with canonical sums.

A *minimum sum of products* expression is one of those SOP expressions for a function that has the fewest number of product terms. If there is more than one expression with the fewest number of terms, then minimum is defined as one or more of those expressions with the fewest

FIRST PAGES

number of literals. As implied by this wording, there may be more than one minimum solution to a given problem. Each of the following expressions are equal (meaning that whatever values are chosen for x, y, and z, each expression produces the same value). Note that the first is a sum of standard product terms.

(1) $x'yz' + x'yz + xy'z' + xy'z + xyz$	5 terms, 15 literals
(2) $x'y + xy' + xyz$	3 terms, 7 literals
(3) $x'y + xy' + xz$	3 terms, 6 literals
(4) $x'y + xy' + yz$	3 terms, 6 literals

Expressions (3) and (4) are the minima. (It should be clear that those are minimum among the expressions shown; it is not so obvious that there is not yet another expression with fewer terms or literals.) (A word of caution: When looking for all of the minimum solutions, do **not** include any solution with more terms or more literals than the best already found.)

Actually, we have enough algebra at this point to be able to go from the first expression to the last two. First, we will reduce the first expression to the second:

$$x'yz' + x'yz + xy'z' + xy'z + xyz$$
$$= (x'yz' + x'yz) + (xy'z' + xy'z) + xyz \qquad \text{associative}$$
$$= x'y(z' + z) + xy'(z' + z) + xyz \qquad \text{distributive}$$
$$= x'y \cdot 1 + xy' \cdot 1 + xyz \qquad \text{complement}$$
$$= x'y + xy' + xyz \qquad \text{identity}$$

The first step takes advantage of P2a, which allows us to group terms in any way we wish. We then utilized P8a to factor $x'y$ out of the first two terms and xy' out of the third and fourth terms. Next we used P5aa to replace $z' + z$ by 1. In the final step, we used P3b to reduce the expression.

The last three steps can be combined into a single step. We can add a property

adjacency

P9a. $ab + ab' = a$ **P9b.** $(a + b)(a + b') = a$

where, in the first case, $a = x'y$ and $b = z'$. Thus, if there are two product terms in a sum that are identical, except that one of the variables is uncomplemented in one and complemented in the other, they can be combined, using P9a. (The proof of this property follows the same three steps we used before—P8a to factor out the a, P5a to replace $b + b'$ by 1, and finally P3b to produce the result.) The dual can be proved using the dual steps, P8b, P5b, and P3a.

The easiest way to get to expression (3), that is, to go to six literals, is to use P6a, and make two copies of $xy'z$, that is,

$$xy'z = xy'z + xy'z$$

FIRST PAGES

The expression becomes

$$\begin{aligned} &x'yz' + x'yz + xy'z' + xy'z + xyz + xy'z \\ &\quad = (x'yz' + x'yz) + (xy'z' + xy'z) + (xyz + xy'z) \\ &\quad = x'y(z' + z) + xy'(z' + z) + xz(y + y') \\ &\quad = x'y \cdot 1 + xy' \cdot 1 + xz \cdot 1 \\ &\quad = x'y + xy' + xz \end{aligned}$$

We added the second copy of $xy'z$ at the end and combined it with the last term (xyz). The manipulation then proceeded in the same way as before. The other expression can be obtained in a similar manner by using P6a on $x'yz$ and combining the second copy with xyz. Notice that we freely reordered the terms in the first sum of products expression when we utilized P6a to insert a second copy of one of the terms.

In general, we may be able to combine a term on the list with more than one other term. If that is the case, we can replicate a term as many times as are needed.

Another property that will allow us to reduce the system to six literals without the need to make extra copies of a term is

P10a. $a + a'b = a + b$ **P10b.** $a(a' + b) = ab$ **simplification**

We can demonstrate the validity of P10a by using P8b, P5a, and P3bb as follows:

$$\begin{aligned} a + a'b &= (a + a')(a + b) && \text{distributive} \\ &= 1 \cdot (a + b) && \text{complement} \\ &= a + b && \text{identity} \end{aligned}$$

P10b can be demonstrated as follows:

$$a(a' + b) = aa' + ab = 0 + ab = ab$$

We can apply this property to the example by factoring x out of the last two terms:

$$\begin{aligned} &x'y + xy' + xyz \\ &\quad = x'y + x(y' + yz) && \text{distributive} \\ &\quad = x'y + x(y' + z) && \text{simplification} \\ &\quad = x'y + xy' + xz && \text{distributive} \end{aligned}$$

We used P10a where $a = y'$ and $b = z$ in going from line 2 to 3. Instead, we could have factored y out of the first and last terms, producing

$$\begin{aligned} &y(x' + xz) + xy' \\ &\quad = y(x' + z) + xy' \\ &\quad = x'y + yz + xy' \end{aligned}$$

which is the other six literal equivalent.

FIRST PAGES

Consider the following example, an expression in canonical form.

EXAMPLE 2.3

$a'b'c' + a'bc' + a'bc + ab'c'$

The first two terms can be combined using P9a, producing

$a'c' + a'bc + ab'c'$

Now, we can factor a' from the first two terms and use P10a to reduce this to

$a'c' + a'b + ab'c'$

and repeat the process with c' and the first and last terms, resulting in the expression

$a'c' + a'b + b'c'$

Although this expression is simpler than any of the previous ones, it is not minimum. With the properties we have developed so far, we have reached a dead end, and we have no way of knowing that this is not the minimum. Returning to the original expression, we can group the first term with the last and the middle two terms. Then, when we apply P9a, we get an expression with only two terms and four literals:

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

Later, we will see a property that allows us to go from the three-term expression to the one with only two terms.

Each terminology defined earlier has a dual that will also prove useful.

A *sum term* is one or more literals connected by OR operators. Examples are $a + b' + c$ and b' (just one literal).

A *standard sum term,* also called a *maxterm,* is a sum term that includes each variable of the problem, either uncomplemented or complemented. Thus, for a function of four variables, w, x, y, and z, the terms $w' + x + y + z'$ and $w + x + y + z$ are standard sum terms, but $w + y' + z$ is not.

A *product of sums* expression (POS) is one or more sum terms connected by AND operators. Examples of product of sums expressions:

$(w + x)(w + y)$	2 terms
$w(x + y)$	2 terms
w	1 term
$w + x$	1 term
$(w + x' + y' + z')(w' + x + y + z')$	2 terms

A *canonical product,* or *product of standard sum terms,* is just a POS expression in which all of the terms are standard sum terms. The last example above is the only canonical sum (if there are four variables

FIRST PAGES

in all of the problems). Often, the starting point for algebraic manipulations is with canonical sums.

Minimum is defined the same way for both POS and SOP, namely, the expressions with the fewest number of terms, and, among those with the same number of terms, those with the fewest number of literals. A given function (or expression) can be reduced to minimum sum of products form and to minimum product of sums form. They may both have the same number of terms and literals or either may have fewer than the other. (We will see examples later, when we have further developed our minimization techniques.)

An expression may be in sum of products form, product of sums form, both, or neither. Examples are

SOP: $x'y + xy' + xyz$

POS: $(x + y')(x' + y)(x' + z')$

both: $x' + y + z$ or xyz'

neither: $x(w' + yz)$ or $z' + wx'y + v(xz + w')$

We will now look at an example of the simplification of functions in maxterms form. (Later, we will look at methods of going from sum of products to product of sums and from product of sums to sum of products forms.)

$$g = (w' + x' + y + z')(w' + x' + y + z)(w + x' + y + z')$$

The first two terms can be combined, using P9b, where

$$a = w' + x' + y \quad \text{and} \quad b = z'$$

producing

$$g = (w' + x' + y)(w + x' + y + z')$$

That can most easily be reduced further by using P6b, to create a second copy of the first term, which can be combined with the last term, where

$$a = x' + y + z' \quad \text{and} \quad b = w$$

producing the final answer

$$g = (w' + x' + y)(x' + y + z')$$

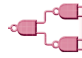
We could also do the following manipulation (parallel to what we did with the SOP expression)

$$\begin{aligned} g &= (w' + x' + y)(w + x' + y + z') \\ &= x' + y + w'(w + z') && \textbf{[P8b]} \\ &= x' + y + w'z' && \textbf{[P10b]} \\ &= (x' + y + w')(x' + y + z') && \textbf{[P8b]} \end{aligned}$$

FIRST PAGES

which, after reordering the literals in the first set of parentheses, is the same expression as before.

[SP 6, 7, 8, 9; EX 6, 7, 8, 9]



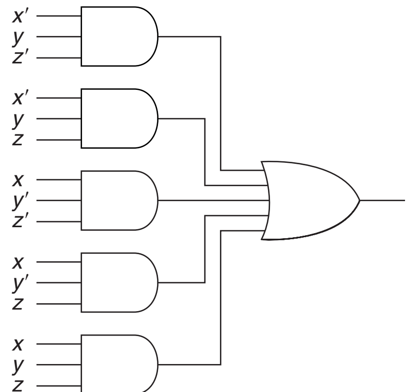
2.3 IMPLEMENTATION OF FUNCTIONS WITH AND, OR, AND NOT GATES

We will first look at the implementation of switching functions using networks of AND, OR, and NOT gates. (After all, the goal of our design is to produce the block diagram of a circuit to implement the given switching function.) When we defined minimum SOP expressions, we introduced, as an example, the function

$$f = x'yz' + x'yz + xy'z' + xy'z + xyz$$

A block diagram of a circuit to implement this is shown in Figure 2.7. Each of the product terms is formed by an AND gate. In this example, all of the AND gates have three inputs. The outputs of the AND gates are used as inputs to an OR (in this case a five-input OR). This implementation assumes that all of the inputs are available, both uncomplemented and complemented (that is, for example, both x and x' are available as inputs). This is usually the case if the input to the combinational logic circuit comes from a flip flop, a storage device in sequential systems. It is not usually true, however, if the input is a system input.

Figure 2.7 Block diagram of f in sum of standard products form.



This is an example of a *two-level* circuit. The number of levels is the maximum number of gates through which a signal must pass from the input to the output. In this example, all signals go first through an AND gate and then through an OR. When inputs are available both uncomplemented and complemented, implementations of both SOP and POS expressions result in two-level circuits.

We saw that this same function can be manipulated to a minimum SOP expression, one version of which is

FIRST PAGES

1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
20
1
2
3
4
5
6
7
8
9
30
1
2
3
4
5
6
7
8
9
40
1
2
3
4
5

$$f = x'y + xy' + xz$$

This, of course, leads to a less complex circuit, namely, the one shown in Figure 2.8.

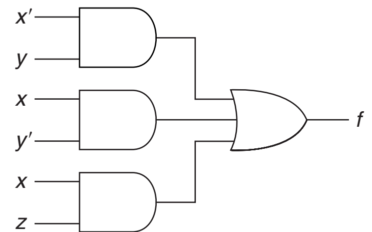
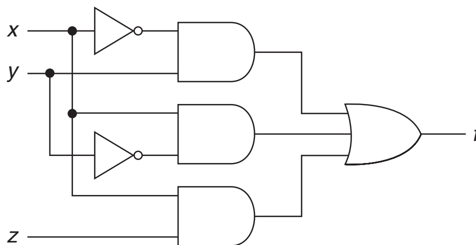


Figure 2.8 Minimum sum of product implementation of *f*.

We have reduced the complexity of the circuit from six gates with 20 gate inputs (three to each of the five ANDs and five to the OR) to one with four gates and 9 gate inputs. The simplest definition of minimum for a gate network is minimum number of gates and, among those with the same number of gates, minimum number of gate inputs. For two-level circuits, this always corresponds to minimum sum of products or minimum product of sums functions.

If complemented inputs are not available, then an inverter (a NOT gate) is needed for each input that is required to be complemented (x and y in this example). The circuit of Figure 2.9 shows the NOT gates that must be added to the circuit of Figure 2.8 to implement f. Note that in this version we showed each input once, with that input line connected to whatever gates required it. That is surely what happens when we actually construct the circuit. However, for clarity, we will draw circuits more like the previous one (except, of course, we will only have one NOT gate for each input, with the output of that gate going to those gates that require it). (This is a three-level circuit because some of the paths pass through three gates: a NOT, an AND, and then an OR.)

Figure 2.9 Circuit with only uncomplemented inputs.



A POS expression (assuming all inputs are available both uncomplemented and complemented) corresponds to a two-level OR–AND network. For this same example, the minimum POS (although that is not obvious based on the algebra we have developed to this point)

$$f = (x + y)(x' + y' + z)$$

is implemented with the circuit of Figure 2.10.

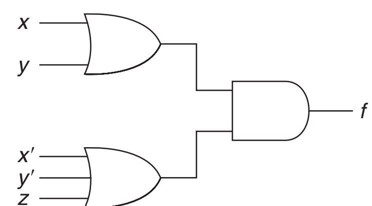


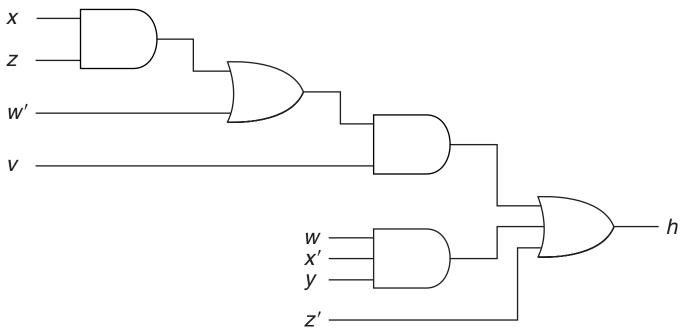
Figure 2.10 A product of sums implementation.

When we implement functions that are in neither SOP nor POS form, the resulting circuits are more than two levels. As an example, consider the following function:

$$h = z' + wx'y + v(xz + w')$$

FIRST PAGES

Figure 2.11 A multilevel circuit.



We begin inside the parentheses and build an AND gate with inputs x and z. The output of that goes to an OR gate, the other input of which is w'. That is ANDed with v, which is ORed with the input z' and the output of the AND gate, producing $wx'y$, which results in the circuit of Figure 2.11.

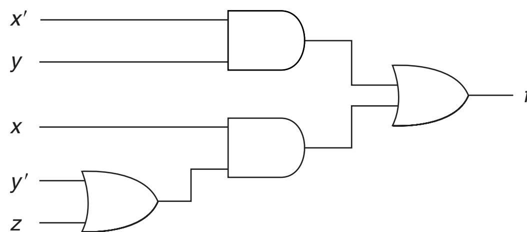
This is a four-level circuit because the signals x and z pass first through an AND gate, then an OR, then an AND, and finally through an OR—a total of four gates.

EXAMPLE 2.4

If we took the version of f used for Figure 2.8, and factored x from the last two terms, we obtain

$$f = x'y + x(y' + z)$$

That would result in the three-level circuit



This (three-level) solution uses 4 two-input gates.

Gates are typically available in dual in-line pin packages (DIPs) of 14 connector pins. These packages are often referred to as *chips*. (Larger packages of 16, 18, 22, and more pins are used for more complex logic.) These packages contain *integrated circuits* (ICs). Integrated circuits are categorized as *small-scale integration* (SSI) when they contain just a few gates. Those are the ones that we will refer to in this chapter. Medium-scale (MSI) circuits contain as many as 100 gates; we will see examples

FIRST PAGES

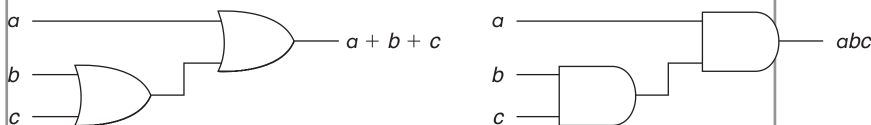
of these later. The terminology *large-scale integration* (LSI), *very large-scale integration* (VLSI), and giga-scale integration (GSI) is used for even more complex packages, including complete computers.

Two of the connector pins are used to provide power to the chip. That leaves 12 pins for logic connections (on a 14-pin chip). Thus, we can fit 4 two-input gates on a chip. (Each gate has two input connections and one output connection. There are enough pins for four such gates.) Similarly, there are enough pins for 6 one-input gates (NOTs), 3 three-input gates, and 2 four-input gates (with two pins unused). In examples that refer to specific integrated circuits, we will discuss *transistor–transistor logic* (TTL) and, in particular, the 7400 series of chips.* For these chips, the power connections are 5 V and ground (0 V).

