Combinational Logic

O purblind race of miserable men,
How many among us at this very hour
Do forge a lifelong trouble for ourselves,
By taking true for false, and false for true!
—Alfred, Lord Tennyson

It is a riddle wrapped in a mystery
inside an enigma.
— Sir Winston Churchill

Introduction

This chapter begins our detailed examination of the implementation of digital systems. We start with *combinational logic design*, the design and implementation of logic functions whose outputs depend solely on their inputs. The full adder introduced in Chapter 1 is just such a circuit.

We start with the representation of a function as a truth table or a Boolean equation. We will introduce a "canonical," or standard, representation of Boolean equations, called the *sum of products two-level form*. We can think of this as a unique way to represent a Boolean function, like a fingerprint. The form expresses the function as ANDed terms (first level of gates) that are then ORed together (second level of gates). An alternative canonical form, the *product of sums form*, has ORs at the first level and ANDs at the second level.

You can implement a Boolean function as logic gates in more than one way. It is highly desirable to find the simplest implementation—that is, the one with the smallest number of gates or wires. The process of reducing a Boolean function to its simplest two-level form is called *Boolean minimization* or *reduction*. The formal process for Boolean minimization will be described in Chapter 4.

Just as a complex algebraic expression can be simplified by factoring out common subexpressions, you can implement a Boolean function in fewer gates if you factor it judiciously. This leads to a fundamental trade-off between time (more lev-

els of logic to pass through) and space (fewer gates needed to implement the function). We will introduce the basic ideas of multi-level logic in this chapter.

Logic circuits are more than simply abstract implementations of mathematical equations. They are constructed from physical devices that take some time to compute their functions. An actual implementation of a logic circuit does not determine its output instantly; it takes time for the information signals to propagate from a change in inputs to a final change in outputs. In this chapter, we will introduce the concept of *time response in digital networks.*

Hardware description languages (HDLs) are an alternative way to describe logic circuits in a textual rather than graphical form. We will introduce the basic concepts here and will use a simplified form of a particular HDL called Verilog throughout this book.

2.1 Outputs as a Function of Inputs

2.1.1 Combinational Logic Defined

Combinational logic is the kind of digital system whose output behavior depends only on the current inputs. Such a system is *memoryless*: its outputs are independent of the historical sequence of values presented to it as inputs.

For example, a digital system that adds two input bits together to form sum and carry output bits is combinational. Changing the inputs causes the outputs to change, after a small delay. But the previous input values have no effect on the final output of the system.

S*equential logic*, on the other hand, adds the notion of *memory* or *state* to combinational logic to produce systems whose output behavior does depend on the sequence of inputs and not just the last inputs (and hence the name sequential logic). We will examine this kind of digital systems in detail in Chapter 7.

A traffic light controller is an example of sequential logic. A traffic light cycles through the sequence green-yellow-red. So when the light changes in response to input changes (perhaps a pedestrian has pushed the crossing button, a car is detected as waiting to cross the intersection, or a timer has gone off), the next light to illuminate depends also on the currently illuminated light.

There are many ways to describe combinational logic: Boolean algebra expressions, wired up logic gates, truth tables tabulating input and output combinations, and even program statements in a hardware description language. Each of these will be introduced in the following sections.

2.1.2 More Examples of Combinational Logic

Let’s look at several digital systems that can be implemented as combinational logic. We will begin with a system that detects equivalence among its inputs. Given two binary inputs, X and Y, the Equal output is set to 1 if both inputs have the same value. Either X and Y are both 0 or both 1. Such a system depends only its current inputs, and not the sequence of previous inputs. Thus the system is

Equivalence Circuit

X	Y	Equal
0	0	1
0	1	0
1	0	0
1	1	1

Figure 2.1 Truth table for the Equivalence Circuit

Tally Circuit

X	Y	Zero	One	Two
0	0	1	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1

Figure 2.2 Truth Table for the Tally Circuit

Divide by 2, 3, 5 Circuit

X	Y	Z	By2	By3	By5
0	0	0	1	1	1
0	0	1	0	0	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	1	0	1	1	0
1	1	1	0	0	0

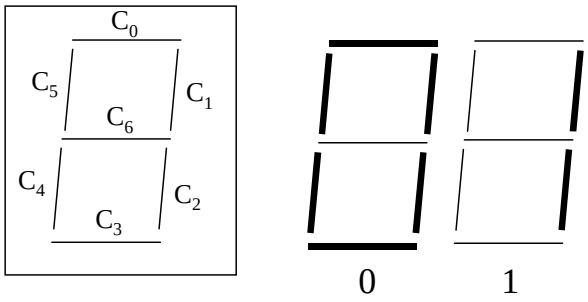
Figure 2.3 Truth Table for Divide by 2, 3, 5

combinational. It is easily described in terms of a truth table, as shown in Figure 2.1.

Now consider a "tally" circuit of two binary inputs, X and Y, and three binary outputs, Zero, One, and Two. This digital system counts the number of ones among its inputs, asserting the appropriate output to indicate the result. The Tally circuit makes Zero true if both inputs are 0. One is true if either input is 1 but not both. Two is true only if both inputs are 1. Again, the outputs depend only on the current inputs, and so the system is combinational. Its truth table is shown in Figure 2.2.

Suppose that you wanted to design a system that given a binary number in the range of 000 to 111 (0 to 7), it could determine if the number was divisible by 2, 3, or 5 without a remainder. Such a circuit turns out to be a useful component of a variety of clock and timer systems. A generalization of this circuit could be used, for example, to cause a digital watch to chime differently at 15 minutes before and after the hour (the time is divisible by 3 and 5 but not 2) then at the half hour and hour (divisible by 2, 3, and 5 at the same time). We can write down the desired behavior as another truth table, as shown in Figure 2.3. Since the outputs only depend on the current inputs, the circuit is once again combinational.

As a final example of a combinational logic system, suppose we have been asked to design a circuit that accepts a binary digit, 0 or 1, and decodes it into a set of signals to drive a 7 segment display. The display element and its control signals are shown in Figure 2.4. A segment within the display is illuminated if its associated control signal is asserted. To display a 0, all segments except the center segment are turned on: the control signals C_0 through C_5 are asserted. A 1 is represented by illuminating the segments controlled by C_1 and C_2. Since the illuminated segments only depend on the current inputs, the system must be combinational.



X	C_0	C_1	C_2	C_3	C_4	C_5	C_6
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0

Figure 2.4 Binary Digit Display Combinational System

We don't want to leave you with the impression that just about every digital system is combinational. Here are some examples of systems that are *not* combinational. A digital system takes as input a single binary digit stream, and asserts its single flag output whenever it has seen the sequence 1 0 0 in its inputs. Such a system needs to "remember" that it has seen a 1 followed by a 0 followed by another 0 before it can assert its output. Since its behavior depends on its memory of the earlier input sequence, it is a sequential system.

Counters are another example of digital systems that are inherently sequential. A three-bit counter that advances in binary sequence, such as 000, 001, 010, 011, 100, 101, 110, 111, and repeats, needs to keep track of the current position in this count sequence before it advances to the next one in the proper order. This implies that the counter must use memory for this purpose, and so cannot determine the next state from the external inputs alone.

A combinational lock offers a third and final example of a sequential system. Consider an unlock sequence that requires that a specific string of bits be entered into the lock, one bit at a time. The lock is released only if the correct sequence is entered. The system must remember this sequence before it can exercise its lock release output behavior. This implies that it must be a sequential circuit.

2.2 Laws and Theorems of Boolean Logic

In Chapter 1 you saw that (at least some) Boolean expressions can be represented by logic gates and vice versa. Actually, all Boolean functions can be implemented in terms of AND, OR, and NOT gates. Because of this close relationship between the laws of Boolean algebra and the behavior of logic gates, theorems that are true for Boolean algebra can also be used to transform digital logic into simpler and standardized forms.

2.2.1 Axioms of Boolean Algebra

A Boolean algebra consists of a set of elements *B,* together with two binary operations {+} and {•} and a unary operation {'}, so that the following hold:

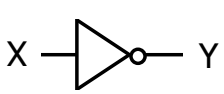
1. The set B contains at least two elements a, b such that a is not equal to b.
2. *Closure*: For every a, b in B,
   - **a.** $a + b$ is in B
   - **b.** $a \bullet b$ is in B
3. *Commutative laws*: For every a, b in B,
   - **a.** $a + b = b + a$
   - **b.** $a \bullet b = b \bullet a$
4. *Associative laws*: For every a, b, c in B,
   - **a.** $(a + b) + c = a + (b + c) = a + b + c$
   - **b.** $(a \bullet b) \bullet c = a \bullet (b \bullet c) = a \bullet b \bullet c$
5. *Identities*:
   - **a.** There exists an identity element with respect to $\{+\}$, designated by 0, such that $a + 0 = a$ for every a in B.
   - **b.** There exists an identity element with respect to $\{\bullet\}$, designated by 1, such that $a \bullet 1 = a$ for every a in B.
6. *Distributive laws*: For every a, b, c in B,
   - **a.** $a + (b \bullet c) = (a + b) \bullet (a + c)$
   - **b.** $a \bullet (b + c) = (a \bullet b) + (a \bullet c)$
7. *Complement*: For each a in B, there exists an element a' in B (the complement of a) such that
   - **a.** $a + a' = 1$
   - **b.** $a \bullet a' = 0$

Note: We use A' and $\overline{A}$ interchangeably to represent complement in this chapter.

It is easy to verify that the set $B = \{0, 1\}$ and the logical operations OR, AND, and NOT satisfy all the axioms of a Boolean algebra. Simply substitute 0 and 1 for a and b, OR for +, AND for $\bullet$, and NOT for ', and show that the expressions are true. For example, to verify the commutative law for +:

$$0 + 1 = 1 + 0 \qquad\qquad 0 \bullet 1 = 1 \bullet 0$$
$$1 = 1\ \surd \qquad\qquad 0 = 0\ \surd$$

A *Boolean function* uniquely maps some number of inputs over the set $\{0, 1\}$ into the set $\{0, 1\}$. A *Boolean expression* is an algebraic statement containing Boolean variables and operators. A theorem in Boolean algebra states that any Boolean function can be expressed in terms of AND, OR, and NOT operations. For example, in Section 1.3.3 we saw one way to map a truth table into a Boolean expression in the *sum* (OR) of *products* (AND) form.