A list of the common AND, OR, and NOT integrated circuits that might be encountered in the laboratory is

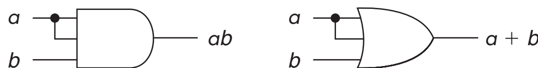
7404	6 (hex) NOT gates
7408	4 (quadruple) two-input AND gates
7411	3 (triple) three-input AND gates
7421	2 four-input (dual) AND gates
7432	4 (quadruple) two-input OR gates

If a three-input OR (or AND) is needed, and only two-input ones are available, it can be constructed as follows:

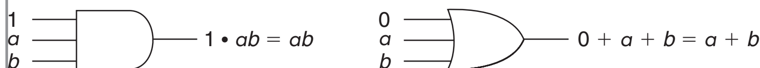


This idea can be extended to gates with larger numbers of inputs.†

Also, if we need a two-input gate and there is a leftover three-input one (because they come three to a package), we can either connect the same signal to two of the inputs (since $aa = a$, and $a + a = a$)



Also, we could connect a logic 1 (+5 V) to one of the inputs of an AND or a logic 0 (ground) to one of the inputs of an OR:



*Even within the 7400 series, there are a number of variations, indicated by a letter or letters after the 74 (such as 74H10). We will not be concerned with that detail; it is left for a course on digital electronics.

†Caution: This approach does not work for NAND and NOR gates (which we will introduce in Section 2.6).

FIRST PAGES

In the laboratory, logic 0 and logic 1 are represented by two voltages: often 0 and 5 V. Most commonly, the higher voltage is used to represent 1 and the lower voltage to represent 0. This is referred to as *positive logic*. The opposite choice is also possible, that is, use the higher voltage to represent 0. That is referred to as *negative logic*. When dealing with 1's and 0's, the concept does not really come up. However, the same electronic circuit has different logic meanings depending on which choice we make.

Consider the truth table of Table 2.9a, where the behavior of the gate is described just in terms of high (H) and low (L). The positive logic interpretation of Table 2.9b produces the truth table for an OR gate. The negative logic interpretation of Table 2.9c is that of an AND gate.

Table 2.9

a. High/Low

a	b	f
L	L	L
L	H	H
H	L	H
H	H	H

b. Positive logic

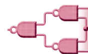
a	b	f
0	0	0
0	1	1
1	0	1
1	1	1

c. Negative logic

a	b	f
1	1	1
1	0	0
0	1	0
0	0	0

Most implementations use positive logic; we will do that consistently throughout this book. Occasionally, negative logic, or even a mixture of the two, is used.

[SP 10, 11; EX 10, 11]



2.4 THE COMPLEMENT

Before we go further, we need to develop one more property. This property is the only one for which a person's name is commonly attached—*DeMorgan's theorem.*

DeMorgan

P11a. $(a + b)' = a'b'$ **P11b.** $(ab)' = a' + b'$

The simplest proof of this property utilizes the truth table of Table 2.10. In Table 2.10, we have produced a column for each of the expressions in the property. (The entries in the table should be obvious because they just

Table 2.10 Proof of DeMorgan's theorem.

a	b	$a + b$	$(a + b)'$	a'	b'	$a'b'$	ab	$(ab)'$	$a' + b'$
0	0	0	1	1	1	1	0	1	1
0	1	1	0	1	0	0	0	1	1
1	0	1	0	0	1	0	0	1	1
1	1	1	0	0	0	0	1	0	0
			11a			11a		11b	11b

FIRST PAGES

involve the AND, OR, and NOT operations on other columns.) Note that the columns (labeled 11a) for $(a + b)'$ and $a'b'$ are the same and those (labeled 11b) for $(ab)'$ and $a' + b'$ are the same.

The property can be extended to more than two operands easily.

P11aa. $(a + b + c \ldots)' = a'b'c' \ldots$

P11bb. $(abc \ldots)' = a' + b' + c' \ldots$

For P11aa, with three variables, the proof goes

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

CAUTION: The most common mistakes in algebraic manipulation involve the misuse of DeMorgan's theorem:

$$(ab)' \neq a'b' \quad \text{rather} \quad (ab)' = a' + b'$$

The NOT (′) cannot be distributed through the parentheses. Just look at the $(ab)'$ and $a'b'$ columns of the truth table and compare the expressions for $a = 0$ and $b = 1$ (or for $a = 1$ and $b = 0$):

$$(0 \cdot 1)' = 0' = 1 \quad 0' \cdot 1' = 1 \cdot 0 = 0$$

The dual of this is also false, that is,

$$(a + b)' \neq a' + b'$$

Once again, the two sides differ when a and b differ.

There will be times when we are given a function and need to find its complement, that is, given $f(w, x, y, z)$, we need $f'(w, x, y, z)$. The straightforward approach is to use DeMorgan's theorem repeatedly.

EXAMPLE 2.5

$$f = wx'y + xy' + wxz$$

then

$$\begin{aligned} f' &= (wx'y + xy' + wxz)' \\ &= (wx'y)'(xy')'(wxz)' && \text{[P11a]} \\ &= (w' + x + y')(x' + y)(w' + x' + z') && \text{[P11b]} \end{aligned}$$

Note that if the function is in SOP form, the complement is in POS form (and the complement of a POS expression is a SOP one).

To find the complement of more general expressions, we can repeatedly apply DeMorgan's theorem or we can follow this set of rules:

1. Complement each variable (that is, a to a' or a' to a).
2. Replace 0 by 1 and 1 by 0.
3. Replace AND by OR and OR by AND, being sure to preserve the order of operations. That sometimes requires additional parentheses.

FIRST PAGES

EXAMPLE 2.6

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

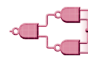
Note that in f, the last operation to be performed is an OR of the complex first term with the product term. To preserve the order, parentheses were needed in f'; making the AND the last operation. We could have used square brackets, [ ], in order to make the expression more readable, making it

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

We would produce the same result, with much more work, by using P11a and P11b over and over again:

$$\begin{aligned} f' &= [ab'(c + d'e) + a'bc']' \\ &= [ab'(c + d'e)]'[a'bc']' \\ &= [a' + b + (c + d'e)'][a + b' + c] \\ &= [a' + b + c'(d'e)'][a + b' + c] \\ &= [a' + b + c'(d + e')][a + b' + c] \end{aligned}$$

[SP 12; EX 12]



2.5 FROM THE TRUTH TABLE TO ALGEBRAIC EXPRESSIONS

Table 2.11 A two-variable truth table.

a	b	f
0	0	0
0	1	1
1	0	1
1	1	1

Often, a design problem is stated in terms of the truth table that describes the output in terms of the inputs. Other times, verbal descriptions of systems can most easily be translated into the truth table. Thus, we need the ability to go from the truth table to an algebraic expression. To understand the process, consider the two-variable truth table of Table 2.11.

Because this is a two-variable problem, the truth table has $4(= 2^2)$ rows, that is, there are 4 possible combinations of inputs. (This is the truth table for the OR as we defined it at the beginning of this chapter, but that is irrelevant to this discussion.) What the table says is that

f is 1 if $a = 0$ AND $b = 1$ OR
if $a = 1$ AND $b = 0$ OR
if $a = 1$ AND $b = 1$

However, this is the same as saying

f is 1 if $a' = 1$ AND $b = 1$ OR
if $a = 1$ AND $b' = 1$ OR
if $a = 1$ AND $b = 1$

But $a' = 1$ AND $b = 1$ is the same as saying $a'b = 1$ and thus

f is 1 if $a'b = 1$ OR if $ab' = 1$ OR if $ab = 1$

FIRST PAGES

1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
20
1
2
3
4
5
6
7
8
9
30
1
2
3
4
5
6
7
8
9
40
1
2
3
4
5

That finally produces the expression

$$f = a'b + ab' + ab$$

Each row of the truth table corresponds to a product term. A SOP expression is formed by ORing those product terms corresponding to rows of the truth table for which the function is 1. Each product term has each variable included, with that variable complemented when the entry in the input column for that variable contains a 0 and uncomplemented when it contains a 1. Thus, for example, row 10 produces the term ab'. These product terms include all of the variables; they are minterms. Minterms are often referred to by number, by just converting the binary number in the input row of the truth table to decimal. Both of the following notations are common:

$$f(a, b) = m_1 + m_2 + m_3$$
$$f(a, b) = \Sigma m(1, 2, 3)$$

We show, in Table 2.12, the minterms and minterm numbers that are used for all functions of three variables.

For a specific function, those terms for which the function is 1 are used to form a SOP expression for f, and those terms for which the function is 0 used to form a SOP expression for f'. We can then complement f' to form a POS expression for f.

Table 2.12 Minterms.

ABC	Minterm	Number
0 0 0	$A'B'C'$	0
0 0 1	$A'B'C$	1
0 1 0	$A'BC'$	2
0 1 1	$A'BC$	3
1 0 0	$AB'C'$	4
1 0 1	$AB'C$	5
1 1 0	ABC'	6
1 1 1	ABC	7

EXAMPLE 2.7

ABC	*f*	*f'*
0 0 0	0	1
0 0 1	1	0
0 1 0	1	0
0 1 1	1	0
1 0 0	1	0
1 0 1	1	0
1 1 0	0	1
1 1 1	0	1

where the truth table shows both the function, f, and its complement, f'. We can write

$$f(A, B, C) = \Sigma m(1, 2, 3, 4, 5)$$
$$= A'B'C + A'BC' + A'BC + AB'C' + AB'C$$

Either from the truth table, or by recognizing that every minterm is included in either f or f', we can then write

$$f'(A, B, C) = \Sigma m(0, 6, 7)$$
$$= A'B'C' + ABC' + ABC$$

FIRST PAGES

The two sum of minterm forms are SOP expressions. In most cases, including this one, the sum of minterms expression is not a minimum sum of products expression. We could reduce f from 5 terms with 15 literals to either of two functions with 3 terms and 6 literals as follows:

$$
\begin{aligned}
f &= A'B'C + A'BC' + A'BC + AB'C' + AB'C \\
&= A'B'C + A'B + AB' \qquad \text{[P9a, P9a]} \\
&= A'C + A'B + AB' \\
&= B'C + A'B + AB'
\end{aligned}
$$

where the final expressions are obtained using P8a and P10a on the first term and either the second or the third. Similarly, we can reduce f' from 3 terms with 9 literals to 2 terms with 5 literals, using P9a:

$$f' = A'B'C' + AB$$

Using P11, we can then obtain the POS expression* for f,

$$f = (f')' = (A + B + C)(A' + B' + C)(A' + B' + C')$$

To find a minimum POS expression, we can either manipulate the previous POS expression (using P9b on the last two terms) to obtain

$$f = (A + B + C)(A' + B')$$

or we could simplify the SOP expression for f' and then use DeMorgan to convert it to a POS expression. Both approaches produce the same result.

In much of the material of Chapter 3, we will specify functions by just listing their minterms (by number). We must, of course, list the variables of the problem as part of that statement. Thus,

$$f(w, x, y, z) = \Sigma m(0, 1, 5, 9, 11, 15)$$

is the simplest way to specify the function

$$f = w'x'y'z' + w'x'y'z + w'xy'z + wx'y'z + wx'yz + wxyz$$

If the function includes don't cares, then those terms are included in a separate sum (Σ).

*It is possible to obtain POS expressions directly from the truth table without first finding the SOP expression. Each 0 of f produces a maxterm in the POS expression. We have omitted that approach here, because it tends to lead to confusion.

FIRST PAGES

1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
20
1
2
3
4
5
6
7
8
9
30
1
2
3
4
5
6
7
8
9
40
1
2
3
4
5

EXAMPLE 2.8

$$f(a, b, c) = \Sigma m(1, 2, 5) + \Sigma d(0, 3)$$

implies that minterms 1, 2, and 5 are included in the function and that 0 and 3 are don't cares, that is the truth table is as follows:

abc	f
0 0 0	X
0 0 1	1
0 1 0	1
0 1 1	X
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	0

Let us now return to the first three of our continuing examples and develop algebraic expressions for them.

EXAMPLE 2.9

Using Z_2 for CE1, we get

$$Z_2 = A'BCD + AB'CD + ABC'D + ABCD' + ABCD$$

directly from the truth table. The last term ($ABCD$) can be combined with each of the others (using P10a). Thus, if we make four copies of it (using P6a repeatedly) and then utilize P10a four times, we obtain

$$Z_2 = BCD + ACD + ABD + ABC$$

No further simplification is possible; this is the minimum sum of products expression. Notice that if we used Z_1, we would have

$$Z_1 = A'BCD + AB'CD + ABC'D + ABCD'$$

No simplification is possible. This expression also has four terms, but it has 16 literals, whereas the expression for Z_2 only has 12.

EXAMPLE 2.10

For CE2, we have either

$$f = ab'c + abc' \quad \text{or} \quad f = ab'c' + abc$$

depending on which truth table we choose. Again, no simplification is possible.

For f', we have (for the first version)

$$\begin{aligned} f' &= a'b'c' + a'b'c + a'bc' + a'bc + ab'c' + abc \\ &= a'b' + a'b + ab'c' + abc && \text{[P9a, P9b]} \\ &= a' + ab'c' + abc = a' + b'c' + bc && \text{[P9a, P10a]} \end{aligned}$$

Thus, the product of maxterms is

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

and the minimum POS is

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

FIRST PAGES

EXAMPLE 2.11

For the full adder, CE3, (using c for the carry in, c_{in}, to simplify the algebraic expressions, we get from the truth table

$$c_{out} = a'bc + ab'c + abc' + abc$$
$$s = a'b'c + a'bc' + ab'c' + abc$$

The simplification of carry out is very much like that of Z_2 in Example 2.9, resulting in

$$c_{out} = bc + ac + ab$$

but s is already in minimum SOP form. We will return to the implementation of the full adder in Section 2.8.

We will next take a brief look at a more general approach to switching functions. How many different functions of n variables are there?

For two variables, there are 16 possible truth tables, resulting in 16 different functions. The truth table of Table 2.13 shows all of these functions. (Each output column of the table corresponds to one of the 16 possible 4-bit binary numbers.)

Table 2.13 All two-variable functions.

a	b	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 2.14 Number of functions of n variables.

Variables	Terms
1	4
2	16
3	256
4	65,536
5	4,294,967,296

[SP 13, 14; EX 13, 14, 15]

Some of the functions are trivial, such as f_0 and f_{15}, and some are really just functions of one of the variables, such as f_3. The set of functions, reduced to minimum SOP form, are

$$f_0 = 0 \qquad f_6 = a'b + ab' \qquad f_{12} = a'$$
$$f_1 = ab \qquad f_7 = a + b \qquad f_{13} = a' + b$$
$$f_2 = ab' \qquad f_8 = a'b' \qquad f_{14} = a' + b'$$
$$f_3 = a \qquad f_9 = a'b' + ab \qquad f_{15} = 1$$
$$f_4 = a'b \qquad f_{10} = b'$$
$$f_5 = b \qquad f_{11} = a + b'$$

For n variables, the truth table has 2^n rows and thus, we can choose any 2^n-bit number for a column. Thus, there are 2^{2^n} different functions of n variables. That number grows very quickly, as can be seen from Table 2.14.

(Thus, we can find a nearly unlimited variety of problems of four or more variables for exercises or tests.)

FIRST PAGES

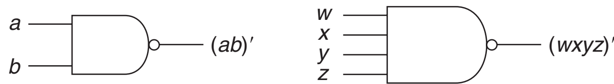


Figure 2.12 NAND gates.

2.6 NAND, NOR, AND EXCLUSIVE-OR GATES

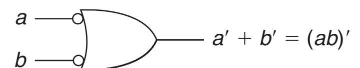
In this section we will introduce three other commonly used types of gates, the NAND, the NOR, and the Exclusive-OR, and see how to implement circuits using them.

The NAND has the symbol shown in Figure 2.12. Like the AND and the OR, the NAND is commercially available in several sizes, typically two-, three-, four-, and eight-input varieties. When first introduced, it was referred to as an AND-NOT, which perfectly describes its function, but the shorter name, NAND, has become widely accepted. Note that DeMorgan's theorem states that

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

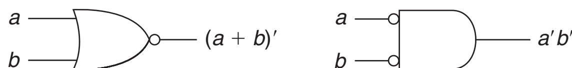
and thus an alternative symbol for the two-input NAND is shown in Figure 2.13. The symbols may be used interchangeably; they refer to the same component.

Figure 2.13 Alternative symbol for NAND.



The NOR gate (OR-NOT) uses the symbols shown in Figure 2.14. Of course, $(a + b)' = a'b'$. NOR gates, too, are available with more inputs.

Figure 2.14 Symbols for NOR gate.



Why use NAND and NOR gates, rather than AND, OR, and NOT gates? After all, the logic expressions are in terms of AND, OR, and NOT operators and thus the implementation with those gates is straightforward. Many electronic implementations naturally invert (complement) signals; thus, the NAND is more convenient to implement than the AND. The most important reason is that with either NAND or NOR, only one type of gate is required. On the other hand, both AND and OR gates are required; and, often, NOT gates are needed, as well. As can be seen from the circuits of Figure 2.15, NOT gates and two-input AND and OR gates can be replaced by just two-input NANDs. Thus, these operators are said to be *functionally complete*. (We could implement gates with more than two inputs using NANDs with more inputs. We could also implement AND, OR, and NOT gates using only NORs; that is left as an exercise.)

Using these gate equivalences, the function $f(= x'y + xy' + xz)$ that we first implemented with AND and OR gates in Figure 2.8 (Section 2.3) can now be implemented with NAND gates, as shown in Figure 2.16.

FIRST PAGES

Figure 2.15 Functional completeness of NAND.

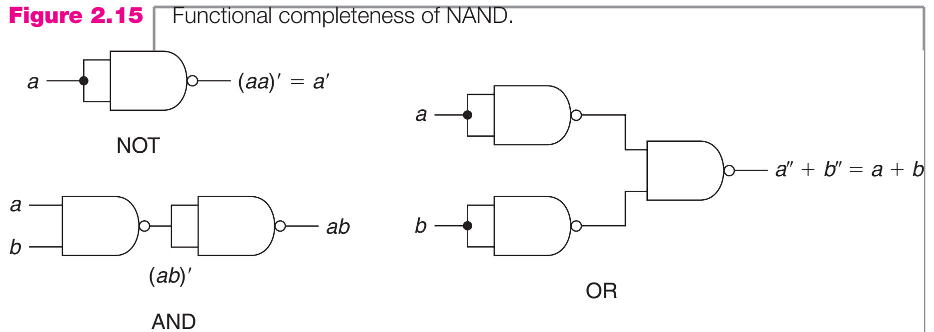


Figure 2.16 NAND gate implementation.

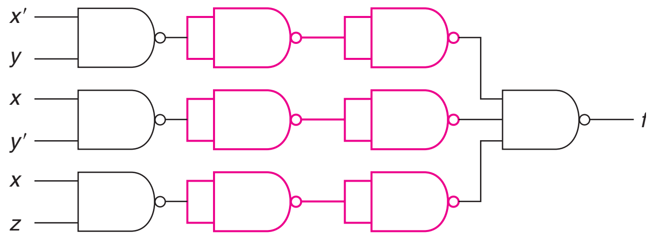
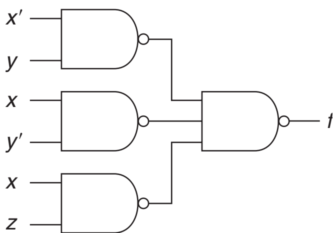


Figure 2.17 Better NAND gate implementation.



But note that we have two NOT gates in a row in each of the green paths. They serve no purpose logically (P7 states $(a')' = a$), and thus they can be removed from the circuit, yielding that of Figure 2.17. That is, all of the AND and OR gates of the original circuit became NANDs. Nothing else was changed.

This process can be greatly simplified when we have a circuit consisting of AND and OR gates such that

1. the output of the circuit comes from an OR,
2. the inputs to all OR gates come either from a system input or from the output of an AND, and
3. the inputs to all AND gates come either from a system input or from the output of an OR.

All gates are replaced by NAND gates, and any input coming directly into an OR is complemented.

We can obtain the same result by starting at the output gate and putting a bubble (a NOT) at both ends of each input line to that OR gate. If the circuit is not two-level, we repeat this process at the input of each of the OR gates. Thus, the AND/OR implementation of f becomes that of Figure 2.18, where all of the gates have become NAND gates (in one of the two notations we introduced earlier).

This approach works with any circuit that meets these conditions, with only one additional step. If an input comes directly into an OR gate,

FIRST PAGES

there is no place for the second NOT; thus, that input must be complemented. For example, the circuit for h

$$h = z' + wx'y + v(xz + w')$$

is shown in Figure 2.19. Again, all of the AND and OR gates become NANDs, but the two inputs that came directly into the OR gates were complemented.

Figure 2.19 A multilevel NAND implementation.

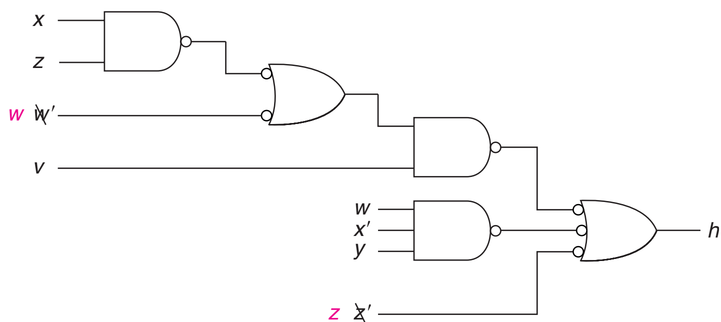
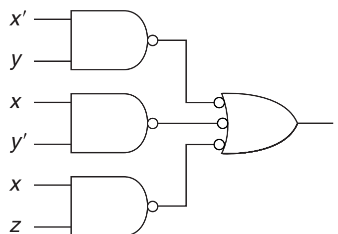


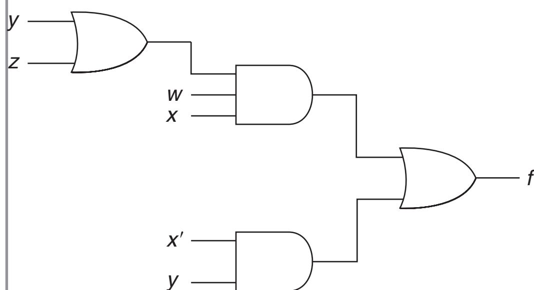
Figure 2.18 Double NOT gate approach.

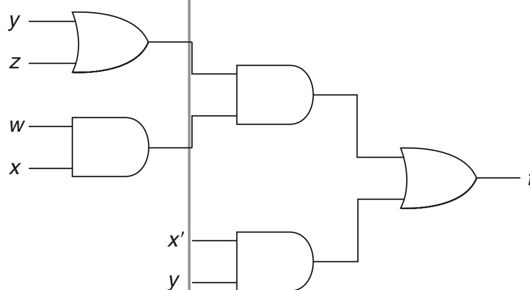


EXAMPLE 2.12

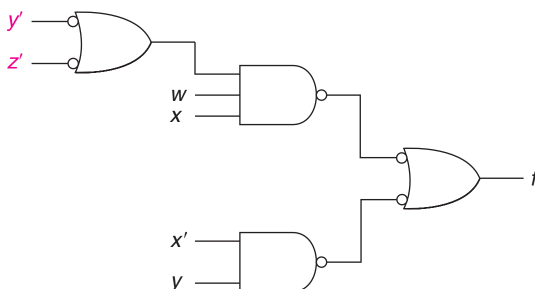
$$f = wx(y + z) + x'y$$

This would be implemented with AND and OR gates in either of two ways.



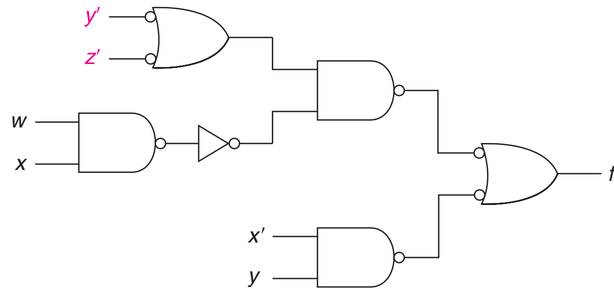


The first version can be directly converted to NAND gates, as follows.



FIRST PAGES

The second version cannot be converted to NAND gates without adding an extra NOT gate, because it violates the third rule—an AND gets an input from another AND. Thus, this circuit would become



where the NOT is required to implement the AND that forms *wx*. Expressions such as this one are often obtained starting from SOP solutions. We will see some examples of this in Section 2.8.

The dual approach works for implementing circuits with NOR gates. When we have a circuit consisting of AND and OR gates such that

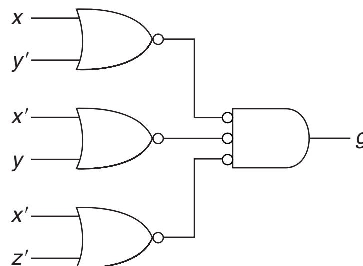
1. the output of the circuit comes from an AND,
2. the inputs to OR gates come either from a system input or from the output of an AND, and
3. the inputs to AND gates come either from a system input or from the output of an OR.

Then all gates can be converted to NOR gates, and, if an input comes directly into an AND gate, that input must be complemented.

EXAMPLE 2.13

$$g = (x + y')(x' + y)(x' + z')$$

is implemented



where all gates are NOR gates.

FIRST PAGES

The Exclusive-OR gate implements the expression

$$a'b + ab'$$

which is sometimes written $a \oplus b$. The terminology comes from the definition that $a \oplus b$ is 1 if $a = 1$ (and $b = 0$) **or** if $b = 1$ (and $a = 0$), but not both $a = 1$ and $b = 1$. The operand we have been referring to as OR (+) is sometimes referred to as the Inclusive-OR to distinguish it from the Exclusive-OR. The logic symbol for the Exclusive-OR is similar to that for the OR except that it has a double line on the input, as shown in Figure 2.20a. Also commonly available is the Exclusive-NOR gate, as shown in Figure 2.20b. It is just an Exclusive-OR with a NOT on the output and produces the function

$$(a \oplus b)' = a'b' + ab.$$

This sometimes is referred to as a comparator, since the Exclusive-NOR is 1 if $a = b$, and is 0 if $a \neq b$.

A NAND gate implementation of the Exclusive-OR is shown in Figure 2.21a, where only uncomplemented inputs are assumed.

The two NOT gates (implemented as two-input NANDs) can be replaced by a single gate, as shown in Figure 2.21b, since

$$a(a' + b') + b(a' + b') = aa' + ab' + ba' + bb' = ab' + a'b$$

Figure 2.20 (a) An Exclusive-OR gate. (b) An Exclusive-NOR gate.

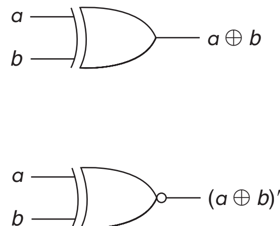
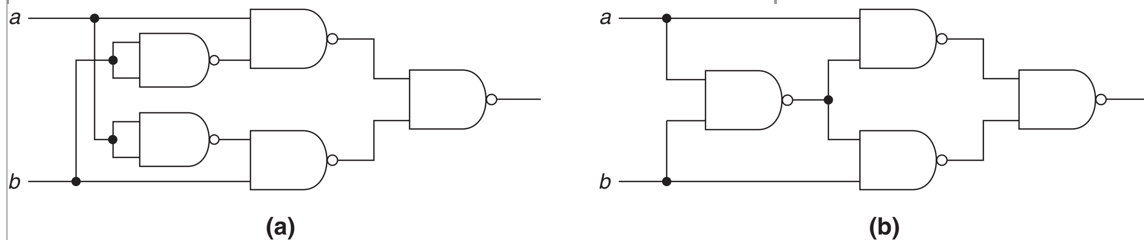


Figure 2.21 Exclusive-OR gates.



Some useful properties of the Exclusive-OR are

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

The Exclusive-OR has both the commutative and associative properties, that is,

$$a \oplus b = b \oplus a$$
$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

FIRST PAGES

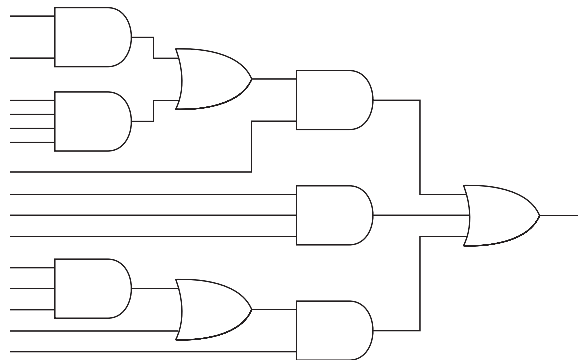
A list of some of the more common NAND, NOR, and Exclusive-OR integrated circuit packages that we may encounter in the laboratory is as follows:

7400	4 (quadruple) two-input NAND gates
7410	3 (triple) three-input NAND gates
7420	2 (dual) four-input NAND gates
7430	1 eight-input NAND gate
7402	4 (quadruple) two-input NOR gates
7427	3 (triple) three-input NOR gates
7486	4 (quadruple) two-input Exclusive-OR gates

To build a circuit, we utilize packages. Even if we only need 1 three-input NAND gate, we must buy a package with three gates on it (a 7410). Recognize, however, that a three-input gate can be used as a two-input gate by connecting two of the inputs together or by connecting one of the inputs to a logic 1.

EXAMPLE 2.14

Consider the following circuit, constructed with ANDs and ORs; the input variables have been omitted because they are irrelevant to the discussion.



The number of gates and packages are shown in the left part of the following table

	Gates		Packs			
Inputs	**AND**	**OR**	**AND**	**OR**	**NAND**	**Packs**
2	3	2	1		5	1
3	2	1	1	1	3	1
4	1		1		1	1
Total	6	3	3	1	9	3

With AND and OR gates, four packages are needed: three ANDs and one OR package (because the 2 two-input OR gates can be constructed with the leftover three-input gates.

FIRST PAGES

If all of the gates are converted to NANDs (and some of the inputs are complemented) the gate and package count is shown in the right part of the table. Only three packages are needed. The second four-input gate on the 7420 would be used as the fifth two-input gate (by tying three of the inputs together).

[SP 15, 16, 17; EX 16, 17, 18]

2.7 SIMPLIFICATION OF ALGEBRAIC EXPRESSIONS

We have already looked at the process of simplifying algebraic expressions, starting with a sum of minterms or a product of maxterms. The primary tools were

P9a. $ab + ab' = a$ **P9b.** $(a + b)(a + b') = a$

P10a. $a + a'b = a + b$ **P10b.** $a(a' + b) = ab$

although many of the other properties were used, particularly,

P6a. $a + a = a$ **P6b.** $a \cdot a = a$

P8a. $a(b + c) = ab + ac$ **P8b.** $a + bc = (a + b)(a + c)$

If the function is stated in other than one of the standard forms, two other properties are useful. First,

P12a. $a + ab = a$ **P12b.** $a(a + b) = a$

absorption

The proof of P12a uses P3b, P8a, P4aa, and P3b (again).

$$a + ab = a \cdot 1 + ab = a(1 + b) = a \cdot 1 = a$$

Remember that we only need to prove one half of the property, because the dual of a property is always true. However, we could have proven P12b using the duals of each of the theorems we used to prove P12a. Instead, we could distribute the a from the left side of P12b, producing

$$a \cdot a + ab = a + ab$$

However, that is just the left side of P12a, which we have already proved is equal to a.

P10a and P12a look very similar; yet we used two very different approaches to demonstrate their validity. In P10a, we did

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

[P8b, P5a, P3bb]

whereas for P12a, we used P3b, P8a, P4aa, and P3b. How did we know not to start the proof of P11a by using P8b to obtain

$$a + ab = (a + a)(a + b) = a(a + b)?$$

Those steps are all valid, but they do not get us anywhere toward showing that these expressions equal a. Similarly, if we started the proof of P10a by using P3b, that is,

$$a + a'b = a \cdot 1 + a'b$$

FIRST PAGES

we also do not get anywhere toward a solution. How does the novice know where to begin? Unfortunately, the answer to that is either trial and error or experience. After solving a number of problems, we can often make the correct guess as to where to start on a new one. If that approach does not work, then we must try another one. This is not much of a problem in trying to demonstrate that two expressions are equal. We know that we can quit when we have worked one side to be the same as the other.

Before proceeding with a number of examples, some comments on the process are in order. There is no algorithm for algebraic simplification, that is, there is no ordered list of properties to apply. On the other hand, of the properties we have up to this point, 12, 9, and 10 are the ones most likely to reduce the number of terms or literals. Another difficulty is that we often do not know when we are finished, that is, what is the minimum. In most of the examples we have worked so far, the final expressions that we obtained appear to be as simple as we can go. However, we will see a number of examples where it is not obvious that there is not a more minimum expression. We will not be able to get around this until Chapter 3 when we develop other simplification methods. (Note that in the Solved Problems and the Exercises, the number of terms and literals in the minimum solution is given. Once that is reached, we know we are done; if we end up with more, we need to try another approach.)

We will now look at several examples of algebraic simplification.