X	Y
0	1
1	0

Figure 2.5 Alternative Representations of complement.

In fact, there are other ways to represent the function using new logical operations, to be introduced next. These operations are interesting because they are easier to implement with real transistor switches.

Boolean Operations Revisited Let's review the elementary Boolean operations and how these are represented as gates and truth tables. Figure 2.5, Figure 2.6, and Figure 2.7 summarize the representations for the COMPLEMENT/NOT, AND, and OR operations, respectively.

Take the Boolean expression . A version of the expression with parentheses would be . Each pair of parentheses represents the expression generated by a single gate. Thus, the circuit is built up through a set of intermediate results, from the inside out:

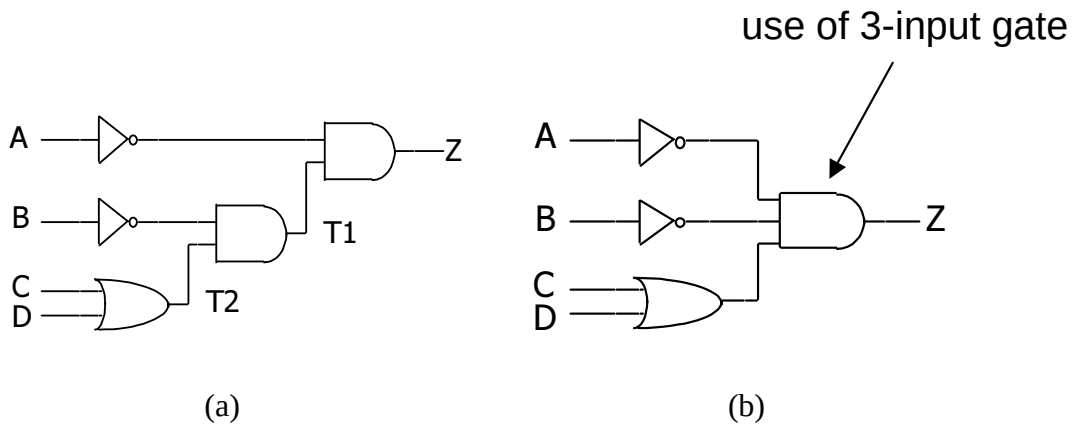


Figure 2.8 Two equivalent gate-level implementations.

The gate-level implementation is shown in Figure 2.8(a), using two-input gates. The primitive gates need not be limited to two inputs, however. Figure 2.8(b) shows the same circuit implemented using a three-input AND gate. These implementations are equivalent because of the associative law of Boolean algebra.

Each appearance of a variable or its complement in an expression is called a *literal*. In the preceding expression, we can see that there are four variables and four literals. The following expression has 10 literals but only three variables (*A*, *B*, and *C*): $Z = A\bar{B}C + AB + \bar{A}B\bar{C} + B\bar{C}$. Each literal represents the connection of a variable or its complement to a unique gate input. Later, we will use literals as a rough measure of the complexity of a Boolean functions.

2.2.2 Theorems of Boolean Algebra

Boolean algebra provides the foundation for all of the simplification techniques we shall discuss. Based on the Boolean laws of Section 2.1, we can prove additional theorems that can be used as tools to simplify Boolean expressions. For example, if E_1 and E_2 are two expressions for the same Boolean function, we say that E_2 is simpler than E_1 if it contains fewer literals. This usually (but not always) means that the simpler expression will also contain fewer Boolean operations.

Duality Before we provide a tabulation of useful laws and theorems, it is important to describe the concept of *duality*. Every Boolean expression has a dual. It is derived from the original by replacing AND operations by OR operations and vice versa, and replacing constant logic 0's by logic 1's and vice versa, while leaving

the literals unchanged. It is a fundamental theorem of Boolean algebra, which we do not prove here, that any statement that is true about a Boolean expression is also true for its dual. Once we discover a useful theorem for simplifying a Boolean expression, we obtain its dual as a bonus. For example, the *dual* of the Boolean theorem $X + 0 = X$, written $(X + 0)^D$, is the theorem $X \bullet 1 = X$.

Useful Theorems The following is a list of frequently used laws and theorems of Boolean algebra. Some are generalized from Section 2.2.1. The second column shows the duals of the expression in the first column.
Operations with 0 and 1:

1. $X + 0 = X$ **1D.** $X \bullet 1 = X$

2. $X + 1 = 1$ **2D.** $X \bullet 0 = 0$

Idempotent theorem:

3. $X + X = X$ **3D.** $X \bullet X = X$

Involution theorem:

4. $(X')' = X$

Theorem of complementarity:

5. $X + X' = 1$ **5D.** $X \bullet X' = 0$

Commutative law:

6. $X + Y = Y + X$ **6D.** $X \bullet Y = Y \bullet X$

Associative law:

7. $(X + Y) + Z = X + (Y + Z) = X + Y + Z$ **7D.** $(X \bullet Y) \bullet Z = X \bullet (Y \bullet Z) = X \bullet Y \bullet Z$

Distributive law:

8. $X \bullet (Y + Z) = X \bullet Y + X \bullet Z$ **8D.** $X + (Y \bullet Z) = (X + Y) \bullet (X + Z)$

Simplification theorems:

9. $X \bullet Y + X \bullet Y' = X$ **9D.** $(X + Y) \bullet (X + Y') = X$

10. $X + X \bullet Y = X$ **10D.** $X \bullet (X + Y) = X$

11. $(X + Y') \bullet Y = X \bullet Y$ **11D.** $(X \bullet Y') + Y = X + Y$

DeMorgan's theorem:

12. $(X + Y + Z + ...)' = X' \bullet Y' \bullet Z' \bullet \ldots$ **12D.** $(X \bullet Y \bullet Z \bullet ...)' = X' + Y' + Z' + \ldots$

13. $\{f(X_1, X_2, ..., X_n, 0, 1, +, \bullet)\}' = \{f(X_1', X_2', ..., X_n', 1, 0, \bullet, +)\}$

Duality:

14. $(X + Y + Z + ...)^D = X \bullet Y \bullet Z \bullet \ldots$ **14D.** $(X \bullet Y \bullet Z \bullet ...)^D = X + Y + Z + \ldots$

15. $\{f(X_1,X_2,...,X_n,0,1,+,\bullet)\}^D = f(X_1,X_2,...,X_n,1,0,\bullet,+)$

Theorem for multiplying and factoring:

16. $(X + Y) \bullet (X' + Z) = X \bullet Z + X' \bullet Y$ **16D.** $X \bullet Y + X' \bullet Z = (X + Z) \bullet (X' + Y)$

Consensus theorem:

17. $X \bullet Y + Y \bullet Z + X' \bullet Z = X \bullet Y + X' \bullet Z$ **17D.** $(X + Y) \bullet (Y + Z) \bullet (X' + Z)$

$= (X + Y) \bullet (X' + Z)$

The notation $f(X_1,X_2,...,X_n,0,1,+,\bullet)$ used in theorems 13 and 15 represents an expression in terms of the variables X_1, X_2, …, X_n, the constants 0, 1, and the Boolean operations + and •. Theorem 13 states succinctly that, in forming the complement of an expression, the variables are replaced by their complements—that is, 0 is replaced by 1 and 1 by 0, and + is replaced by • and • by +.

Since any of the listed theorems can be derived from the original laws shown in Section 2.1, there is no reason to memorize all of them. The first eight theorems are the most heavily used in simplifying Boolean expressions.

Verifying the Boolean Theorems Each of the theorems can be derived from the axioms of Boolean algebra. We can prove the first simplification theorem, sometimes called the *uniting theorem*, as follows:

$X \bullet Y + X \bullet Y'$	=	X?	
$X\,(Y + Y')$	=	X	distributive law (8)
$X\,(1)$	=	X	complementarity theorem (5)
X	=	X √	identity (1D)

As another example, let's look at the second simplification theorem:

$X + X \bullet Y$	=	X?	
$X \bullet 1 + X \bullet Y$	=	X	identity (1D)
$X\,(1 + Y)$	=	X	distributive law (8)
$X\,(1)$	=	X	identity (2)
X	=	X √	identity (1)

Example The Full Adder Carry-Out [FULL ADDER WAS ORIGINALLY INTRODUCED IN OLD CHAPTER 1. STILL THERE?] We can use the laws and theorems just introduced to verify the simplified expression for the full adder's carry-out function. The original expression, derived from the truth table, is

$$Cout = \overline{A}B\,Cin + A\overline{B}\,Cin + AB\,\overline{Cin} + AB\,Cin$$

The first step uses theorem 3, the idempotent theorem, to introduce a copy of the term $AB\,Cin$. Then we use the commutative law to rearrange the terms:

$$= \overline{A}B\,Cin + A\overline{B}\,Cin + AB\,\overline{Cin} + AB\,Cin + AB\,Cin$$

$$= \overline{A}B\,Cin + AB\,Cin + A\overline{B}\,Cin + AB\,\overline{Cin} + AB\,Cin$$

We next use the distributive law to factor out the common literals from the first two terms:

$$= (\overline{A} + A)B\,Cin + A\overline{B}\,Cin + AB\,\overline{Cin} + AB\,Cin$$

We apply the complementarity law:

$$= (1)B\,Cin + A\overline{B}\,Cin + AB\,\overline{Cin} + AB\,Cin$$

and the identity law:

$$= B\,Cin + A\overline{B}\,Cin + AB\,\overline{Cin} + AB\,Cin$$

We can repeat the process for the second and third terms. The steps are: (1) idempotent theorem to introduce a redundant term, (2) commutative law to rearrange terms, (3) distributive law to factor out common literals, (4) complementarity theorem to replace $(X + \overline{X})$ with 1, and (5) identity law to replace $1 \bullet X$ by X:

$$= B\,Cin + A\overline{B}\,Cin + AB\,\overline{Cin} + AB\,Cin + AB\,Cin$$

$$= B\,Cin + A\overline{B}\,Cin + AB\,Cin + AB\,\overline{Cin} + AB\,Cin$$

$$= B\,Cin + A(\overline{B} + B)Cin + AB\,\overline{Cin} + AB\,Cin$$

$$= B\,Cin + A(1)Cin + AB\,\overline{Cin} + AB\,Cin$$

The final simplification, using the distributive theorem, complementarity theorem, and identity law, proceeds similarly:

$$= B\,Cin + A\,Cin + AB(\overline{Cin} + Cin)$$

$$= B\,Cin + A\,Cin + AB(1)$$

$$= B\,Cin + A\,Cin + AB$$

This is exactly the reduced form of the expression we used in Chapter 1. Although it leads to a simpler expression, applying the rules of Boolean algebra in this fashion does not guarantee you will obtain the simplest expression. A more systematic approach will be introduced in Section 2.3.

2.2.3 Duality and DeMorgan's Law

Duality Duality is a very useful property of Boolean algebra. Recall that the dual of a Boolean expression is derived by replacing • by +, + by •, 0 by 1 and 1 by 0, while leaving the Boolean variables unchanged (laws 14 and 15 in the last section). Any theorem of Boolean algebra that is shown to be true implies that its dual is also true. In essence, duality is a *meta-theorem*, a theorem about theorems. While it is not a way to simplify expressions directly, it does allow you to derive new theorems from those you already know to help in the simplification process.

For example, the dual of the Unifying Theorem, $X \bullet Y + X \bullet Y' = X$, is $(X + Y) \bullet (X + Y') = X$. The proof of the dual follows step-by-step from the original, simply using the duals of the laws used in the original proof. Compare the following:

$(X + Y) \bullet (X + Y') = X?$

$X + (Y \bullet Y')$	=	X	distributive law (8D)
$X + 0$	=	X	complementarity theorem (5D)
X	=	$X\ \surd$	identity (1)

DeMorgan's Theorem DeMorgan's theorem gives a procedure for complementing a complex function. The complemented expression is formed from the original by replacing all literals by their complements; all 1's become 0's and vice versa, and ANDs become ORs and vice versa. This theorem indicates an interesting relationship between AND, OR, and their complements Not OR (NOR) and Not AND (NAND):

$$\overline{X + Y} = \overline{X} \bullet \overline{Y} \qquad \overline{X \bullet Y} = \overline{X} + \overline{Y}$$

Note that $\overline{X + Y} \neq \overline{X} + \overline{Y}$ and $\overline{X \bullet Y} \neq \overline{X} \bullet \overline{Y}$. In other words, NOR is the same as AND with complemented inputs while NAND is equivalent to OR with complemented inputs! This is easily seen to be true from the truth tables of Figure 2.9.

X	Y	X'	Y'	(X + Y)'	X' • Y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

X	Y	X'	Y'	(X • Y)'	X' + Y'
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

Figure 2.9 DeMorgan's law.

Let's use DeMorgan's theorem to find the complement of the following expression:

$$Z = \overline{A}\overline{B}\overline{C} + \overline{A}BC + A\overline{B}C + AB\overline{C}$$

Step by step, the complement is formed as follows:

$$\begin{aligned}\overline{Z} &= \overline{\overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + AB\overline{C}} \\ &= \overline{\overline{A}\overline{B}C} \bullet \overline{\overline{A}B\overline{C}} \bullet \overline{A\overline{B}\overline{C}} \bullet \overline{AB\overline{C}} \\ &= (A + B + \overline{C})(A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)\end{aligned}$$

Note that duality and DeMorgan's law are *not* the same thing. The procedure for producing the dual is similar, but the literals are not complemented during the process. Thus, the dual of NOR is NAND (and vice versa); the dual of OR is AND (and vice versa). Remember, any theorem that is true for an expression is also true for its dual. When the duality theorem is applied to a function the result is a function that is totally different from the orignal, when DeMorgan's theorem is applied to a function, the result is a function that is the complement of the original. For example if we have the expression:

$Z = \overline{A}B + C\overline{D}$

then the dual of Z is:

$Zdual = (\overline{A} + B)(C + \overline{D})$

while applying DeMorgan's theorem yields:

$\overline{Z} = (A + \overline{B})(\overline{C} + D)$

2.3 Realizing Boolean Formulas

2.3.1 Logic Gates (SSI)

There are other functions of two Boolean variables besides AND, OR, and NOT. In fact, there are 16 possible functions, the number of different ways you can write down the different choices of 0 and 1 for the four possible truth table rows. A truth table representation of the 16 functions is shown in Figure 2.10. The constant functions 0 and 1 and the functions X, $\overline{X}$, Y, $\overline{Y}$ $X \bullet Y$, and $X + Y$ represent only half of the possible functions. We now introduce the remaining Boolean operators.

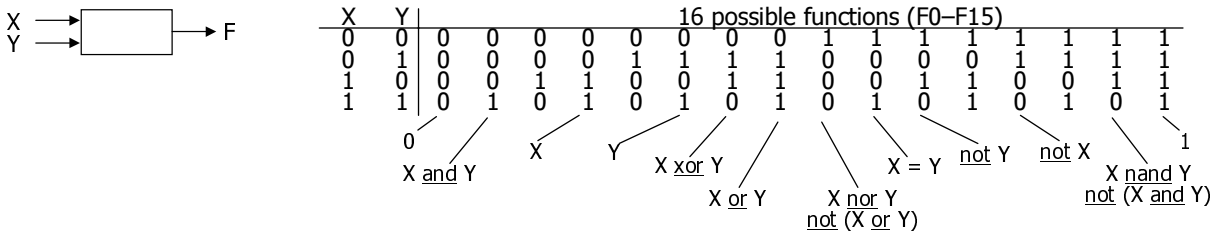


Figure 2.10 The 16 functions of two Boolean variables.

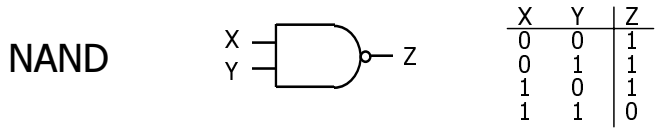


Figure 2.11 Gate and Truth Table Representations of NAND.

In addition to their different truth tables, each function has a different number of switches associated with its implementation For example, F_0 and F_{15}, constants 0 and 1 respectively, require no switches for their implementation at all. These logical values can be obtained by directly connecting to the reference low and high voltage signals. Implementation complexity: X (F_3) and Y (F_5) also require no switches, as they are simply direct-through connections to the appropriate input. $\overline{X}$ (F_{12}) and $\overline{Y}$ (F_{10}) are the complement functions of X and Y, and use two switches to implement an inverter. X NOR Y (F_4) and X AND Y (F_{14}) are implemented using only four switches. X OR Y (F_7) and X AND Y (F_1), on the other hand, require 6 switches, while X = Y (F_9) and X XNOR Y (F_6) demand 16 switches for their implementations. So while one way of implementing a given system might minimize the number of Boolean operations, it may not minimize the number of switches for its realization.

NAND and NOR Two of the most frequently encountered Boolean operators are NAND (not AND) and NOR (not OR). Their gate and truth table representations are shown in Figure 2.11. The NAND operation behaves as if an AND is composed with a NOT: it yields a logic 0 in the truth table rows where AND is a 1, and it yields a 1 in the rows where AND is 0. The gate representation is an AND gate with a small circle or "bubble" at its output, denoting negation.

If you take a close look at the truth table representation in Figure 2.11 and compare it to Figure 2.6, you will see that it looks like an AND function with the true and false outputs reversed. NAND is true when either *X* is 0 or *Y* is 0. Alternatively, it is false when *X* and *Y* are both true.

NOR behaves in a similar fashion, but now with respect to OR. Once again the truth table outputs are complemented, and we draw the NOR gate as an OR gate

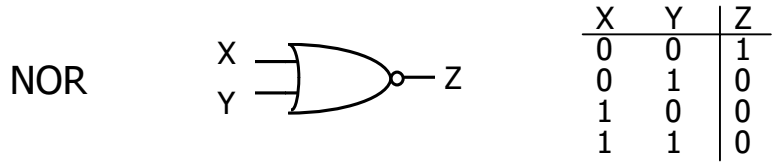


Figure 2.12 Representations of NOR.

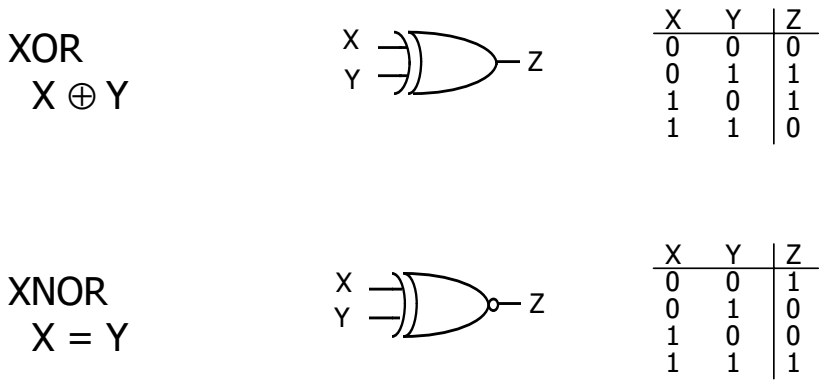


Figure 2.13 Representations of the XOR and XNOR operations.

with a bubble at the output. Both X and Y must be zero to force the output to be true.

NAND and NOR gates far outnumber AND and OR gates in a typical digital design, even though these functions are less intuitive, for the simple reason that they can be implemented in fewer switches than AND and OR gates.