EXAMPLE 2.15

$$xyz + x'y + x'y'$$
$$= xyz + x' \qquad \textbf{[P9a]}$$
$$= x' + yz \qquad \textbf{[P10a]}$$

where $a = x'$, $a' = x$, and $b = yz$

EXAMPLE 2.16

$$wx + wxy + w'yz + w'y'z + w'xyz'$$
$$= (wx + wxy) + (w'yz + w'y'z) + w'xyz'$$
$$= wx + w'z + w'xyz' \qquad \textbf{[P12a, P9a]}$$
$$= wx + w'(z + xyz')$$
$$= wx + w'(z + xy) \qquad \textbf{[P10a]}$$
$$= wx + w'z + w'xy$$
$$= w'z + x(w + w'y)$$
$$= w'z + x(w + y) \qquad \textbf{[P10a]}$$
$$= w'z + wx + xy$$

That property could have been used first with wx (resulting in xyz'). That approach, however, would leave us with an expression

$$w'z + wx + xyz'$$

for which there are no algebraic clues as to how to proceed. The only way we can now reduce it is to add terms to the expression. Shortly, we will introduce another property that will enable us to go from this expression to the minimum one.

FIRST PAGES

EXAMPLE 2.17

$(x + y)(x + y + z') + y' = (x + y) + y'$ **[P12b]**
$= x + (y + y') = x + 1 = 1$ **[P5a, P4a]**

EXAMPLE 2.18

$(a + b' + c)(a + c')(a' + b' + c)(a + c + d)$
$= (b' + c)(a + c')(a + d)$ **[P9b, P10b]**

where the second simplification really took several steps

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

One more tool is useful in the algebraic simplification of switching functions. The operator *consensus* (indicated by the symbol ¢) is defined as follows:

For any two product terms where exactly one variable appears uncomplemented in one and complemented in the other, the consensus is defined as the product of the remaining literals. If no such variable exists or if more than one such variable exists, then the consensus is undefined. If we write one term as at_1 and the second as $a't_2$ (where t_1 and t_2 represent product terms), then, if the consensus is defined,

$at_1 ¢ a't_2 = t_1t_2$

EXAMPLE 2.19

$ab'c ¢ a'd = b'cd$

$ab'c ¢ a'cd = b'cd$

$abc' ¢ bcd' = abd'$

$b'c'd' ¢ b'cd' = b'd'$

$abc' ¢ bc'd =$ undefined—no such variable

$a'bd ¢ ab'cd =$ undefined—two variables, a and b

We then have the following property that is useful in reducing functions.

consensus

P13a. $at_1 + a't_2 + t_1t_2 = at_1 + a't_2$
P13b. $(a + t_1)(a' + t_2)(t_1 + t_2) = (a + t_1)(a' + t_2)$

P13a states that the consensus term is redundant and can be removed from a SOP expression. (Of course, this property, like all of the others, can be used in the other direction to add a term. We will see an example of that shortly.)

CAUTION: It is the consensus term that can be removed (t_1t_2), **not** the other two terms (**not** $at_1 + a't_2$). A similar kind of simplification can be obtained in POS expressions using the dual (P13b). We will not pursue that further.

First, we will derive this property from the others. Using P12a twice, the right-hand side becomes

$at_1 + a't_2 = (at_1 + at_1t_2) + (a't_2 + a't_1t_2)$ **[P12a]**
$= at_1 + a't_2 + (at_1t_2 + a't_1t_2)$
$= at_1 + a't_2 + t_1t_2$ **[P9a]**

FIRST PAGES

It is also useful to look at the truth table for this theorem. From Table 2.15, we see that the consensus term, t_1t_2, is 1 only when one of the other terms is already 1. Thus, if we OR that term with RHS, it does not change anything, that is, LHS is the same as RHS.

Table 2.15 Consensus.

a	t_1	t_2	at_1	$a't_2$	RHS	t_1t_2	LHS
0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	1
0	1	0	0	0	0	0	0
0	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	1	0	1	0	1
1	1	1	1	0	1	1	1

EXAMPLE 2.20

In Example 2.3 (Section 2.2.3), we reduced the function as

$$f = a'b'c' + a'bc' + a'bc + ab'c'$$

to

$$f_1 = a'c' + a'b + b'c'$$

by combining the first two terms using P9a, and then applying P10a twice. At that point, we were at a dead end. However, we found by starting over with a different grouping that we could reduce this to

$$f_2 = b'c' + a'b$$

Indeed, the term eliminated, $a'c'$, is the consensus of the other terms; we could use P13a to go from f_1 to f_2.

EXAMPLE 2.21

$$g = bc' + abd + acd$$

Because Properties 1 through 12 produce no simplification, we now try consensus. The only consensus term defined is

$$bc' ¢ acd = abd$$

Property 13 now allows us to remove the consensus term. Thus,

$$g = bc' + acd$$

With the following function, there is no way to apply Properties 12, 9 and 10:

$$f = w'y' + w'xz + wxy + wyz'$$

Next, we try consensus. An approach that ensures that we try to find the consensus of all pairs of terms is to start with consensus of the second

FIRST PAGES

term with the first; then try the third with the second and the first; and so forth. Following this approach (or any other) for this example, the only consensus that exists is

$$w'xz \text{ ¢ } wxy = xyz$$

When a consensus term was part of the SOP expression, P13a allowed us to remove that term and thus simplify the expression. If the consensus term is not one of the terms in the SOP expression, the same property allows us to add it to the expression. Of course, we don't add another term automatically because that makes the expression less minimum. However, we should keep track of such a term, and, as a last resort, consider adding it to the function. Then, see if that term can be used to form other consensus terms and thus reduce the function. In this example, by adding xyz, f becomes

$$f = w'y' + w'xz + wxy + wyz' + xyz$$

Now, however,

$$xyz \text{ ¢ } wyz' = wxy \quad \text{and} \quad xyz \text{ ¢ } w'y' = w'xz$$

Thus, we can remove both wxy and $w'xz$, leaving

$$f = w'y' + wyz' + xyz \quad \text{(3 terms, 8 literals)}$$

We will now consider an example making use of consensus, as well as all of the other properties. The usual approach is to try to utilize properties 12, 9, and then 10. When we get as far as we can with these, we then turn to consensus.

EXAMPLE 2.22

$$A'BCD + A'BC'D + B'EF + CDE'G + A'DEF + A'B'EF$$
$$= A'BD + B'EF + CDE'G + A'DEF \qquad \text{[P12a, P9a]}$$

But $A'BD$ ¢ $B'EF = A'DEF$ and this reduces to

$$A'BD + B'EF + CDE'G$$

[SP 18; EX 19, 20, 21]

2.8 MANIPULATION OF ALGEBRAIC FUNCTIONS AND NAND GATE IMPLEMENTATIONS

In addition to the need to minimize algebraic expressions, there is sometimes the requirement to put an expression in a certain format, such as SOP, sum of minterms, POS, or product of maxterms. Secondly, to meet design constraints, we sometimes must manipulate the algebra. In this section we will look at some examples and introduce one more property.

If we have a SOP expression and need to expand it to sum of minterms, we have two options. First, we can create a truth table, and, from that, follow the approach of Section 2.5 to produce a sum of minterms. Indeed, this

FIRST PAGES

approach will work for an expression in any format. The other approach is to use P9a to add variables to a term.

EXAMPLE 2.23

$$f = bc + ac + ab$$
$$= bca + bca' + ac + ab$$

We can repeat the process on the other two terms, producing

$$f = bca + bca' + acb + acb' + abc + abc'$$
$$= abc + a'bc + abc + ab'c + abc + abc'$$
$$= a'bc + ab'c + abc' + abc$$

where P6a was used to remove the duplicate terms.

If two literals were missing from a term, that term would produce four minterms, using P9a repeatedly.

EXAMPLE 2.24

$$g = x' + xyz = x'y + x'y' + xyz$$
$$= x'yz + x'yz' + x'y'z + x'y'z' + xyz$$
$$g(x, y, z) = \Sigma m(3, 2, 1, 0, 7) = \Sigma m(0, 1, 2, 3, 7)$$

since minterm numbers are usually written in numeric order.

To convert to product of maxterms, P9b can be used. For example,

EXAMPLE 2.25

$$f = (A + B + C)(A' + B')$$
$$= (A + B + C)(A' + B' + C)(A' + B' + C')$$

One other property is useful in manipulating functions from one form to another.

P14a. $ab + a'c = (a + c)(a' + b)$

(The dual of this is also true; but it is the same property with the variables b and c interchanged.) This property can be demonstrated by first applying P8a to the right side three times:

$$(a + c)(a' + b) = (a + c)a' + (a + c)\,b = aa' + a'c + ab + bc$$

However, $aa' = 0$ and $bc = a'c$ ¢ ab and thus, using P3aa and P13a, we get

$$aa' + a'c + ab + bc = a'c + ab$$

which is equal to the left side of the property.

This property is particularly useful in converting POS expressions to SOP and vice versa.

FIRST PAGES

In Example 2.7, we found the sum of minterms and the minimum SOP expressions, as well as the product of maxterms and the minimum POS expression for

$$f(A, B, C) = \Sigma m\,(1, 2, 3, 4, 5)$$

In Example 2.26, we will start with the minimum POS, and use Property 14 to convert it to a SOP.

EXAMPLE 2.26

$$f = (A + B + C)(A' + B') = AB' + A'(B + C) = AB' + A'B + A'C$$

where the a of P14a is A, the b is $B + C$, and the c is B'. This, indeed, is one of the SOP solutions we found in Example 2.7 for this problem. Although the utilization of this property does not always produce a minimum SOP expression (as it does in this case), it does produce a simpler expression than we would get just using P8a.

$$\begin{aligned} f &= AA' + AB' + BA' + BB' + CA' + CB' \\ &= AB' + A'B + A'C + B'C \end{aligned}$$

The term can then be removed because it is the consensus of AB' and $A'C$.

To go from a POS expression (or a more general expression that is neither SOP nor POS) to a SOP expression, we use primarily the following three properties:

P8b. $a + bc = (a + b)(a + c)$

P14a. $ab + a'c = (a + c)(a' + b)$

P8a. $a(b + c) = ab + ac$

We try to apply them in that order, using the first two from right to left.

EXAMPLE 2.27

$$\begin{aligned} &(A + B' + C)(A + B + D)(A' + C' + D') \\ &= [A + (B' + C)(B + D)](A' + C' + D') && \text{[P8b]} \\ &= (A + B'D + BC)(A' + C' + D') && \text{[P14a]} \\ &= A(C' + D') + A'(B'D + BC) && \text{[P14a]} \\ &= AC' + AD' + A'B'D + A'BC && \text{[P8a]} \end{aligned}$$

The dual of these properties can be used to convert to POS as can be seen in Example 2.28.

EXAMPLE 2.28

$$\begin{aligned} &wxy' + xyz + w'x'z' \\ &= x(wy' + yz) + w'x'z' && \text{[P8a]} \\ &= x(y' + z)(y + w) + w'x'z' && \text{[P14a]} \\ &= (x + w'z')[x' + (y' + z)(y + w)] && \text{[P14a]} \\ &= (x + w')(x + z')(x' + y' + z)(x' + y + w) && \text{[P8b]} \end{aligned}$$

FIRST PAGES

Another application of P14a and this type of algebraic manipulation comes when we wish to implement functions using only two-input NAND or NOR gates (or two- and three-input gates). (We will only consider examples of NAND gate implementations.) Consider the following problem.

The following expression is the only minimum SOP expression for the function f. Assume all inputs are available both uncomplemented and complemented. Find a NAND gate circuit that uses only two-input gates. No gate may be used as a NOT gate.*

$$f = ab'c' + a'c'd' + bd$$

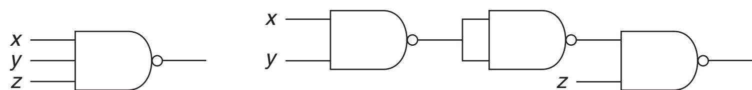
(A two-level solution would require four gates, three of which would be three-input gates, and 11 gate inputs.)

To solve this problem, we must eliminate three-input gates. Thus, the starting point is to attempt to factor something from the three literal terms. In this example, there is a common c' in the first two terms and we can thus obtain

$$f = c'(ab' + a'd') + bd$$

This, indeed, solves the whole problem in one step because not only did we reduce the 2 three-input product terms to two inputs each, but we also got the final OR to a two-input one. Thus, the resulting circuit is shown in Figure 2.22, where we first implemented it with AND and OR gates and then, starting at the output, added double inverters in each path from the input of an OR back to the output of an AND. (In this example, no inputs came directly into an OR.) This solution requires 6 gates and 12 inputs. It should be noted that either solution, this one or the two-level one mentioned earlier requires two integrated circuit packages. This requires two 7400s (4 two-input NANDs each) and would leave two of the gates unused. The two-level solution would require a 7410 (3 three-input gates) and a 7400 for the remaining two-input gate and would leave three of those gates unused. (If we had replaced each three-input gate by 2 two-input ones plus a NOT, the implementation would require 7 two-input gates plus three NOT gates.)

*We could always produce a circuit using two-input gates by replacing a three-input gate by 2 twos and a NOT. For example, a three-input NAND could be implemented as follows:

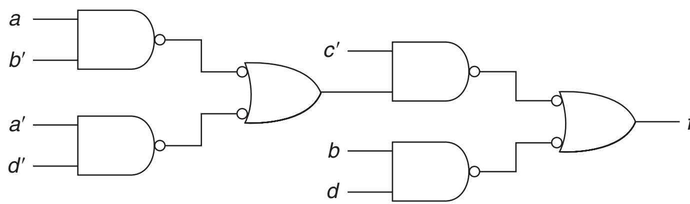


Larger gates could be replaced in a similar fashion. But this approach almost always leads to circuits with more gates than is necessary.

FIRST PAGES

1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
20
1
2
3
4
5
6
7
8
9
30
1
2
3
4
5
6
7
8
9
40
1
2
3
4
5

Figure 2.22 A two-input NAND gate circuit.



More complex examples of finding a two-input gate implementation often require the use of P14a as well as P8a. Consider the function in Example 2.29 (already in minimum sum of products form).

EXAMPLE 2.29

$$G = DE' + A'B'C' + CD'E + ABC'E$$

The four-literal product term is the first place we must attack. We could factor E from the last two terms. That would produce

$$G = DE' + A'B'C' + E(CD' + ABC')$$

But now, there is no way of eliminating the three-input gate corresponding to $A'B'C'$. Instead, we can factor C' from the second and the fourth terms, producing

$$G = C'(A'B' + ABE) + DE' + CD'E$$

We can apply P14a to the expression within the parentheses to get

$$G = C'(A' + BE)(A + B') + DE' + CD'E$$

or, using B instead of A,

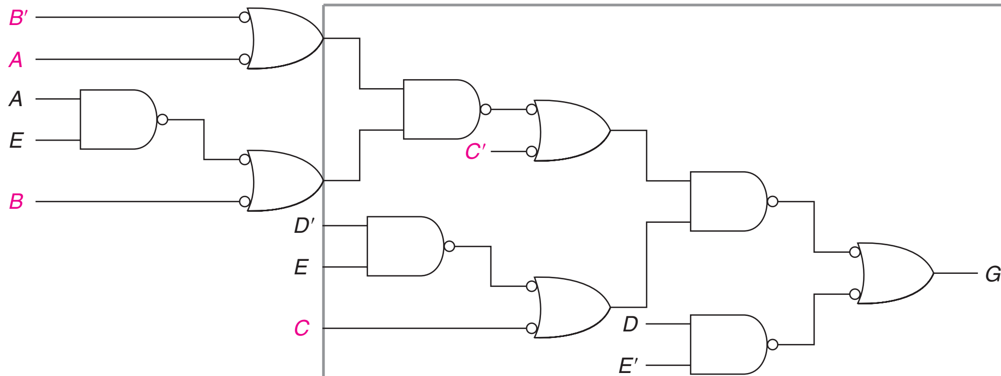
$$G = C'(B' + AE)(B + A') + DE' + CD'E$$

In either case, we still have 2 three-input AND terms, that first product and the last one. (We cannot take the output of the OR gate that forms $B' + AE$ and the output of the OR gate that forms $B + A'$ and connect them to a two-input AND gate. We would then need to connect the output of that AND gate to the input of another AND gate with C' as its other input. This would violate the third rule for conversion to NAND gates—the inputs to AND gates may not come from the output of another AND gate.) We can reduce it to all two-input gates by applying P14a again, using the C' from the first complex term and the C from the last product term, producing (from the second version) the following expression:

$$G = (C' + D'E)[C + (B' + AE)(B + A')] + DE'$$

This requires 10 gates, as shown in the following NAND gate circuit.

FIRST PAGES



Again, we began by implementing the circuit with ANDs and ORs, starting at the inner most parentheses. Five of the inputs went directly to OR gates and were thus complemented (as shown in green in the circuit).

There is still another approach to manipulating this algebra.

$$
\begin{aligned}
G &= C'(A' + BE)(A + B') + DE' + CD'E \\
&= C'(A' + BE)(A + B') + (D + CE)(D' + E') \\
&= (A' + BE)(AC' + B'C') + (D + CE)(D' + E')
\end{aligned}
$$

In this case, we eliminated the three-input AND by distributing the C' (P8a) and used P14a on the last two product terms. We will leave the implementation of this as an exercise, but we can count 11 gates (one more than before) from the algebraic expression, as seen from the following count.

$$G = (A' + BE)(AC' + B'C') + (D + CE)(D' + E')$$

1 2 3 4 5 6 7 8 9 10 11

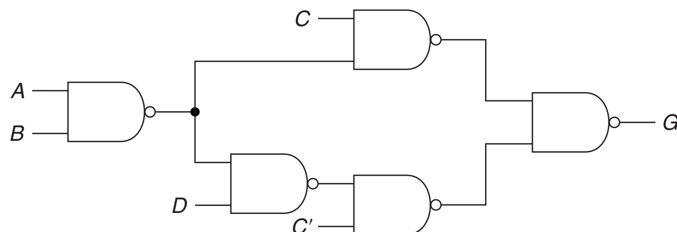
where each gate is numbered below the operator corresponding to that gate.

As an example of sharing a gate, consider the implementation of the following function with two-input NAND gates:

EXAMPLE 2.30

$$
\begin{aligned}
G &= C'D' + ABC' + A'C + B'C \\
&= C'(D' + AB) + C(A' + B')
\end{aligned}
$$

The circuit for that expression is shown next.



FIRST PAGES

Note that only one NAND gate is needed for the product term AB and for the sum term $A' + B'$ (because inputs coming directly to an OR are complemented).

As a final example, we will return to the implementation of the full adder (CE3). The SOP expressions developed in Example 2.11 are repeated (where the carry input, c_{in}, is represented by just c).

EXAMPLE 2.31

$$s = a'b'c + a'bc' + ab'c' + abc$$
$$c_{out} = bc + ac + ab$$

A two-level implementation of these would require 1 four-input NAND gate (for s), 5 three-input NAND gates (four for s and one for c_{out}), and 3 two-input NANDs (for c_{out}), assuming all inputs are available both uncomplemented and complemented. But this assumption is surely not valid for c because that is just the output of combinational logic just like this (from the next less significant bit of the sum). Thus, we need at least one NOT gate (for c') and possibly three (one for each input). The implementation of this adder would thus require four integrated circuit packages (one 7420, two 7410s, and one 7400). (There would be one gate left over of each size, which could be used to create whatever NOTs are needed.)

Although s and c_{out} are in minimum SOP form, we can manipulate the algebra to reduce the gate requirements by first factoring c from two terms of s and from two terms of c_{out}, and factoring c' from the other two terms of s, yielding*

$$s = c(a'b' + ab) + c'(ab' + a'b)$$
$$c_{out} = c(a + b) + ab$$

This requires 11 two-input NAND gates, not including the three NOTs (because ab need only be implemented once for the two terms and $a + b$ is implemented using the same gate as $a'b'$).

Returning to the expression for sum, note that

$$s = c(a \oplus b)' + c'(a \oplus b) = c \oplus (a \oplus b)$$

Furthermore, we could write

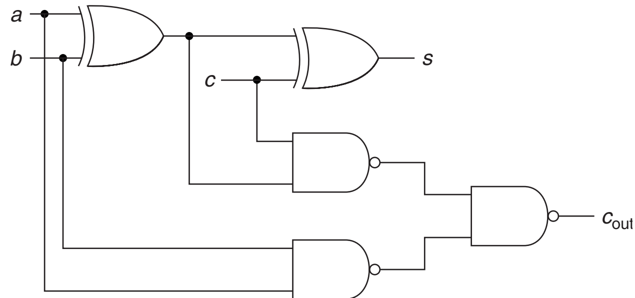
$$c_{out} = c(a \oplus b) + ab$$

(That is a little algebraic trick that is not obvious from any of the properties. However, the difference between $a + b$ and $a \oplus b$ is that the former is 1 when both a and b are 1, but the latter is not. But the expression for c_{out} is 1 for $a = b = 1$ because of the ab term.)

*We could just as easily factor b and b' or a and a' from these expressions; the resulting circuits would have the same layout of gates.

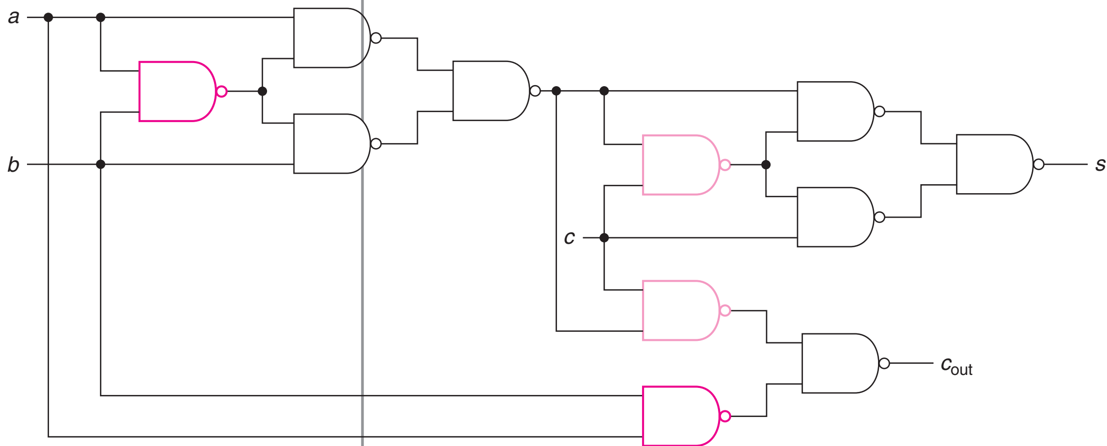
FIRST PAGES

Using these last two expressions, we could implement both the sum and carry using two Exclusive-ORs and 3 two-input NANDs as follows:



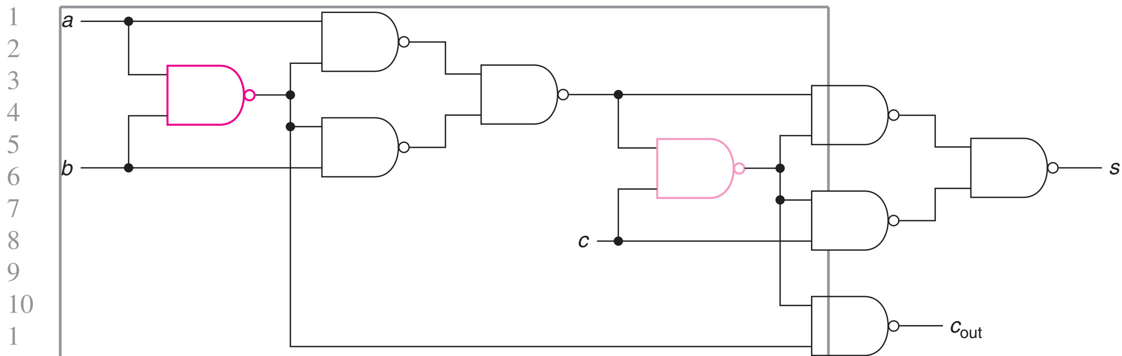
Packages with four Exclusive-OR gates are available (7486) and thus this circuit could be implemented with one of those packages and one 7400. Note that complemented inputs are not necessary for this implementation.

Finally, because we can implement each Exclusive-OR with 4 two-input NAND gates, without requiring complemented inputs, we obtain



Note that the two green NAND gates have the same inputs and the two light green ones also have the same inputs. Only one copy of each is necessary, yielding the final circuit with only nine NAND gates.

FIRST PAGES



This implementation would require three 7400s if we were only building one bit of an adder. However, a 4-bit adder could be built with nine packages (36 two-input gates).

[SP 19, 20, 21, 22, 23; EX 22, 23, 24, 25, 26]

2.9 SOLVED PROBLEMS

1. For each of the following problems, there are four inputs, *A, B, C,* and *D*. Show a truth table for the functions specified. (One truth table with four outputs is shown for the four examples.)

a. The inputs represent a 4-bit unsigned binary number. The output, *W*, is 1 if and only if the number is a multiple of 2 or of 3 but not both.

b. The inputs represent a 4-bit positive binary number. The output, *X*, is 0 if and only if the input is a prime (where 0 never occurs).

c. The first two inputs (*A, B*) represent a 2-bit unsigned binary number (in the range 0 to 3). The last two (*C, D*) represent a second unsigned binary number (in the same range). The output, *Y*, is 1 if and only if the two numbers differ by two or more.

d. The inputs represent a BCD number in Excess-3 code. Those combinations that do not represent one of the digits never occur. The output, *Z*, is 1 if and only if that number is a perfect square.

FIRST PAGES

The truth table contains the answer to all four parts.

A	*B*	*C*	*D*	*W*	*X*	*Y*	*Z*
0	0	0	0	0	X	0	X
0	0	0	1	0	0	0	X
0	0	1	0	1	0	1	X
0	0	1	1	1	0	1	1
0	1	0	0	1	1	0	1
0	1	0	1	0	0	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	0	1	1
1	0	0	0	1	1	1	0
1	0	0	1	1	1	0	0
1	0	1	0	1	1	0	0
1	0	1	1	0	0	0	0
1	1	0	0	0	1	1	1
1	1	0	1	0	0	1	X
1	1	1	0	1	1	0	X
1	1	1	1	1	1	0	X

a. We don't care whether one considers 0 a multiple of 2 or 3 because it is either a multiple of neither or of both. In both cases, $W = 0$. For the next row, 1 is not a multiple of either 2 or 3; thus, $W = 0$. For the next three rows $W = 1$ because 2 and 4 are multiples of 2, but not 3, and 3 is a multiple of 3, but not 2. Both 5 and 7 are multiples of neither and 6 is a multiple of both; thus, for the next three rows, $W = 0$.

b. A prime number is one that is evenly divisible only by 1 or itself. Note that the problem specifies that the output is 0 for primes and is thus 1 for numbers that are not prime. The first nonprime is 4 (2×2). Indeed, all of the even numbers (other than 2) are nonprimes. Because 0 never occurs, the output for the first row is a don't care.

c. For the first four rows, the first number is 0. It is compared on successive rows with 0, 1, 2, and 3. Only 2 and 3 differ from 0 by 2 or more. In the next group of four rows, the first number is 1; it only differs from 3 by 2 or more. In the next four rows, 2 differs only from 0 by 2 or more. Finally, in the last 4 rows, 3 differs from 0 and 1 by 2 or more.

d. A perfect square is an integer obtained by multiplying some integer by itself. Thus, 0, 1, 4, and 9 are perfect squares. Note that the first three rows and the last three rows are all don't cares because those input combinations never occur.

2. The system is a speed warning device. It receives, on two lines, an indication of the speed limit on the highway. There are three possible values: 45, 55, or 65 mph. It receives from the automobile, on

FIRST PAGES

two other lines, an indication of the speed of the vehicle. There are four possible values: under 45, between 46 and 55, between 56 and 65, and over 65 mph. It produces two outputs. The first, *f,* indicates whether the car is going above the speed limit. The second, *g,* indicates that the car is driving at a "dangerous speed"—defined as either over 65 mph or more than 10 mph above the speed limit. Show how each of the inputs and outputs are coded (in terms of binary values) and complete the truth table for this system.

The first step is to code the inputs, as shown in the following tables.

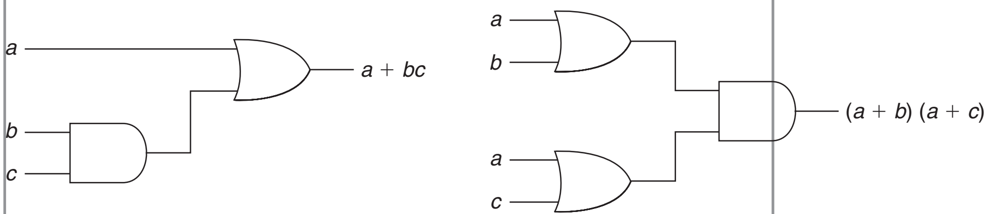
Speed limit	*a*	*b*
45	0	0
55	0	1
65	1	0
unused	1	1

Speed	*c*	*d*
<45	0	0
46–55	0	1
56–65	1	0
>65	1	1

The outputs will be 1 if the car is speeding or driving dangerously.

	a	*b*	*c*	*d*	*f*	*g*
	0	0	0	0	0	0
45	0	0	0	1	1	0
	0	0	1	0	1	1
	0	0	1	1	1	1
	0	1	0	0	0	0
55	0	1	0	1	0	0
	0	1	1	0	1	0
	0	1	1	1	1	1
	1	0	0	0	0	0
65	1	0	0	1	0	0
	1	0	1	0	0	0
	1	0	1	1	1	1
	1	1	0	0	X	X
	1	1	0	1	X	X
	1	1	1	0	X	X
	1	1	1	1	X	X

3. Show a block diagram of a circuit using AND and OR gates for each side of P8b: $a + bc = (a + b)(a + c)$



FIRST PAGES

4. Show a truth table for the following functions:

a. $F = XY' + YZ + X'Y'Z'$

b. $G = X'Y + (X + Z')(Y + Z)$

(a)

XYZ	*XY′*	*YZ*	*X′Y′Z′*	*F*
000	0	0	1	1
001	0	0	0	0
010	0	0	0	0
011	0	1	0	1
100	1	0	0	1
101	1	0	0	1
110	0	0	0	0
111	0	1	0	1

(b)

XYZ	*X′Y*	*X + Z′*	*Y + Z*	() ()	*G*
000	0	1	0	0	0
001	0	0	1	0	0
010	1	1	1	1	1
011	1	0	1	0	1
100	0	1	0	0	0
101	0	1	1	1	1
110	0	1	1	1	1
111	0	1	1	1	1

5. Determine, using truth tables, whether or not each of the groups of expressions are equal:

a. $f = a'c' + a'b + ac$

$g = bc + ac + a'c'$

b. $f = P'Q' + PR + Q'R$

$g = Q' + PQR$

(a)

abc	*a′c′*	*a′b*	*ac*	*f*	*bc*	*ac*	*a′c′*	*g*
000	1	0	0	1	0	0	1	1
001	0	0	0	0	0	0	0	0
010	1	1	0	1	0	0	1	1
011	0	1	0	1	1	0	0	1
100	0	0	0	0	0	0	0	0
101	0	0	1	1	0	1	0	1
110	0	0	0	0	0	0	0	0
111	0	0	1	1	1	1	0	1

The two functions are equal.

(b)

PQR	*P′Q′*	*PR*	*Q′R*	*f*	*Q′*	*PQR*	*g*
000	1	0	0	1	1	0	1
001	1	0	1	1	1	0	1
010	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0
100	0	0	0	0	1	0	1 ←
101	0	1	1	1	1	0	1
110	0	0	0	0	0	0	0
111	0	1	0	1	0	1	1

FIRST PAGES

Note that for row 100 (marked with a green arrow), $f = 0$ and $g = 1$. Thus, the two functions are different.