Since any Boolean expression can be represented in terms of AND, OR, and NOT gates, it is hardly surprising that the same statement can be made about NAND, NOR, and NOT gates. In fact, NOT gates are superfluous: if you carefully examine the truth tables of Figures 2.11 and 2.12, you'll see that NAND and NOR act like NOT when both inputs are identically 0 or 1. We shall see an efficient method for mapping Boolean expressions into NAND and NOR logic in Section 2.2.2 [CHECK REFERENCE].

XOR/XNOR (Equality) This leaves six functions still unnamed in Figure 2.10. Two of these, frequently of use, are exclusive OR (XOR, also known as the *inequality gate* or *difference function*) and exclusive NOR (XNOR, also known as the *equality gate* or *coincidence function*). Their truth tables and gate representations are given in Figure 2.13. XOR is true when its inputs differ in value. XNOR is true when its inputs coincide in value. The Boolean operator for XOR is $\oplus$; XNOR is usually represented as the complement of XOR. As with any Boolean function, these can be implemented in terms of AND, OR, and NOT operations:

XOR: $X \oplus Y = X\overline{Y} + \overline{X}Y$

XNOR: $\overline{X \oplus Y} = \overline{X}\overline{Y} + XY$

If you examine the truth table of Figure 2.13(a), you can see that XOR is precisely the function needed to implement the half adder sum of Chapter 1 [REFERENCE] and XNOR directly implements the Equivalence function of Section 2.1.2.

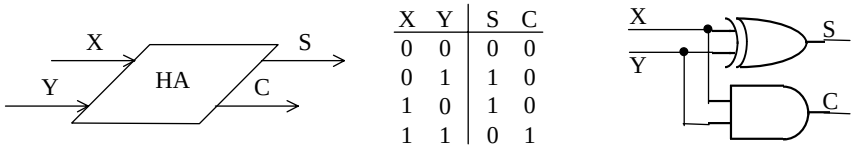


Figure 2.14 Half Adder Function Implemented Via Interconnected Gates

Implication The remaining four functions are based on a Boolean operator called *implication*. X implies Y (written $X \Rightarrow Y$) is true under two conditions: either X is false or both X and Y are true. The four remaining functions become $X \Rightarrow Y$, $Y \Rightarrow X$, NOT ($X \Rightarrow Y$), and NOT ($Y \Rightarrow X$). These are not commonly found as primitives readily available for realizing digital systems, so they won't be of much use to you.

2.3.2 Logic Blocks and Hierarchy

Just as a program can be constructed from simpler subroutines and a logic function can be built up by wiring primitive gates, even the most complex logic function can be constructed by composing more primitive functions.

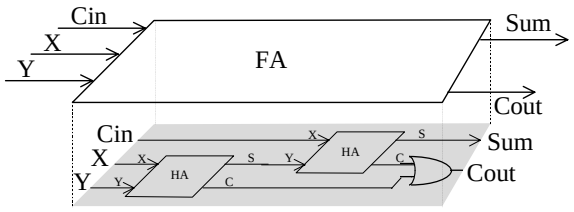
In most integrated circuit technologies, libraries of prepackaged functions are made available to the designer. The library components include all of the major logic gates, in the form of 2-, 3-, and possibly higher number of inputs, as well as more complex function constructed from many logic gates. Examples of these are the half adder, full adder, and multi-bit adder functions.

Example The Half Adder (HA) function takes two binary inputs (X, Y), and generates their sum (S) and carryout (C). Figure 2.14 represents the half adder as a block diagram, a black box with inputs and outputs. Associated with it is an associated specification, which in this case is the truth table that describes the function's input/output behavior. The half adder's sum output can be implemented by the XOR gate, and the carry by the AND gate.

A multiple-bit adder can be built by up by wiring together single bit adders. The stage that computes the sum of the bits in the first, lowest order position passes its carry-out to the carry-in of the next higher order adder. Unfortunately, the half adder function, since it doesn't have a carry-in input, cannot be used in this way. This requires a function called the Full Adder (FA).

The FA block diagram is shown in Figure 2.15: three inputs, X, Y, and Carry-In (Cin), and two outputs Sum and Carry-Out (Cout). The full adder functionality is described by the truth table in the figure, and can be implemented by the cascaded half adder as shown.

It is a fair question as to how this particular combination of half adder building blocks and an OR gate realizes the full adder function. It comes from a careful examination of the full adder truth table. Consider the case when Cin = 0, basically the bottom half of the truth table. The function for SUM is exactly like the HA. When Cin = 1, the top half of the truth table, the SUM output is exactly the



Cin	X	Y	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 2.15. Full Adder Function Implemented as Cascaded Half Adders

complement of the HA's S output. The XOR gate's behavior is such that when one input is 0, the output is the same as the other input. And when one input is 1, the output is the complement of the other input. Connecting one XOR input to the FA carry-in input, and the other to the sum of X and Y computed by a first stage HA, gives us exactly the Sum output behavior we need. But this is exactly what the HA does to its two inputs!

Implementing the FA carry-out is a little more complicated. The HA C output is almost the same as the FA Cout output: it is one whenever X = Y = 1, independent of Cin. It is also one whenever Cin is one and it is not the case that both X and Y are both zero. But these cases are exactly ones that would be asserted for a HA whose inputs were Cin and (X plus Y). The OR gate combines the first case with this second case. You can check that this composition implements the Full Adder by working through the truth table for each of the possible truth table input combinations.

Now the two bit adder can be constructed by interconnecting two full adders. The wiring is shown in Figure 2.16. The carry-in for the low order stage is connected to logic 0. Its carry-out is connected to the carry-in of the high order stage.

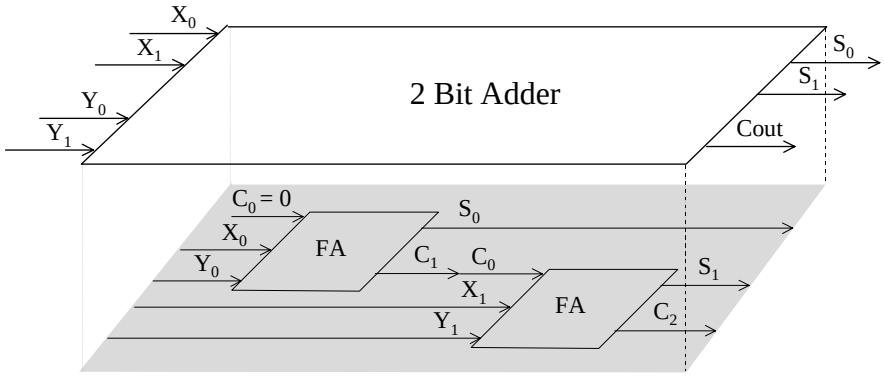


Figure 2.16 Two Bit Binary Adder Composed from Two Full Adders

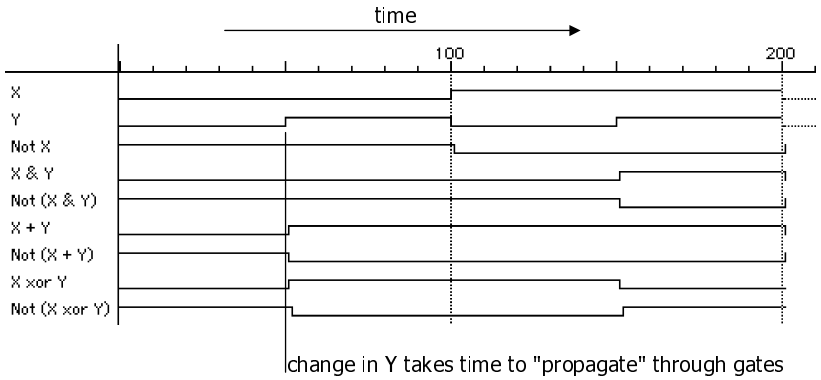


Figure 2.17 Timing waveforms for NOT, AND, NAND, OR, NOR, XOR, XNOR.

2.3.3 Time Behavior and Waveforms

Logic gates do not operate infinitely fast. When the inputs change, it takes some time for the these changes to be reflected in the gate's output, as the electrical signals propagate through the interconnected switches.

A waveform representation is a good way to represent signal propagation over time. The X-axis represents the time steps. The Y axis shows the logical value of various points in the circuit. These *probe* points can be an output of the function, or the value on any internal wire of the circuit.

For ease of discussion, we consider the delay through any gate as taking exactly one time unit. This is called the *unit delay model*, and it really is a simplifying assumption. Real gates rarely exhibit such uniformly simple performance. For example, an AND gate is usually implemented as a NAND gate composed with a NOT gate. So it makes sense that an AND gate should be slower to change its output in response to changes in its inputs than a simpler, more primitive NAND gate.

Timing waveforms for the common logic gates, AND, OR, NAND, NOR, XOR, and XNOR, are shown in Figure 2.17. The figure assumes a single-time-unit gate delay in computing new outputs from inputs.

Example Need to show examples of HA delay, FA delay, 2-bit adder delays (using different scale than 2.17 with unit delays for gates but inputs changing every 10 time steps).

2.3.4 Minimizing the Number of Gates and Wires

Logic minimization uses a variety of techniques to obtain the simplest gate-level implementation of a Boolean function. But simplicity depends on the metric we use. We examine these metrics in this subsection.

Time and Space Trade-Offs One way to measure the complexity of a Boolean function is to count the number of literals it contains. Literals measure the amount of wiring needed to implement a function. For electrical and packaging reasons, gates in a given technology typically have a limited number of inputs. While two-, three-, and four-input gates are common, gates with more than eight or nine inputs are rather rare. Thus, one of the primary reasons for performing logic minimization is to reduce the number of literals in the expression of the function, thus reducing the number of gate inputs.