6. For each of the following expressions, indicate which (if any) of the following apply (more than one may apply):

i. Product term

ii. Sum of products expression

iii. Sum term

iv. Product of sums expression

a. ab'

b. $a'b + ad$

c. $(a + b)(c + a'd)$

d. $a' + b'$

e. $(a + b')(b + c)(a' + c + d)$

a. i. product of two literals

ii. sum of one product term

iv. product of two sum terms

b. ii. sum of two product terms

c. none; second term is not a sum term

d. ii. sum of two product terms

iii. sum of two literals

iv. product of one sum term

e. iv. product of three sum terms

7. In the expressions of problem 6, how many literals are in each?

a. 2 **b.** 4 **c.** 5 **d.** 2 **e.** 7

8. Using Properties 1 to 10, reduce the following expressions to a minimum SOP form. Show each step (number of terms and number of literals in minimum shown in parentheses).

a. $xyz' + xyz$ (1 term, 2 literals)

b. $x'y'z' + x'y'z + x'yz + xy'z + xyz$ (2 terms, 3 literals)

c. $f = abc' + ab'c + a'bc + abc$ (3 terms, 6 literals)

a. $xyz' + xyz = xy(z' + z) = xy \cdot 1 = xy$ **[P8a, P5aa, P3b]**

or, in one step, using P9a, where $a = xy$ and $b = z'$

b. $x'y'z' + x'y'z + x'yz + xy'z + xyz$

FIRST PAGES

Make two copies of $x'y'z$

$= (x'y'z' + x'y'z) + (x'y'z + x'yz) + (xy'z + xyz)$ **[P6a]**

$= x'y'(z' + z) + x'z(y' + y) + xz(y' + y)$ **[P8a]**

$= x'y' \cdot 1 + x'z \cdot 1 + xz \cdot 1$ **[P5aa]**

$= x'y' + x'z + xz$ **[P3b]**

$= x'y' + (x' + x)z = x'y' + 1 \cdot z$ **[P8a, P5aa]**

$= x'y' + z$ **[P3bb]**

or, without using P6a,

$= (x'y'z' + x'y'z) + x'yz + (xy'z + xyz)$

$= x'y' + x'yz + xz$ **[P9a]**

$= x'(y' + yz) + xz$ **[P8a]**

$= x'(y' + z) + xz$ **[P10a]**

$= x'y' + x'z + xz$ **[P8a]**

$= x'y' + z$ **[P9a]**

Note that we could follow a path that does not lead us to the correct answer, by combining the last two terms in the second line of this second sequence, yielding

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

$= x'y' + z(y + x)$ **[P10a]**

$= x'y' + yz + xz$ **[P8a]**

This is a dead end. It has more terms than the minimum (which was given) and we do not have the tools (in Properties 1 to 10) to reduce this further without backing up to the original expression (or, at least, the first reduction). We should then go back and start again.

c. There are two approaches to this problem. In the first, we note that *abc* can be combined with each of the other terms. Thus, we make three copies of it, using

$abc = abc + abc + abc$ **[P6a]**

$f = (abc' + abc) + (ab'c + abc) + (a'bc + abc)$

$= ab + ac + bc$ **[P9a]**

In the second approach, we just use *abc* to combine with the term next to it, producing

$f = abc' + ab'c + a'bc + abc = abc' + ab'c + bc$ **[P9a]**

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

$= abc' + bc + ac$ **[P10a]**

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

$= ac + ab + bc$ **[P10a]**

FIRST PAGES

or, in place of the last two lines,

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

$= bc + ab + ac$ **[P10a]**

In this approach, we used P10a twice to eliminate a literal from the second term and then the first. We could have done it in any order. Indeed, there were two ways to do the last step (as shown on the last two lines).

9. Using Properties 1 to 10, reduce the following expressions to a minimum POS form. Show each step (number of terms and number of literals in parentheses).

a. $(a + b' + c)(a + b' + c')(a' + b + c)(a' + b' + c)$ (2 terms, 4 literals)

b. $(x' + y' + z')(x' + y + z')(x' + y + z)$ (2 terms, 4 literals)

a. We group the first two and the last two terms, and use Property 9b

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

$[a + b'][a' + c]$

b. We can make a second copy of the middle term and group it with each of the others

$(x' + y' + z')(x' + y + z')(x' + y + z) = [(x' + y' + z')$

$(x' + y + z')][(x' + y + z)(x' + y + z')] = [x' + z'][x' + y]$

If we don't make the second copy, we get

$[x' + z'](x' + y + z)$

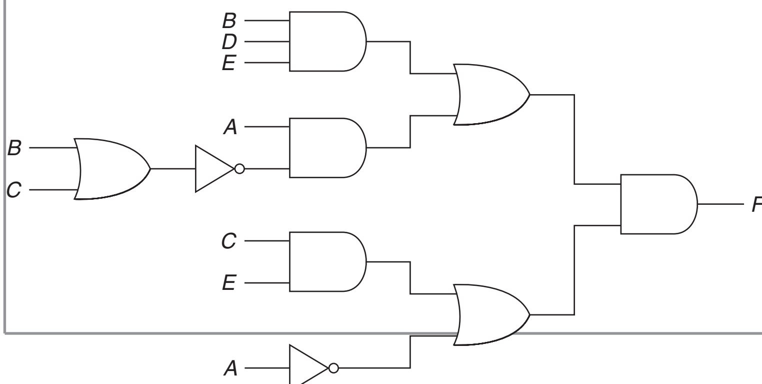
We can then use P8b to get

$x' + z'(y + z) = x' + yz'$ **[P8a, P5bb, P3a]**

$= (x' + y)(x' + z)$ **[P8b]**

10. Show a block diagram of a system using AND, OR, and NOT gates to implement the following function. Assume that variables are available only uncomplemented. Do not manipulate the algebra.

$F = [A(B + C)' + BDE](A' + CE)$



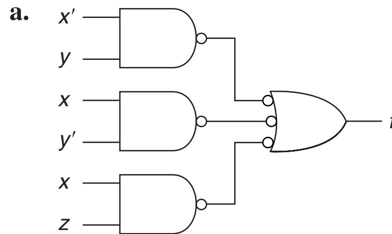
FIRST PAGES

11. For each of the following circuits,

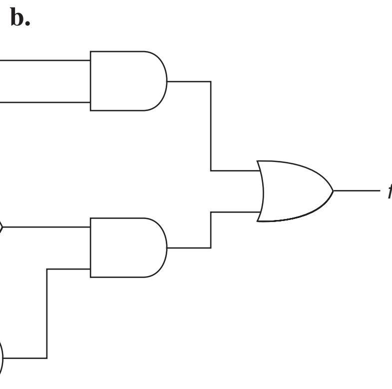
i. find an algebraic expression

ii. put it in SOP form.

a.



b.



a. i. $g = (d + e)c' + cde'$

ii. $g = c'd + c'e + cde'$

b. i. $f = ac + ab'[cd + c'(a + b)]$

ii. $f = ac + ab'cd + ab'c' + ab'c'b$

$= ac + ab'cd + ab'c'$

$= ac + ab'cd + ab'$* **[P10a]**

12. Find the complement of the following expressions. Only single variables may be complemented in the answer.

a. $f = x'yz' + xy'z' + xyz$

b. $g = (w + x' + y)(w' + x + z)(w + x + y + z)$

c. $h = (a + b'c)d' + (a' + c')(c + d)$

a. $f' = (x + y' + z)(x' + y + z)(x' + y' + z')$

SOP becomes POS.

* We will see later that this can be reduced even further.

FIRST PAGES

1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
20
1
2
3
4
5
6
7
8
9
30
1
2
3
4
5
6
7
8
9
40
1
2
3
4
5

b. $g' = w'xy' + wx'z' + w'x'y'z'$

POS becomes SOP.

c. $h' = [a'(b + c') + d][ac + c'd']$

or, step by step

$$h' = [(a + b'c)d']'[(a' + c')(c + d)]'$$
$$= [(a + b'c)' + d][(a' + c')' + (c + d)']$$
$$= [a'(b'c)' + d][ac + c'd']$$
$$= [a'(b + c') + d][ac + c'd']$$

13. For the following truth table,

a	*b*	*c*	*f*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

a. Show the minterms in numerical form.

b. Show an algebraic expression in sum of minterm form.

c. Show a minimum SOP expression (two solutions, three terms, six literals each).

d. Show the minterms of f' (complement of f) in numeric form.

e. Show an algebraic expression in product of maxterm form.

f. Show a minimum POS expression (two terms, five literals).

a. $f(a, b, c) + \Sigma m(1, 3, 4, 6, 7)$

b. $f = a'b'c + a'bc + ab'c' + abc' + abc$

c. $f = a'c + ac' + abc$

$= a'c + ac' + ab$ (using P10a on last two terms)

$= a'c + ac' + bc$ (using P10a on first and last term)

d. $f'(a, b, c) = \Sigma m(0, 2, 5)$

e. $f'(a, b, c) = \Sigma m(0, 2, 5)$

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

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

f. Reordering the first two terms of f, we see that adjacency (P9b) is useful

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

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

FIRST PAGES

Or, we can minimize f' and then use DeMorgan's theorem:

$$f' = a'c' + ab'c$$

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

14. For the following function,

$$f(x, y, z) = \Sigma m(2, 3, 5, 6, 7)$$

a. Show the truth table.

b. Show an algebraic expression in sum of minterm form.

c. Show a minimum SOP expression (two terms, three literals).

d. Show the minterms of f' (complement of f) in numeric form.

e. Show an algebraic expression in product of maxterm form.

f. Show a minimum POS expression (two terms, five literals).

a.

$x\ y\ z$	f
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

b. $f = x'yz' + x'yz + xy'z + xyz' + xyz$

c. $f = x'y + xy'z + xy$

$= y + xy'z$

$= y + xz$

e. $f'(x, y, z) = \Sigma m(0, 1, 4)$

$= x'y'z' + x'y'z + xy'z'$

$f = (x + y + z)(x + y + z')(x' + y + z)$

f. $f' = x'y'z' + x'y'z + xy'z' + x'y'z'$

$= x'y' + y'z'$

$f = (x + y)(y + z)$

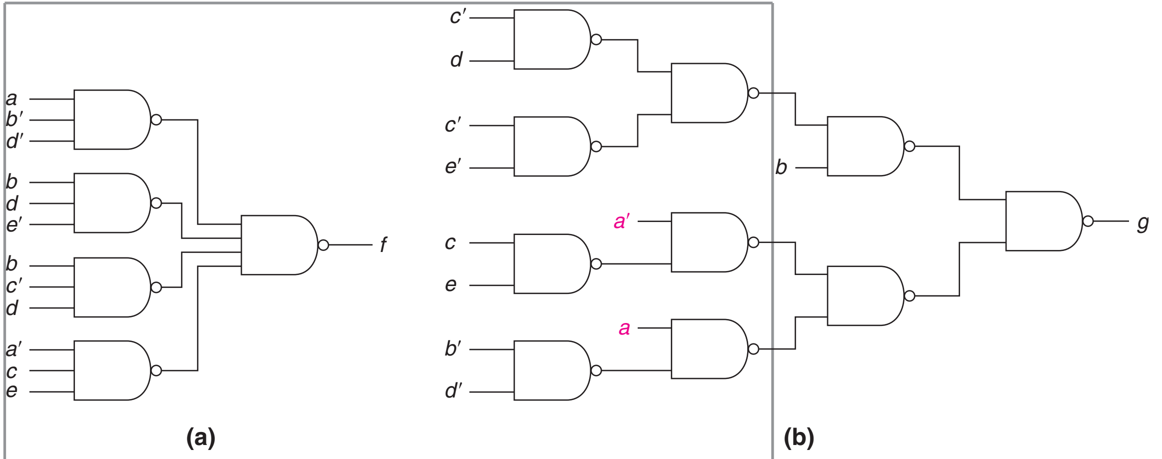
15. Show a block diagram corresponding to each of the expressions below using only NAND gates. Assume all inputs are available both uncomplemented and complemented. There is no need to manipulate the functions to simplify the algebra.

a. $f = ab'd' + bde' + bc'd + a'ce$

b. $g = b(c'd + c'e') + (a + ce)(a' + b'd')$

FIRST PAGES

1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7



8
9
20

Note that in part (a), this is a two-level circuit. In part (b), the only inputs that go directly into an OR are a and a'; they are complemented.

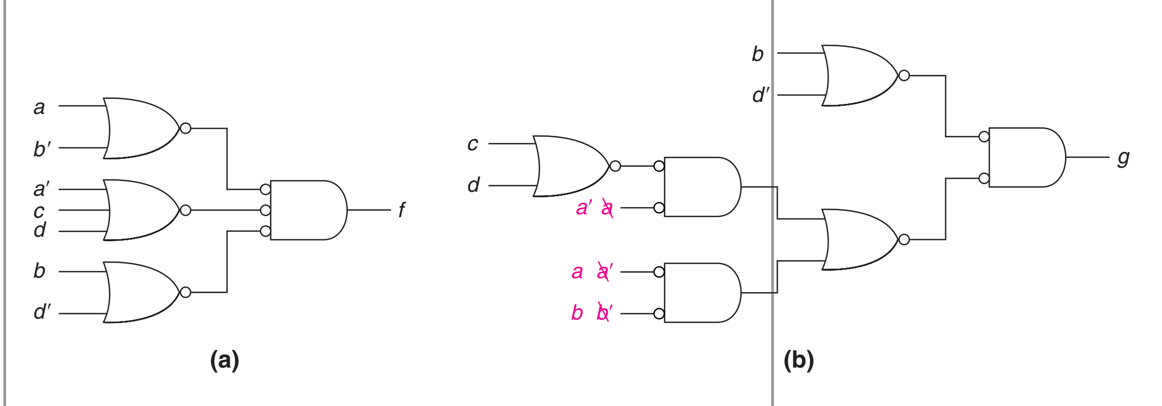
1
2
3
4
5
6
7

16. Show a block diagram corresponding to each of the following expressions using only NOR gates. Assume all inputs are available both uncomplemented and complemented. There is no need to manipulate the functions to simplify the algebra.

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

b. $g = [a'b' + a(c + d)](b + d')$

8
9
30
1
2
3
4
5
6
7
8
9
40



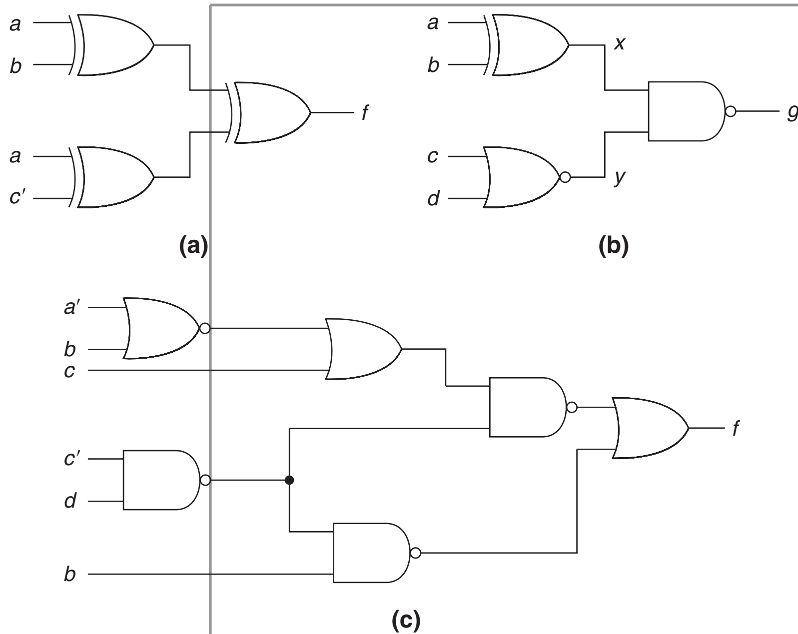
1
2
3
4
5

17. For each of the following circuits,

i. Find an algebraic expression.

ii. Put it in SOP form.

FIRST PAGES



a. i. $f = (a \oplus b) \oplus (a \oplus c')$

ii. $f = (a'b + ab') \oplus (a'c' + ac)$

$= (a'b + ab')'\,(a'c' + ac) + (a'b + ab')\,(a'c' + ac)'$

$= (a'b' + ab)\,(a'c' + ac) + (a'b + ab')(a'c + ac')$

$= a'b'c' + abc + a'bc + ab'c'$

$= b'c' + bc$

b. i. $g = x' + y' = (a'b + ab')' + c + d$

$= ab + a'b' + c + d$

c. i. $f = \{[(a' + b)' + c](c'd)'\}' + [(b(c'd)']'$

ii. $f = \{[(a' + b)' + c]' + (c'd)\} + [b' + c'd]$

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

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

$= c' + a'c' + c'd' + b' = c' + b'$

18. Reduce the following expressions to a minimum SOP form. Show each step (number of terms and number of literals in minimum shown in parentheses).

a. $F = A + B + A'B'C'D$ (3 terms, 4 literals)

b. $f = x'y'z + w'xz + wxyz' + wxz + w'xyz$ (3 terms, 7 literals)

c. $H = AB + B'C + ACD + ABD' + ACD'$ (2 terms, 4 literals)

FIRST PAGES

1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1 2 3 4 5 6 7 8 9 40 1 2 3 4 5

d. $G = ABC' + A'C'D + AB'C' + BC'D + A'D$

(2 terms, 4 literals)

e. $f = abc + b'cd + acd + abd'$

(2 solutions, 3 terms, 9 literals)

a. $F = A + B + A'B'C'D$

$= (A + A'B'C'D) + B$

$= (A + B'C'D) + B$ **[P10a]**

$= A + (B + B'C'D)$

$= A + B + C'D$ **[P10a]**

We can also achieve the same result using a different approach.