An alternative metric is the number of gates, which measures circuit area. There is a strong correlation between the number of gates in a design and the number of components, whether library modules or integrated circuit packages, needed for its implementation. The simplest design to manufacture is often the one with the fewest gates, not the fewest literals.

A third metric is the number of cascaded levels of gates. Reducing the number of logic levels reduces overall delay, as there are fewer gate delays on the path from inputs to outputs. However, putting a circuit in a form suitable for minimum delay rarely yields an implementation with the fewest gates or the simplest gates. It is not possible to minimize all three metrics at the same time.

The traditional minimization techniques you will study in this chapter emphasize reducing delay at the expense of adding more gates. Newer methods, covered in the next chapter, allow a trade-off between increased circuit delay and reduced gate count.

Example To illustrate the trade-offs just discussed, consider the following three-variable Boolean function:

$$Z = \overline{A}\overline{B}C + \overline{A}BC + A\overline{B}C + AB\overline{C}$$

The truth table for this function is shown in Figure 2.18(a). You would probably implement the function directly from the preceding equation, using three NOT gates, four 3-input AND gates, and a single 4-input OR gate. This is called a *two-level implementation*, with variables and their complements at the zeroth level, AND gates at the first level, and an OR gate at the second level.

You could implement the same truth table with fewer gates. An alternative two-level implementation is given in Figure 2.18(b) as function Z_1:

$$Z_1 = AB\overline{C} + \overline{A}C + \overline{B}C$$

It uses the same number of inverters and OR gates but only three AND gates. The original function has 12 literals. This alternative has only seven, thus reducing the wiring complexity.

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

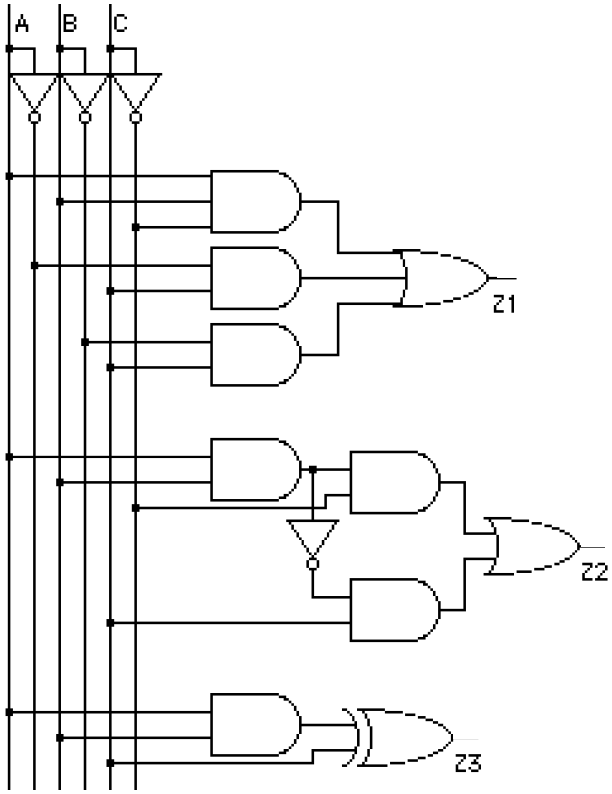


Figure 2.18 Alternative realizations of *Z*.

The implementation of function Z_2 is called *multilevel*:

$$Z_2 = T\overline{C} + \overline{T}C$$

$$T = AB$$

The longest path from an input to an output passes through four gates. This contrasts with three gate delays in the two-level functions. In terms of gate counts, the circuit uses two rather than three inverters and only two-input AND and OR gates. Here you can see a trade-off between gate count and performance: Z_2 uses simpler gates, but it is not as fast as Z_1.

Z_3 shows a third realization that uses an XOR gate:

$$Z_3 = (AB) \oplus C$$

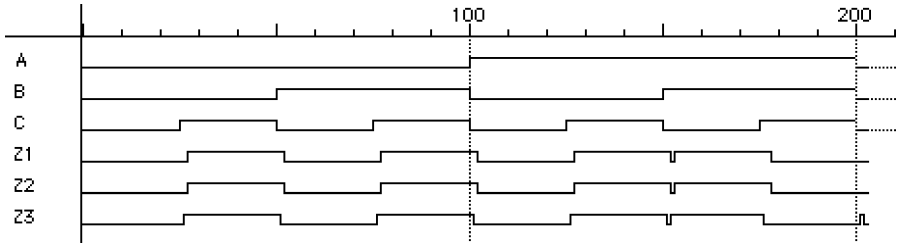


Figure 2.19 Waveform behavior of three implementations of the truth table of Figure 2.18(a).

XOR is sometimes called a *complex gate*, because you normally implement it by combining several NAND or NOR gates (see Exercise 2.12). Although this implementation has the lowest gate count, it is also likely to have the worst signal delay. An XOR gate tends to be slow compared with the implementations for *Z* based on simple AND and OR gates.

Figure 2.19 shows the timing waveforms for the three circuit alternatives, assuming (somewhat unrealistically) a single time unit delay per gate. All have equivalent behavior, although they exhibit slightly different propagation delays. All three circuits show a glitch on the transition *ABC* = 1 0 1 to *ABC* = 1 1 0.

2.3.5 Application: 7-Segment Decoder

We have already introduced the 7-Segment Display and simple logic for displaying a binary digit in Section 2.1.2 and Figure 2.4. In this subsection, we will generalize the combinational logic to handle the case of the decimal digits 0 through 9. The input to the system is four bits in a Binary Coded Decimal form, where decimal 0 is represented by the four binary functions ABCD = 0000, 1 is represented by ABCD = 0001, 2 by 0010, 3 by 0011, through 9 = 1001. We assume that the bit patterns 1010, 1011, 1100, 1101, 1110, 1111 are never encountered in practice (we call such inputs *don't cares*, and we will see how to exploit them in Section X).

Figure 2.20 shows a block diagram of the decoder, the display element, and the way the segments should be driven to represent the appropriate digit. Figure 2.21 shows the truth table for the system.

Let's specify the Boolean logic for just one of the decoder's outputs: C2. This particular segment is illuminated for every valid input combination except for digit

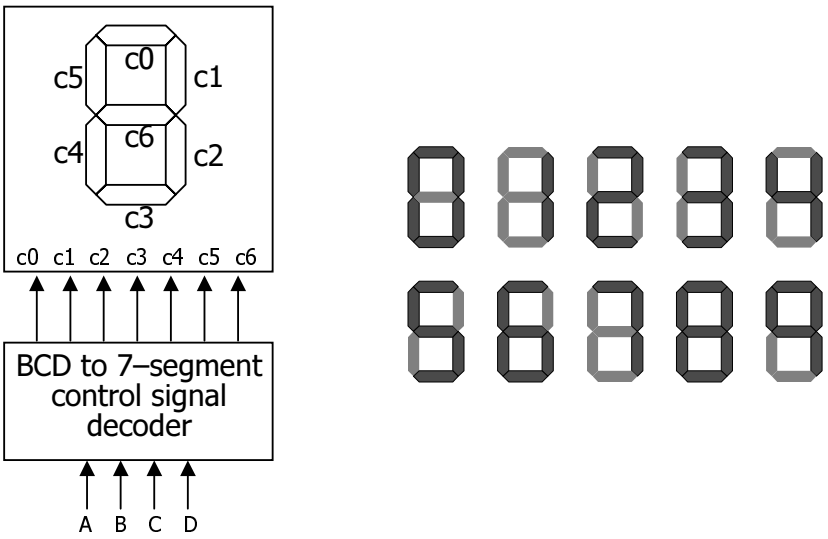


Figure 2.20 Decimal Digital Display System

A	B	C	D	C_0	C_1	C_2	C_3	C_4	C_5	C_6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Figure 2.21 Digital Display System Truth Table

2 (0010). The Boolean expression for C_2 is all of the input combinations that cause that segment to be illuminated ored together:

$$C_2 = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D + \overline{A}\overline{B}CD + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}BC\overline{D} + \overline{A}BCD + A\overline{B}\overline{C}\overline{D} + \overline{A}BC\overline{D}$$

The Boolean function in this form is quite complicated: 36 literals, nine 4-input AND gates, and one 9-input OR gate. Fortunately, its complexity can be significantly reduced by the right use of simplification theorems. By rearranging terms, introducing redundant terms, and factoring, we can rewrite C_2 as follows:

$$C_2 = \overline{A}B(\overline{C}\overline{D} + \overline{C}D + \overline{C}D + CD) + \overline{B}\overline{C}(\overline{A}\overline{D} + \overline{A}D + A\overline{D} + AD) + \overline{A}D(\overline{B}\overline{C} + \overline{B}C + B\overline{C} + BC)$$

Successive use of the Unifying theorem and the Complementarity theorem for the expressions in the parentheses reduces them to the constant 1. The resulting simplification is:

$$C_2 = \overline{A}B + \overline{B}\overline{C} + \overline{A}D$$

This expression has only 6 literals, three 2-input AND gates, and one 3-input OR gate. That is a considerable savings in terms of all of the relevant metrics: less wires, fewer gates, and simpler gates! If we take advantage of the don't care elements of the truth table, we could simplify the expression even further, as we will see in Section 2.4.2.

2.4 Two-Level Formulas

You now know that there are many gate-level implementations with the same truth table behavior. In this section, you will learn the methods for deriving a reduced gate-level implementation of a Boolean function in two-level form. This usually yields circuits with minimum delay, although gate counts are typically not minimized.

2.4.1 Canonical Forms

To compare Boolean functions when expressed in algebraic terms, it is useful to have a standard form with which to represent the function. This standard term is called a *canonical form*, and it is a unique algebraic signature of the function. You will frequently encounter two alternative forms: *sum of products* and *product of sums*. We introduce these next.

Sum of Products You have already met the *sum of products* form in Section 1.4.1. It is also sometimes known as a *disjunctive normal form* or *minterm expansion*. A sum of products expression is formed as follows. Each row of the

A	B	C	F	F'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Figure 2.22 Sample truth table.

A	B	C	minterms	
0	0	0	A'B'C'	m0
0	0	1	A'B'C	m1
0	1	0	A'BC'	m2
0	1	1	A'BC	m3
1	0	0	AB'C'	m4
1	0	1	AB'C	m5
1	1	0	ABC'	m6
1	1	1	ABC	m7

Figure 2.23 Shorthand notation for minterms of three variables.

truth table in which the function takes on the value 1 contributes an ANDed term, using the asserted variable if there is a 1 in its column for that row or its complement if there is a 0. These are called *minterms*. Technically, a minterm is defined as an ANDed product of literals in which each variable appears exactly once in either true or complemented form, but not both. The minterms are then ORed to form the expression for the function. The minterm expansion is unique because it is deterministically derived from the truth table.

Figure 2.22 shows a truth table for a function and its complement. The minterm expansions for F and $\overline{F}$ are

$$F = \overline{A}\overline{B}C + \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

$$\overline{F} = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}\overline{C}$$

We can write such expressions in a shorthand notation using the binary number system to encode the minterms. Figure 2.23 shows the relationship between the truth table row and the numbering of the minterm. Note that the ordering of the Boolean variables is critical in deriving the minterm index. In this case, A determines the most significant bit and C is the least significant bit. You can write the shorthand expression for F and $\overline{F}$ as

$$F(A,B,C) = \Sigma\, m(1,3,5,6,7) = m_1 + m_3 + m_5 + m_6 + m_7$$

$$\overline{F}(A,B,C) = \Sigma\, m(0,2,4) = m_0 + m_2 + m_4$$

where m_i represents the i^{th} minterm. In Figure 2.23, the minterms that represent the *cover* of the function of Figure 2.22 are shaded. The function is true if the combination of input values for inputs A, B, and C lies within any one of the shaded regions, hence, a disjuctive normal form (OR) because the function is true if the inputs fall within the box labelled m1, or the box labelled m3, or ... .

The indices generalize for functions of more variables. For example, if F is defined over the variables A, B, C, then m_3 (011_2) is the minterm $\overline{A}BC$. But if F is defined over A, B, C, D, then m_3 (0011_2) is $\overline{A}\overline{B}CD$.

The minterm expansion is not guaranteed to be the simplest form of the function, in terms of the fewest literals or terms, nor is it likely to be. You can further reduce the expression for F by applying Boolean algebra:

$$\begin{aligned} F(A, B, C) &= \overline{A}\overline{B}C + \overline{A}BC + A\overline{B}C + ABC + AB\overline{C} \\ &= (\overline{A}\overline{B} + \overline{A}B + A\overline{B} + AB)C + AB\overline{C} \\ &= ((\overline{A} + A)(\overline{B} + B))C + AB\overline{C} \\ &= C + AB\overline{C} \\ &= AB\overline{C} + C \\ &= AB + C \end{aligned}$$

The one step you may find tricky is the last one, which applies rule 11D, $(X \bullet \overline{Y}) + Y = X + Y$, substituting AB for X and C for Y.

AB and *C* are called *product terms*: ANDed strings of literals containing a subset of the possible Boolean variables or their complements. For *F* defined over the variables *A*, *B*, and *C*, $\overline{A}BC$ is a minterm and a product term, but *AB* is only a product term.

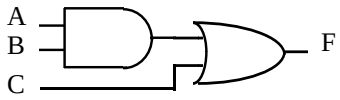


Figure 2.24 Two-level AND-OR gate-level implementation.

The minimized gate-level implementation of *F* is shown in Figure 2.24. Each product term is realized by its own AND gate. The product term *A* is the degenerate case of a single literal. No AND gate is needed to form this term. The product terms' implementations are then input to a second-level OR gate. The sum of products form leads directly to a two-level realization.

We can repeat the simplification process for $\overline{F}$, but DeMorgan's theorem gives us a good starting point for applying Boolean simplification:

$$\overline{F} = \overline{(AB + C)} = (\overline{A} + \overline{B})\overline{C} = \overline{A}\,\overline{C} + \overline{B}\,\overline{C}$$

Although this procedure is not guaranteed to obtain the simplest form of the complement, it does so in this case.

Product of Sums The involution theorem states that the complement of a Boolean expression's complement is the expression itself. By using DeMorgan's theorem twice, we can derive a second canonical form for Boolean equations. This form is called the *product of sums* and sometimes the *conjunctive normal form* or *maxterm expansion*.

The procedure for deriving a product of sums expression from a truth table is the logical dual of the sum of products case. First, find the rows of the truth table where the function is 0. A *maxterm* is defined as an ORed sum of literals in which each variable appears exactly once in either true or complemented form, but not both. We form a maxterm by ORing the uncomplemented variable if there is a 0 in its column for that row, or the complemented variable if there is a 1 there. This is exactly opposite to the way we formed minterms. There is one maxterm for each 0 row of the truth table; these are ANDed together at the second level.

The products of sums for the functions *F* and $\overline{F}$ of Figure 2.22 are

$$F = (A + B + C)(A + \overline{B} + C)(\overline{A} + B + C)$$

$$\overline{F} = (A + B + \overline{C})(A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)(\overline{A} + \overline{B} + \overline{C})$$

A	B	C	maxterms	
0	0	0	A+B+C	M0
0	0	1	A+B+C'	M1
0	1	0	A+B'+C	M2
0	1	1	A+B'+C'	M3
1	0	0	A'+B+C	M4
1	0	1	A'+B+C'	M5
1	1	0	A'+B'+C	M6
1	1	1	A'+B'+C'	M7

Figure 2.25 Maxterm shorthand for a function of three variables.

Once again, we often use a shorthand notation. Figure 2.25 shows the relationship between maxterms and their shorthand form. We can write *F* and $\overline{F}$ as

$$F(A, B, C) = \Pi M(0,2,4) = M_0 \bullet M_2 \bullet M_4$$

$$\overline{F}(A, B, C) = \Pi M(1,3,5,6,7) = M_1 \bullet M_3 \bullet M_5 \bullet M_6 \bullet M_7$$

where M_i is the i^{th} maxterm. In Figure 2.25, the maxterms that represent the *cover* of the function of Figure 2.22 are shaded. Each maxterm is true for every combination except the one for which it is numbered. There are three different shaded regions (with different indentation). The first corresponds to M0, the second (broken into two pieces) corresponds to M2 and the third, similarly, corresponds to M4. The function is true if the combination of input values for inputs A, B, and C lies within all of the shaded regions, hence, a conjuctive normal form because the function is true if the inputs fall within the region for M0 and the region for M2 and the region for M4.

Interestingly, the maxterm expansion of F could have been formed directly by applying DeMorgan's theorem to the minterm expansion of $\overline{F}$:

$$\overline{F} = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}\overline{C}$$
$$\overline{\overline{F}} = \overline{(\overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}\overline{C})}$$
$$F = (A + B + C)(A + \overline{B} + C)(\overline{A} + B + C)$$

Of course, the same is true for deriving the minterm form of F from the maxterm form of $\overline{\ }$:

$$\overline{F} = (A + B + \overline{C})(A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)(\overline{A} + \overline{B} + \overline{C})$$
$$\overline{\overline{F}} = \overline{(A + B + \overline{C})(A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)(\overline{A} + \overline{B} + \overline{C})}$$
$$F = \overline{A}\overline{B}C + \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

It is easy to translate a product of sums expression into a gate-level realization. The zeroth level forms the complements of the variables if they are needed to realize the function. The first level creates the individual maxterms as outputs of OR gates. The second level is an AND gate that combines the maxterms.

We can find a minimized product of sums form by starting with the minimized sum of products expression of $\overline{F}$. To complement this expression, we use DeMorgan's theorem:

$$\overline{F} = \overline{A}\overline{C} + \overline{B}\overline{C}$$
$$\overline{\overline{F}} = \overline{\overline{A}\overline{C} + \overline{B}\overline{C}}$$
$$F = (A + C)(B + C)$$

Figure 2.26 shows the four different gate-level implementations for F discussed so far: canonical sum of products (F_1), minimized sum of products (F_2), canonical product of sums (F_3), and minimized product of sums (F_4). In terms of gate counts, the product of sums canonical form is more economical than the sum of products canonical form. But the minimized sum of products form uses fewer gates than the minimized product of sums form. Depending on the function, one or the other of these forms will be better for implementing the function.

To demonstrate that the implementations are equivalent, Figure 2.27 shows the timing waveforms for the circuits' responses to the same inputs. Except for short-duration glitches in the waveforms, their shapes are identical.