$A + B + A'B'C'D = (A + B) + (A + B)'C'D$ **[P11a]**

$= (A + B) + C'D$ **[P10a]**

b. $f = x'y'z + w'xz + wxyz' + wxz + w'xyz$

$= x'y'z + w'xz + wxyz' + wxz$ **[P12a]**

$= x'y'z + xz + wxyz'$ **[P9a]**

$= x'y'z + x(z + wyz')$

$= x'y'z + x(z + wy)$ **[P10a]**

$= x'y'z + xz + wxy$

$= z(x'y' + x) + wxy$

$= z(y' + x) + wxy$ **[P10a]**

$= y'z + xz + wxy$

c. $H = AB + B'C + ACD + ABD' + ACD'$

$= AB + B'C + AC$ **[P12a, P9a]**

$= AB + B'C$ **[P13a]**

d. $G = ABC' + A'C'D + AB'C' + BC'D + A'D$

$= ABC' + AB'C' + A'D + BC'D$ **[P12a]**

$= AC' + A'D + BC'D$ **[P9a]**

But,

$AC' ¢ A'D = C'D$

$G = AC' + A'D + BC'D + C'D$ **[P13a]**

$= AC' + A'D + C'D$ **[P12a]**

$= AC' + A'D$ **[P13a]**

Note that we used consensus to first add a term and then to remove that same term.

e. $f = abc + b'cd + acd + abd'$

Since

$abc ¢ b'cd = acd$

FIRST PAGES

the consensus term can be removed and thus

$$f = abc + b'cd + abd'$$

No further reduction is possible; the only consensus that exists among the terms in this reduced expression produces the term *acd*, the one that we just removed. None of the other properties can be used to reduce this function further.

However, if we go back to the original function, we note that another consensus does exist:

$$acd ¢ abd' = abc$$

and thus the term *abc* can be removed, producing

$$f = b'cd + acd + abd'$$

That is another equally good minimum solution (because no further minimization is possible). Even though we found two applications of consensus in this function, we cannot take advantage of both of them because no matter which one we use first, the term needed to form the second consensus has been removed.

19. Expand the following function to sum of minterms form

$$F(A, B, C) = A + B'C$$

We have a choice of two approaches. We could use P3b, P5aa (both from right to left) and P8a repeatedly to produce

$$\begin{aligned} A + B'C &= A(B' + B) + (A' + A)B'C \\ &= AB' + AB + A'B'C + AB'C \\ &= AB'(C' + C) + AB(C' + C) + A'B'C + AB'C \\ &= AB'C' + AB'C + ABC' + ABC + A'B'C + AB'C \\ &= AB'C' + AB'C + ABC' + ABC + A'B'C \end{aligned}$$

having removed the duplicated term ($AB'C$). Or we could use a truth table, such as

A B C	B′C	F
0 0 0	0	0
0 0 1	1	1
0 1 0	0	0
0 1 1	0	0
1 0 0	0	1
1 0 1	1	1
1 1 0	0	1
1 1 1	0	1

FIRST PAGES

1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1 2 3 4 5 6 7 8 9 40 1 2 3 4 5

and thus,

$$F = A'B'C + AB'C' + AB'C + ABC' + ABC$$

which is the same as the previous expression reordered, or

$$F(A, B, C) = \Sigma m(1, 4, 5, 6, 7)$$

20. Convert each of the following expressions to SOP form:

a. $(w + x' + z)(w' + y + z')(x + y + z)$

b. $(a + b + c + d')(b + c + d)(b' + c')$

a. $(w + x' + z)(w' + y + z')(x + y + z)$

$= [z + (w + x')(x + y)](w' + y + z')$ **[P8b]**

$= (z + wx + x'y)(w' + y + z')$ **[P14a]**

$= z(w' + y) + z'(wx + x'y)$ **[P14a]**

$= w'z + yz + wxz' + x'yz'$ **[P8a]**

Note that this is not a minimum SOP expression, even though the original was a minimum POS expression. Using P10a, we could reduce this to

$$w'z + yz + wxz' + x'y$$

b. $(a + b + c + d')(b + c + d)(b' + c')$

$= [b + c + (a + d')d](b' + c')$ **[P8b]**

$= (b + c + ad)(b' + c')$ **[P8b, P5b, P3a]**

$= bc' + b'(c + ad)$ **[P14a]**

$= bc' + b'c + ab'd$ **[P8a]**

or using c instead of b for P14a

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

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

$= b'c + bc' + ac'd$

These are two equally good solutions.

21. Convert the following expression to POS form:

$$a'c'd + a'cd' + bc$$

$a'c'd + a'cd' + bc$

$= c(b + a'd') + c'a'd$ **[P8a]**

$= (c + a'd)(c' + b + a'd')$ **[P14a]**

$= (c + a')(c + d)(c' + b + a')(c' + b + d')$ **[P8b]**

FIRST PAGES

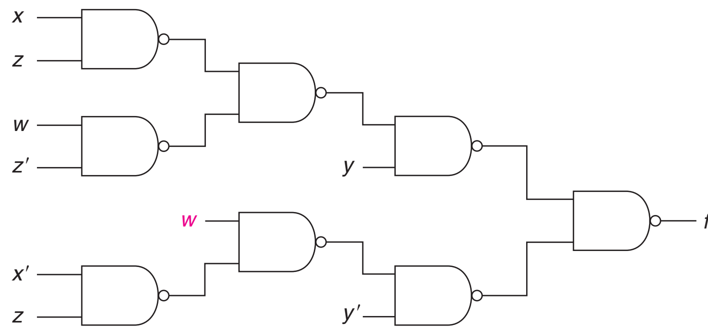
Two comments are in order. This is not in minimum POS form. P12b allows us to manipulate the first and third terms so as to replace the third term by $(a' + b)$. We could have started the process by factoring a' from the first two terms, but that would require more work.

22. Implement each of the following expressions (which are already in minimum SOP form) using only two-input NAND gates. No gate may be used as a NOT. All inputs are available both uncomplemented and complemented. (The number of gates required is shown in parentheses.)

a. $f = w'y' + xyz + wyz' + x'y'z$ (8 gates)

b. $g = a'b'c'd' + abcd' + a'ce + ab'd + be$ (12 gates)

a. $f = y'(w' + x'z) + y(xz + wz')$



b. $g = a'b'c'd' + abcd' + a'ce + ab'd + be$

The first attempt at a solution yields one with 13 gates.

$$\begin{aligned} g &= d'(a'b'c' + abc) + e(b + a'c) + ab'd \\ &= (d' + ab')(d + a'b'c' + abc) + e(b + a'c) \\ &= (d' + ab')(d + (a + b'c')(a' + bc)) + e(b + a'c) \end{aligned}$$

1 2 3 4 5 6 7 8 9 10 11 12 13

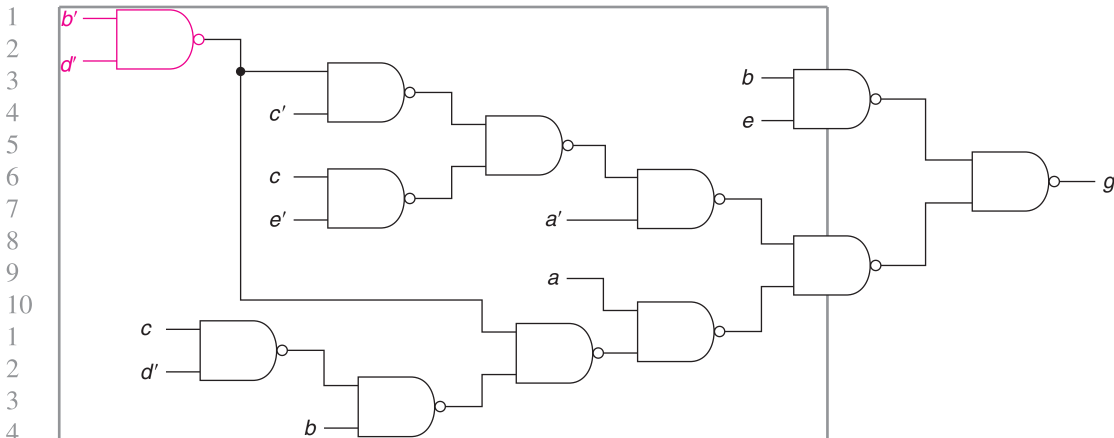
Another approach is

$$\begin{aligned} g &= a'(b'c'd' + ce) + a(bcd' + b'd) + be \\ &= [a + b'c'd' + ce][a' + bcd' + b'd] + be \\ &= [a + (c + b'd')(c' + e)][a' + (b + d)(b' + cd')] + be \end{aligned}$$

1 2 3 4 5 6 7 3 8 9 10 11 12

where gate three is used twice, as follows.

FIRST PAGES



23. For the following function, show the block diagram for a NAND gate implementation that uses only four 7400 series NAND gate modules. No gate may be used as a NOT. Assume that all variables are available both uncomplemented and complemented. (Note that a two-level solution would require 2 six-input gates and a five-input gate (each of which would be implemented with a 7430 module containing 1 eight-input gate), plus a 7420 for the four-input gate and a 7410 for the 2 three-input gates and the 1 two-input gate.)

$$g = abcdef + d'e'f + a'b' + c'd'e' + a'def' + abcd'f'$$

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

This requires 1 four-input gate (for the first term), 4 three-input gates, and 5 two-input gates (one 7420, with the second gate used as a three-input one, one 7410, and two 7400s with three gates unused). If we required that no four-input gates be used, we could further manipulate the algebra as follows:

$$g = [a' + bc(def + d'f')][a + b' + def'] + d'e'(c' + f)$$

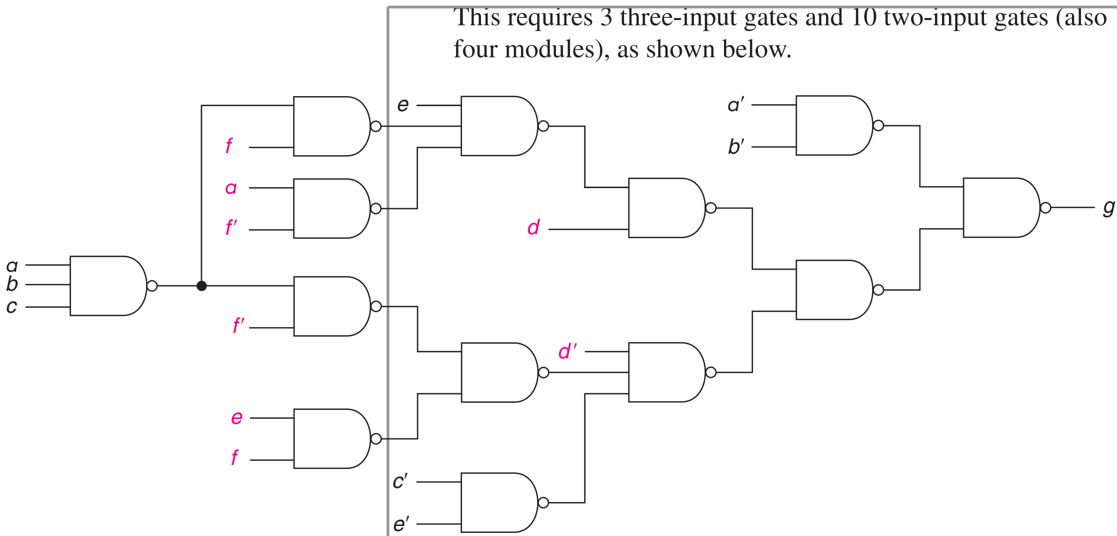
using P14a on the first and last terms, which would require 5 three-input gates and 6 two-input gates (still four modules).

We could also do a completely different factoring, yielding

$$\begin{aligned} g &= de(abcf + a'f') + d'(abcf' + e'f + c'e') + a'b' \\ &= [d' + e(abcf + a'f')][d + abcf' + e'f + c'e'] + a'b' \\ &= [d' + e(a' + f)(f' + abc)] \\ &\qquad \cdot [d + c'e' + (f + abc)(f' + e')] + a'b' \end{aligned}$$

FIRST PAGES

This requires 3 three-input gates and 10 two-input gates (also four modules), as shown below.



2.10 EXERCISES

1. Show a truth table for a 1-bit full subtractor that has a borrow input b_{in} and inputs x and y, and produces a difference, d, and a borrow output, b_{out}.

$$\begin{array}{r} b^{in} \\ x \\ \underline{-y} \\ b_{out}\ d \end{array}$$

2. Show truth tables for each of the following.

   *a. There are four inputs and three outputs. The inputs, w, x, y, z, are codes for the grade that may be received:

0000 A	0100 B−	1000 D+	1100 Incomplete
0001 A−	0101 C+	1001 D	1101 Satisfactory
0010 B+	0110 C	1010 D−	1110 Unsatisfactory
0011 B	0111 C−	1011 F	1111 Pass

   The outputs are

   1: a 1 if and only if the grade is C or better (only letter grades; C− is not C or better)

   2: a 1 if and only if the university will count it toward the 120 credits required for a degree (passing grade only)

   3: a 1 if and only if it will be counted in computing a grade point average (letter grades only).

FIRST PAGES

b. This system has four inputs and three outputs. The first two inputs, a and b, represent a 2-bit binary number (range of 0 to 3). A second binary number (same range) is represented by the other two inputs, c and d. The output f is to be 1 if and only if the two numbers differ by exactly 2. Output g is to be 1 if and only if the numbers are equal. Output h is to be 1 if and only if the second number is larger than the first.

c. The system has four inputs. The first two, a and b, represent a number in the range 1 to 3 (0 is not used). The other two, c and d, represent a second number in the same range. The output, y, is to be 1 if and only if the first number is greater than the second or the second is 2 greater than the first.

*d. A system has one output, F, and four inputs, where the first two inputs (A, B) represent one 2-bit binary number (in the range 0 to 3) and the second two inputs (C, D) represent another binary number (same range). F is to be 1 if and only if the two numbers are equal or if they differ by exactly 1.

e. A system has one output, F, and four inputs, where the first two inputs (A, B) represent one 2-bit binary number (in the range 0 to 3) and the second two inputs (C, D) represent another binary number (same range). F is to be 1 if and only if the sum of the two numbers is odd.

f. The system has four inputs. The first two, a and b, represent a number in the range 0 to 2 (3 is not used). The other two, c and d, represent a second number in the same range. The output, y, is to be 1 if and only if the two numbers do not differ by more than 1.

g. The problem is to design a ball and strike counter for baseball. The inputs are how many balls (0, 1, 2, or 3) before this pitch, how many strikes (0, 1, 2) before this pitch, and what happens on this pitch. The outputs are how many balls after this pitch (0, 1, 2, 3, 4) or how many strikes after this pitch (0, 1, 2, 3).

In baseball, there are four outcomes of any pitch (from the point of view of this problem). It can be a strike, a foul ball, a ball, or anything else that will end this batter's turn (such as a hit or a fly out).

A foul ball is considered a strike, except when there are already two strikes, in which case the number of strikes remain 2. The output is to indicate the number of balls and strikes after this pitch (even if the pitch is the fourth ball or the third strike, in which case the batter's turn is over). If the batter's turn is over for any other reason, the output should indicate 0 balls and 0 strikes.

FIRST PAGES

Show the code for the inputs (there are six inputs, two for what happened on that pitch, two for the number of balls, and two for the number of strikes) and for the outputs (there should be 5: 3 for balls and 2 for strikes). Then show the 64 line truth table.

*h. The months of the year are coded in four variables, *abcd*, such that January is 0000, February is 0001, . . . , and December is 1011. The remaining 4 combinations are never used. (Remember: 30 days has September, April, June, and November. All the rest have 31, except February. . . .) Show a truth table for a function, *g*, that is 1 if the month has 31 days and 0 if it does not.

i. The months of the year are coded as in 10h, except that February of a leap year is coded as 1100. Show a truth table with five outputs, *v, w, x, y, z* that indicates the number of days in the selected month.

j. Repeat 10i, except that the outputs are to be in BCD (8421 code). There are now six outputs, *u, v, w, x, y, z* (where the first decimal digit is coded 0, 0, *u, v* and the second digit is coded *w, x, y, z*).

k. The system has four inputs, *a, b, c,* and *d* and one output, *f*. The last three inputs (*b, c, d*) represent a binary number, *n*, in the range 0 to 7; however, the input 0 never occurs. The first input (*a*) specifies which of two computations is made.

$a = 0$: f is 1 iff n is a multiple of 2
$a = 1$: f is 1 iff n is a multiple of 3

l. The system has four inputs, *a, b, c,* and *d* and one output, *f*. The first two inputs (*a, b*) represent one binary number (in the range 0 to 3) and the last two (*c, d*) represent another number in the range 1 to 3 (0 never occurs). The output, *f*, is to be 1 iff the second number is at least two larger than the first.