Conversion Between Canonical Forms We can place any Boolean function in one of the two canonical forms, sum of products or product of sums. It is easy to map an expression in one canonical form into the other. The procedure, using the shorthand notation we already introduced, is summarized here:

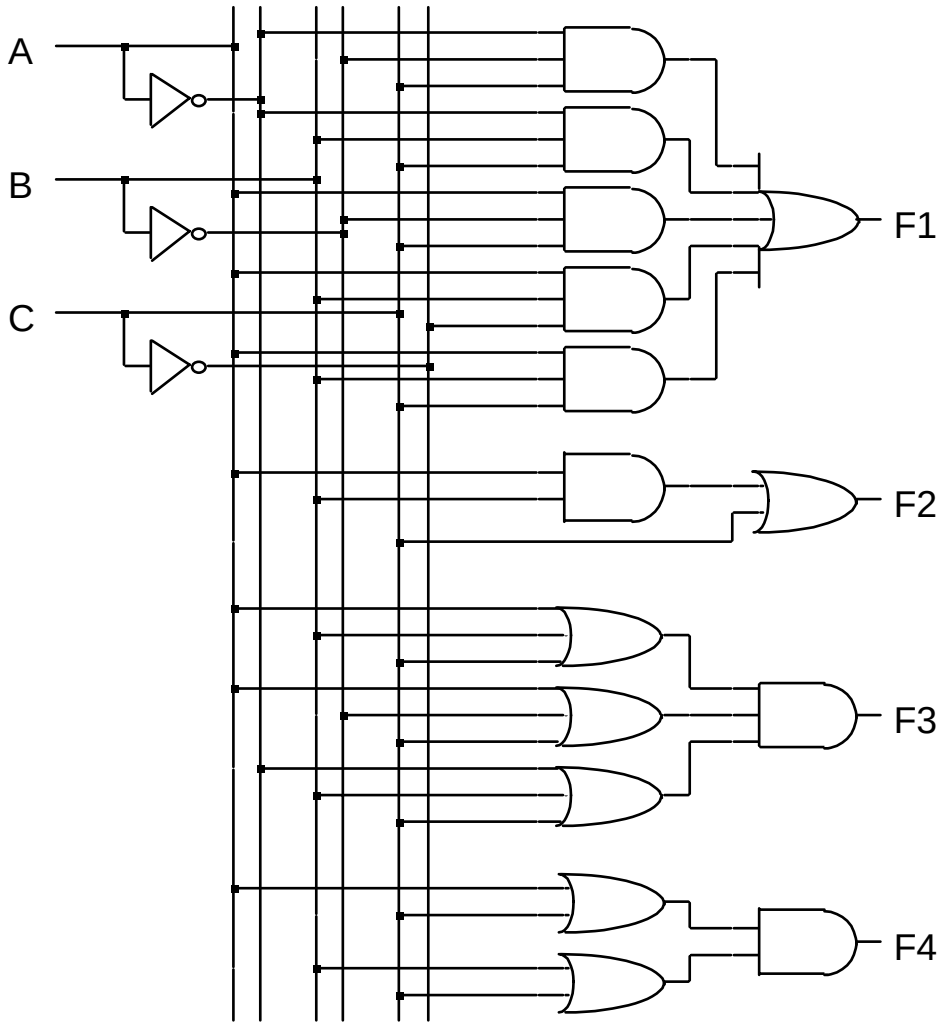


Figure 2.26 Four implementations of *F*.

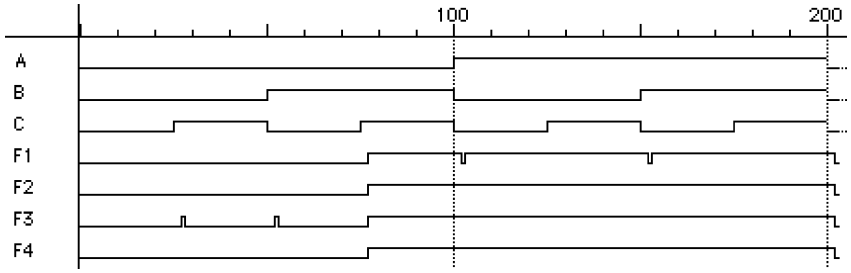


Figure 2.27 Timing waveforms for the four implementations of *F*.

1. To convert from the minterm expansion to the maxterm expansion, you rewrite the minterm shorthand notation to maxterm shorthand, replacing the term numbers with those not used in the minterm list. This is equivalent to applying DeMorgan's theorem to the complement of the function in minterm form.

Example: $F(A,B,C) = \Sigma\, m(1,3,5,6,7) = \Pi\, M(0,2,4)$

2. To convert from the maxterm expansion to the minterm expansion, you rewrite the maxterm shorthand notation to minterm shorthand, replacing term numbers with those not used in the maxterm list. This is equivalent to applying DeMorgan's theorem to the complement of the function in maxterm form.

Example: $F(A,B,C) = \Pi\, M(0,2,4) = \Sigma\, m(1,3,5,6,7)$

3. To obtain the minterm expansion of the complement, given the minterm expansion of the function, you simply list the minterms not in *F*. The same procedure works for obtaining the maxterm complement of a function expressed in maxterm form.

Example:

$$F(A,B,C) = \Sigma\, m(1,3,5,6,7) \qquad F(A,B,C) = \Pi\, M(0,2,4)$$
$$\overline{\ }(A,B,C) = \Sigma\, m(0,2,4) \qquad \overline{F}(A,B,C) = \Pi\, M(1,3,5,6,7)$$

4. To obtain the maxterm expansion of the complement, given the minterm expansion of the function, you simply use the same maxterm numbers as used in *F*'s minterm expansion. The same procedure applies if a minterm expansion of the complement is to be derived from the maxterm expansion of the function.

Example:

$$F(A,B,C) = \Sigma\, m(1,3,5,6,7) \qquad F(A,B,C) = \Pi\, M(0,2,4)$$
$$\overline{F}(A,B,C) = \Pi\, M(1,3,5,6,7) \qquad \overline{F}(A,B,C) = \Sigma\, m(0,2,4)$$

2.4.2 Incompletely Specified Functions

We have assumed that we must define an n-input function on all of its 2^n possible input combinations. This is not always the case. Often we have some flexibility in the specification of a function. Making this flexibility explicit permits optimization procedures to take that flexibility into account when trying to find the smallest or fastest circuit to implement a function. We study the case of incompletely specified functions in this subsection.

Example Incompletely Specified Functions Let's consider a logic function that takes as input a binary-coded decimal (BCD) digit. *BCD digits* are decimal digits, in the range 0 through 9, that are represented by four-bit binary numbers, using the combinations 0000_2 (0) through 1001_2 (9). The other combinations, 1010_2 (10) through 1111_2 (15), should never be encountered. It is possible to simplify the Boolean expressions for the function if we assume that we *do not care* about its behavior in these "out of range" cases.

Figure 2.28 shows the truth table for a BCD increment by 1 circuit. Each BCD number is represented by four Boolean variables, *A B C D*. The output of the incrementer is represented by four 4-variable Boolean functions, *W X Y Z*.

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

Figure 2.28 Truth table for BCD increment by 1.

The output functions have the value "X" for each of the input combinations we should never encounter. When used in a truth table, the value X is often called a *don't care*. Do not confuse this with the value X reported by many logic simulators, where it represents an undefined value or a *don't know*. Any actual implementation of the circuit will generate some output for the don't-care cases. When used in a truth table, an X simply means that we have a choice of assigning a 0 or 1 to the truth table entry. We should choose the value that will lead to the simplest implementation.

To see that don't cares eventually are replaced by some logic value, let's consider the BCD incrementer truth table. The function *Z* looks as if it could be realized quite simply as the function $\overline{D}$. If we choose to implement *Z* in this way, the Xs will be replaced by real logic values. Since the inputs 1010_2 through 1111_2 will never be encountered by the operational circuit, it shouldn't matter which values we assign to those truth table rows. We choose an assignment that makes the implementation as simple as possible.

Example Decimal Digit to Seven Segment Display Decoder Figure 2.28 shows a revised truth table for Seven Segment Display Decoder that takes full advantage of the fact that the input bit patterns 1010 through 1111 should never be presented to the system.

Don't Cares and the Terminology of Canonical Forms In terms of the standard Σ and Π notations, minterms or maxterms assigned a don't care are written as d_i or D_i, respectively. Thus the canonical form for Z is written as:

$$Z = m_0 + m_2 + m_4 + m_6 + m_8 + d_{10} + d_{11} + d_{12} + d_{13} + d_{14} + d_{15}$$

$$Z = M_1 \bullet M_3 \bullet M_5 \bullet M_7 \bullet M_9 \bullet D_{10} \bullet D_{11} \bullet D_{12} \bullet D_{13} \bullet D_{14} \bullet D_{15}$$

Similarly, C_2 can be written as:

$$Z = m_0 + m_1 + m_3 + m_4 + m_5 + m_6 + m_7 + m_8 + m_9 + d_{10} + d_{11} + d_{12} + d_{13} + d_{14} + d_{15}$$

$$Z = M_2 \bullet D_{10} \bullet D_{11} \bullet D_{12} \bullet D_{13} \bullet D_{14} \bullet D_{15}$$

2.4.3 Motivation for Two-Level Simplification

We can always use the rules of Boolean algebra from Section 2.1 to simplify an expression, but this method has a few problems. For one thing, there is no algorithm you can use to determine that you've obtained a minimum solution. When do you stop looking for a simplification theorem to apply? Another problem is that you often have to make the expression more complicated before you can simplify

A	B	C	D	C_0	C_1	C_2	C_3	C_4	C_5	C_6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	X	X	X	X	X	X	X	X
1	1	X	X	X	X	X	X	X	X	X

Figure 2.29 Seven Segment Display Controller Truth Table with Don't Cares

it. In our examples, we sometimes substituted $(X + \overline{X})$ for 1 to add more terms. Then we rearranged the terms to obtain advantageous groupings that helped to simplify the expression in a later step. It is against human nature to climb out of a "local minimum" in the hope of finding a better global solution. But this is exactly what we often have to do. And finally, it is just too cumbersome (and error prone) to manipulate Boolean expressions by hand.

Given that computer-based tools have been developed for Boolean simplification, why bother to learn any hand method, especially when these break down for problems with many variables? Certainly no hand method will be effective for equations involving more than six variables. But you still need knowledge of the basic approach. Observing the symmetries in a circuit's function helps to understand its behavior and to visualize what is going on inside the circuit. As CAD tools become ever more sophisticated, you need a deeper knowledge of algorithms they apply to use the tools effectively. And don't forget that CAD tools are written by mere mortals and do not always do the correct things! You must still be able to check the output of the tool.

A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

(a)

A	B	G
0	0	1
0	1	0
1	0	1
1	1	0

(b)

Figure 2.30 Two simple truth tables.

The Essence of Boolean Simplification Let's look at what is really going on in simplification with the simple truth table of Figure 2.30(a). The function is $F = A\overline{B} + AB$. We can simplify this equation by applying one of the Boolean simplification theorems, called the *Uniting* theorem: $A(\overline{B} + B) = A$.

Notice that the two truth table rows that make up the on-set of F have A asserted, while one row has $B = 0$ and the other has $B = 1$. For a subset of the on-set (in this case, the whole on-set), *A's value stays the same while B's value changes*. This allows us to factor out B using the uniting theorem.