3. Show a block diagram of a circuit using AND and OR gates for each side of each of the following equalities:

*a. P2a: $a + (b + c) = (a + b) + c$

b. P8a: $a(b + c) = ab + ac$

4. Show a truth table for the following functions:

*a. $F = X'Y + Y'Z' + XYZ$

b. $G = XY + (X' + Z)(Y + Z')$

c. $H = WX + XY' + WX'Z + XYZ' + W'XY'$

5. Determine, using truth tables, which expressions in each of the groups are equal:

a. $f = ac' + a'c + bc$
$g = (a + c)(a' + b + c')$

FIRST PAGES

*b. $f = a'c' + bc + ab'$
$g = b'c' + a'c' + ac$
$h = b'c' + ac + a'b$

c. $f = ab + ac + a'bd$
$g = bd + ab'c + abd'$

6. For each of the following expressions, indicate which (if any) of the following apply (more than one may apply):

i. Product term
ii. SOP expression
iii. Sum term
iv. POS expression

a. $abc'd + b'cd + ad'$
*b. $a' + b + cd$
c. $b'c'd'$
*d. $(a + b)c'$
e. $a' + b$
*f. a'
*g. $a(b + c) + a'(b' + d)$
h. $(a + b' + d)(a' + b + c)$

***7.** For the expressions of problem 4, how many literals are in each?

8. Using properties 1 to 10, reduce the following expressions to a minimum SOP form. Show each step (number of terms and number of literals in minimum shown in parentheses).

*a. $x'z + xy'z + xyz$ (1 term, 1 literal)
b. $x'y'z' + x'yz + xyz$ (2 terms, 5 literals)
c. $x'y'z' + x'y'z + xy'z + xyz'$ (3 terms, 7 literals)
*d. $a'b'c' + a'b'c + abc + ab'c$ (2 terms, 4 literals)
e. $x'y'z' + x'yz' + x'yz + xyz$ (2 terms, 4 literals)
*f. $x'y'z' + x'y'z + x'yz + xyz + xyz'$ (2 solutions, each with 3 terms, 6 literals)
g. $x'y'z' + x'y'z + x'yz + xy'z + xyz + xyz'$ (3 terms, 5 literals)
h. $a'b'c' + a'bc' + a'bc + ab'c + abc' + abc$ (3 terms, 5 literals)

9. Using Properties 1 to 10, reduce the following expressions to a minimum POS form. The number of terms and number of literals are shown in parentheses.

a. $(a + b + c)(a + b' + c)(a + b' + c')(a' + b' + c')$ (2 terms, 4 literals)

FIRST PAGES

b. $(x + y + z)(x + y + z')(x + y' + z)(x + y' + z')$
(1 term, 1 literal)

*c. $(a + b + c')(a + b' + c')(a' + b' + c')(a' + b' + c)$
$(a' + b + c)$ (2 solutions, each with 3 terms, 6 literals)

10. Show a block diagram of a system using AND, OR, and NOT gates to implement the following functions. Assume that variables are available only uncomplemented. Do not manipulate the algebra.

a. $P'Q' + PR + Q'R$

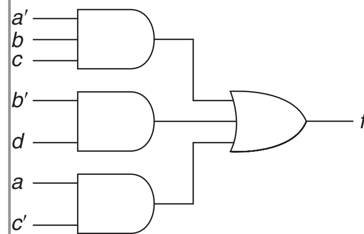
b. $ab + c(a + b)$

*c. $wx'(v + y'z) + (w'y + v')(x + yz)'$

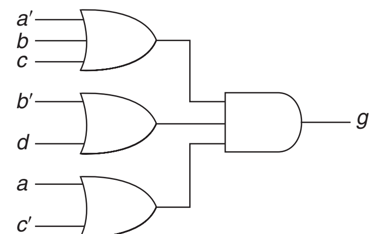
11. For each of the following circuits,

i. find an algebraic expression

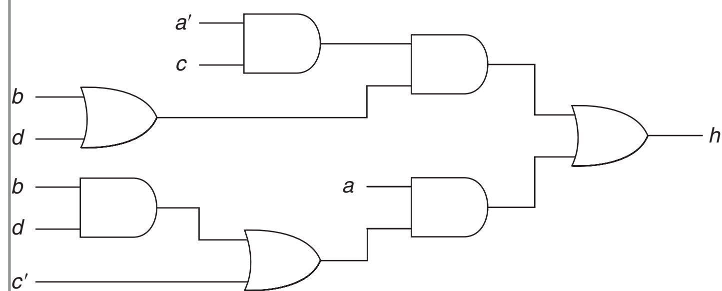
ii. put it in sum of product form.



(a)



(b)



(c)

12. Find the complement of the following expressions. Only single variables may be complemented in the answer.

*a. $f = abd' + b'c' + a'cd + a'bc'd$

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

c. $h = (a + b)(b' + c) + d'(a'b + c)$

FIRST PAGES

1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1 2 3 4 5 6 7 8 9 40 1 2 3 4 5

13. For each of the following functions:

$f(x, y, z) = \Sigma m(1, 3, 6)$
$g(x, y, z) = \Sigma m(0, 2, 4, 6)$

a. Show the truth table.

b. Show an algebraic expression in sum of minterms form.

c. Show a minimum SOP expression (*a*: 2 terms, 5 literals; *b*: 1 term, 1 literal).

d. Show the minterms of f' (complement of f) in numeric form.

e. Show an algebraic expression in product of maxterms form.

f. Show a minimum POS expression (f: 2 solutions, 3 terms, 6 literals; g: 1 term, 1 literal)

***14.** For each of the following functions,

a b c	*f*	*g*
0 0 0	0	1
0 0 1	1	1
0 1 0	0	0
0 1 1	0	0
1 0 0	0	1
1 0 1	1	1
1 1 0	1	1
1 1 1	1	0

a. Show the minterms in numerical form.

b. Show an algebraic expression in sum of minterms form.

c. Show a minimum SOP expression (f: 2 terms, 4 literals; g: 2 terms, 3 literals).

d. Show the minterms of f' (complement of f) in numeric form.

e. Show an algebraic expression in product of maxterms form.

f. Show a minimum POS expression (f: 2 terms, 2 literals; g: 2 terms, 3 literals)

***15.** Consider the following function with don't cares:

$G(X, Y, Z) = \Sigma m(5, 6) + \Sigma d(1, 2, 4)$

For each of the following expressions, indicate whether it could be used as a solution for G. (Note: it may not be a minimum solution.)

a. $XYZ' + XY'Z$

b. $Z' + XY'Z$

c. $X(Y' + Z')$

d. $Y'Z + XZ' + X'Z$

e. $XZ' + X'Z$

f. $YZ' + Y'Z$

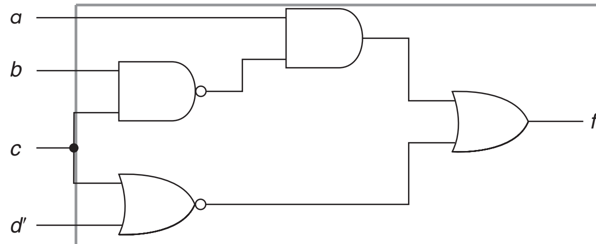
16. Show that the NOR is functionally complete by implementing a NOT, a two-input AND, and a two-input OR using only two-input NORs.

17. For each of the following circuits,

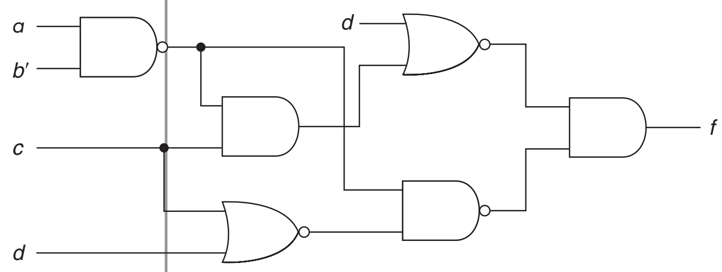
i. find an algebraic expression

ii. put it in SOP form.

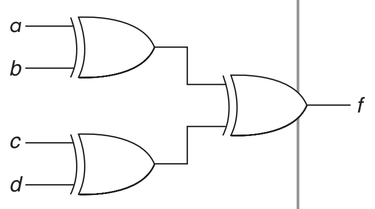
FIRST PAGES



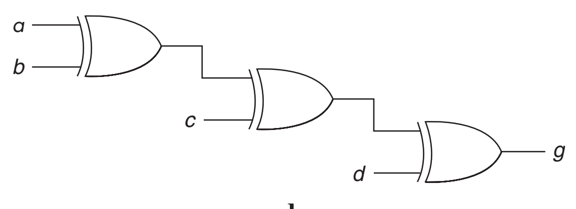
*a.



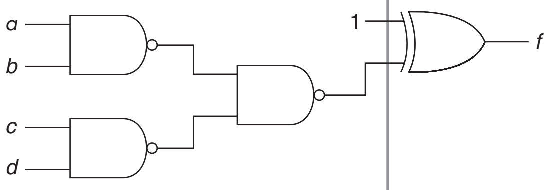
b.



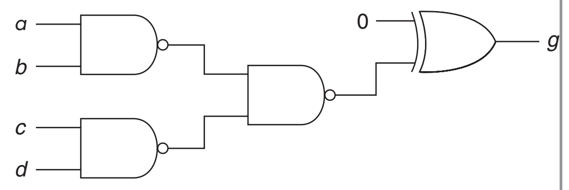
c.



d.



e.



f.

FIRST PAGES

1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1 2 3 4 5 6 7 8 9 40 1 2 3 4 5

18. Show a block diagram corresponding to each of the expressions below using only NAND gates. Assume all inputs are available both uncomplemented and complemented. Do not manipulate the functions to simplify the algebra.

a. $f = wy' + wxz' + xy'z + w'x'z$

b. $g = wx + (w' + y)(x + y')$

c. $h = z(x'y + w'x') + w(y' + xz')$

*d. $F = D[B'(A' + E') + AE(B + C)] + BD'(A'C' + AE')$

19. Reduce the following expressions to a minimum SOP form, using P1 through P12. Show each step (number of terms and number of literals in minimum shown in parentheses).

a. $h = ab'c + bd + bcd' + ab'c' + abc'd$ (3 terms, 6 literals)

b. $h = ab' + bc'd' + abc'd + bc$ (3 terms, 5 literals)

*c. $f = ab + a'bd + bcd + abc' + a'bd' + a'c$ (2 terms, 3 literals)

d. $g = abc + abd + bc'd'$ (2 terms, 5 literals)

20. Reduce the following expressions to a minimum SOP form. Show each step and the property used (number of terms and number of literals in minimum shown in parentheses).

a. $f = x'yz + w'x'z + x'y + wxy + w'y'z$ (3 terms, 7 literals)

b. $G = A'B'C' + AB'D + BCD' + A'BD + CD + A'D$ (4 terms, 9 literals)

*c. $F = W'YZ' + Y'Z + WXZ + WXYZ' + XY'Z + W'Y'Z'$ (3 terms, 7 literals)

d. $g = wxz + xy'z + wz' + xyz + wxy'z + w'y'z'$ (3 terms, 6 literals)

*e. $G = B'C'D + BC + A'BD + ACD + A'D$ (3 terms, 6 literals)

f. $h = abc' + ab'd + bcd + a'bc$ (3 terms, 8 literals)

*g. $g = a'bc' + bc'd + abd + abc + bcd' + a'bd'$ (2 solutions, 3 terms, 9 literals)

21. i. For the following functions, use consensus to add as many new terms to the SOP expression given.

ii. Then reduce each to a minimum SOP, showing each step and the property used.

*a. $f = a'b'c' + a'bd + a'cd' + abc$ (3 terms, 8 literals)

b. $g = wxy + w'y'z + xyz + w'yz'$ (3 terms, 8 literals)

22. Expand the following functions to sum of minterms form:

a. $f(a, b, c) = ab' + b'c'$

*b. $g(x, y, z) = x' + yz + y'z'$

c. $h(a, b, c, d) = ab'c + bd + a'd'$

FIRST PAGES

23. Convert each of the following expressions to SOP form:

a. $(a + b + c + d')(b + c' + d)(a + c)$

b. $(a' + b + c')(b + c' + d)(b' + d')$

*c. $(w' + x)(y + z)(w' + y)(x + y' + z)$

24. Convert each of the following expressions to POS form:

a. $AC + A'D'$

b. $w'xy' + wxy + xz$

*c. $bc'd + a'b'd + b'cd'$

25. Implement each of the following expressions (which are already in minimum SOP form) using only two-input NAND gates. No gate may be used as a NOT. All inputs are available both uncomplemented and complemented. (The number of gates required is shown in parentheses.)

*a. $f = wy' + wxz' + y'z + w'x'z$ (7 gates)

b. $ab'd' + bde' + bc'd + a'ce$ (10 gates)

c. $H = A'B'E' + A'B'CD' + B'D'E' + BDE' + BC'E + ACE'$ (14 gates)

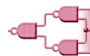
*d. $F = A'B'D' + ABC' + B'CD'E + A'B'C + BC'D$ (11 gates)

e. $G = B'D'E' + A'BC'D + ACE + AC'E' + B'CE$ (12 gates, one of which is shared)

26. Each of the following is already in minimum SOP form. All variables are available both uncomplemented and complemented. Find two solutions each of which uses no more than the number of integrated circuit packages of NAND gates (4 two-input or 3 three-input or 2 four-input gates per package) listed. One solution must use only two and three input gates; the other must use at least 1 four-input gate package.

*a. $F = ABCDE + B'E' + CD'E' + BC'D'E + A'B'C + A'BC'E$ (3 packages)

b. $G = ABCDEF + A'B'D' + C'D'E + AB'CE' + A'BC'DF + ABE'F'$ (4 packages)



2.11 CHAPTER 2 TEST (100 MINUTES, OR TWO 50-MINUTE TESTS)

1. The inputs of this system A and B represent one binary number in the range 0:3. The inputs *C* and *D* represent a second binary number (also in the range 0:3). There are three outputs, *X, Y,* and *Z.*

Show a truth table such that *Y* and *Z* represent a number equal to the magnitude of the difference of the two inputs and *X* is 1 if and only if the first is larger. Two lines of the table are filled in.

FIRST PAGES

A	B	C	D	X	Y	Z
0	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	0	1
0	1	1	1			
1	0	0	0			
1	0	0	1	1	0	1
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			

2. Use a truth table to demonstrate whether or not the following functions are equal:

$f = a'b' + a'c' + ab$
$g = (b' + c')(a' + b)$

a b c	f	g
0 0 0		
0 0 1		
0 1 0		
0 1 1		
1 0 0		
1 0 1		
1 1 0		
1 1 1		

3. Reduce the following expression to a SOP expression with two terms and four literals. Show each step.

$a'b'c + a'bc + ab'c + ab'c'$

4. For each part, assume all variables are available both uncomplemented and complemented.

$f = ab'c + ad + bd$

a. Show a block diagram for a two-level implementation of f using AND and OR gates.

b. Show a block diagram for an implementation of f using only two-input AND and OR gates.

FIRST PAGES

5. For the following truth table

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

a. Write a sum of minterms function in numeric form, for example,

$\Sigma m(0, \ldots)$

b. Write a sum of minterms function in algebraic form, for example,

$x'yz + \cdots$

c. Find one minimum SOP expression (3 terms, 6 literals).

d. Find a POS expression in product of maxterms form.

e. Find a minimum POS form (2 terms, 5 literals).

6. Assume all inputs are available both uncomplemented and complemented. Show a two-level implementation of

$$g = wx + wz + w'x' + w'y'z'$$
$$= (w' + x + z)(w + x' + y')(w + x' + z')$$

a. using NAND gates of any size

b. using NOR gates of any size

c. using two-input NAND gates (none of which may be used as a NOT)

7. For each of the following functions find a minimum SOP expression (3 terms, 6 literals). Show each algebraic step.

a. $f = b'd' + bc'd + b'cd' + bcd + ab'd$

5-POINT BONUS: Find a second minimum sum of products.

b. $g = xy'z' + yz + xy'z + wxy + xz$

8. a. Expand the following to sum of minterms (sum of standard product terms). Eliminate any duplicates.

$$g = a' + ac + b'c$$

b. Manipulate the following to a SOP expression.

$$f = (x' + y)(w' + y + z')(y' + z)(w + y' + z')$$

FIRST PAGES

9. Implement the following function using only two-input NAND gates. No gate may be used as a NOT gate. The function is in minimum SOP form. All inputs are available both uncomplemented and complemented.

$f = ac + bcd + a'b'd'$ (7 gates)

10. Implement the following function using only two-input NAND gates. No gate may be used as a NOT gate. The function is in minimum SOP form. All inputs are available both uncomplemented and complemented.

$f = abc + ac'd'e' + a'd'e + ce + cd$

(Full credit for 11 gates, 5-point bonus for 10 gates)

FIRST PAGES