Now examine Figure 2.30(b). The function is $G = \overline{A}\overline{B} + A\overline{B}$. Applying the uniting theorem again, we obtain $(\overline{A} + A)\overline{B} = \overline{B}$. Once again, the on-set contains two rows in which B's value does not change (it is equal to 0) and A's does change. Thus we can factor out A, leaving $\overline{B}$.

The essence of simplification is repeatedly to find two-element subsets of the on-set in which only one variable changes its value while the other variables do not. You can eliminate the single varying variable from the term by applying the Uniting theorem. We'll explore some algorithms for performing two-level logic minimization in Chapter 4.

2.5 Multilevel Logic

Given a Boolean function expressed in minterm or maxterm canonical form, we know we can reduce it into a minimal two-level form with the fewest terms and literals. Let's consider the function $Z(A, B, C, D, E, F, G)$:

$$Z = ADF + AEF + BDF + BEF + CDF + CEF + G$$

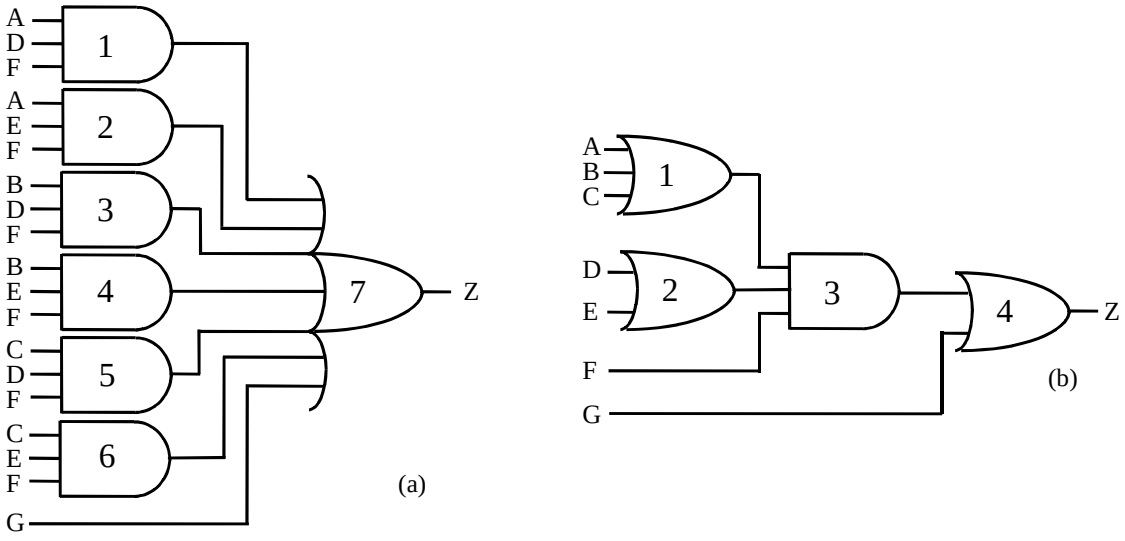


Figure 2.31 Multilevel circuit implementation.

It is already in its minimal sum of products form. Its implementation as a two-level network of AND and OR gates requires six 3-input AND gates and one 7-input OR gate, a total of seven gates and 19 literals (see Figure 2.31(a)).

We can do better if we replace the two-level form with a so-called *factored form*. We express the function with common literals factored out from the product terms whenever possible.

By recursively factoring out common literals, we can express the function Z as:

$$Z = (AD + AE + BD + BE + CD + CE)\,F + G$$

$$Z = [(A + B + C)\,D + (A + B + C)\,E]\,F + G$$

$$Z = (A + B + C)(D + E)\,F + G$$

Expressed as a series of expressions, each in the two-level form, Z becomes

$$Z = XYF + G$$

$$X = A + B + C$$

$$Y = D + E$$

When written this way, the function requires one 3-input OR gate, two 2-input OR gates, and a 3-input AND gate, a total of four gates and nine literals. The intermediate functions X and Y count as literals in the final expression for Z.

The implementation from the factored form is shown in Figure 2.31(b). You can significantly reduce the number of wires and gates needed to implement the

function, but this implementation probably may have worse delay because of the increased levels of logic. On the other hand, the 7-input OR gate of the two-level implementation is likely to be quite slow. In general, it is difficult to tell a priori whether a two-level or multi-level implementation will be faster or smaller. However, multilevel implementations tend to use gates with fewer inputs (because expressions are factored) which tend to be faster, however, their multiple levels of logic could add up to a larger combined delay than a two-level implementation with two levels of larger slower gates.

In this section, we will be concerned with two issues: how to express logic networks solely in terms of NAND and NOR gates, and how to take advantage of multilevel logic to reduce the overall gate count.

2.5.1 Logic Networks (NAND-NOR conversion)

The canonical forms you have studied so far are expressed in terms of AND and OR gates, but you will rarely encounter these in digital systems. The underlying technologies are more efficient at implementing NAND and NOR gates. In fact, AND and OR gates are most commonly realized by following a NAND or NOR gate with an inverter. In addition, NAND and NOR functions are *complete*; that is, a function expressed in terms of AND, OR, and NOT operations can be implemented solely in terms of NAND or NOR operations. Frequently, you will be confronted with the task of mapping a network with an arbitrary number of levels of AND and OR gates into one that consists only of NAND or NOR gates. We will begin the discussion with two-level networks and extend it to multilevel networks.

Visualization: DeMorgan's Theorem and Pushing Bubbles The conversion process depends critically on DeMorgan's theorem. Recall that

$$\overline{AB} = \overline{A} + \overline{B} \qquad \overline{A + B} = \overline{A}\,\overline{B}$$

and that

$$A + B = \overline{(\overline{A}\,\overline{B})} \qquad AB = \overline{(\overline{A} + \overline{B})}$$

In essence, a NAND function can be implemented just as well by an OR gate with its inputs inverted. Similarly, a NOR function can be implemented by an AND gate with its inputs complemented. The conversion from one form to the other is often called "pushing the bubble." This is simply a way to remember DeMorgan's theorem. As the bubble "pushes through" an AND shape, it changes to an OR shape with bubbles on the inputs. Similarly, pushing the bubble through an OR shape transforms it to an AND shape with bubbles on the inputs.

Figure 2.32 summarizes the relationship between OR gates and NAND gates. An OR gate is logically equivalent to a NAND gate with its inputs inverted. Similarly, an AND gate is equivalent to a NOR gate with its inputs complemented.

A	$\overline{A}$	B	$\overline{B}$	A + B	$\overline{\overline{A} \cdot \overline{B}}$	$\overline{A} + \overline{B}$	$\overline{A \cdot B}$
0	1	0	1	0	0	1	1
0	1	1	0	1	1	1	1
1	0	0	1	1	1	1	1
1	0	1	0	1	1	0	0

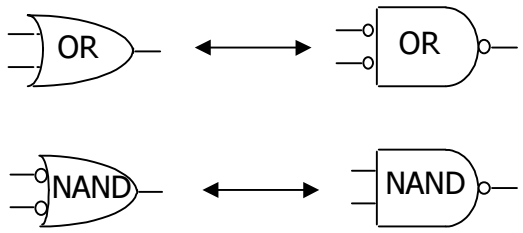


Figure 2.32 OR/NAND equivalences.

A	$\overline{A}$	B	$\overline{B}$	A • B	$\overline{\overline{A} + \overline{B}}$	$\overline{A} \cdot \overline{B}$	$\overline{A + B}$
0	1	0	1	0	0	1	1
0	1	1	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0

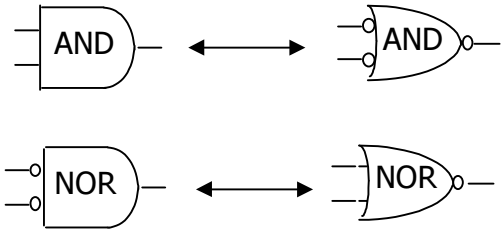


Figure 2.33 AND/NOR equivalence.

This is shown in Figure 2.33. The schematic symbols in Figures 2.32 and 2.33 can be freely exchanged without changing the function's truth table.

AND/OR Conversion to NAND/NAND Networks Consider the AND-OR network in Figure 2.34(a). The bubbles on the first stage gate inputs indicate that these should be complemented. Let's replace the first-level AND gates by their NAND equivalents and the OR gate by its NAND equivalent. The equivalent circuit is shown in Figure 2.34(b). Note that the output inversion at the first-level gates matches the input inversion at the second level. We can replace the second level NAND gate with its more conventional alternative form. The result is shown in Figure 2.34(c).

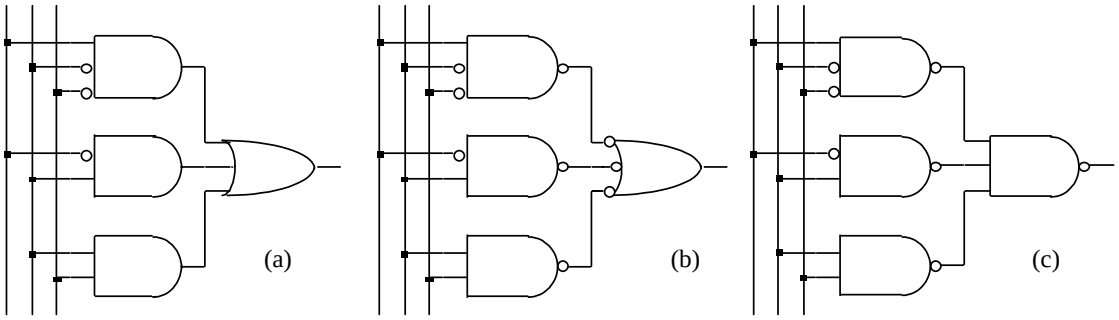


Figure 2.34 AND/OR to NAND/NAND.

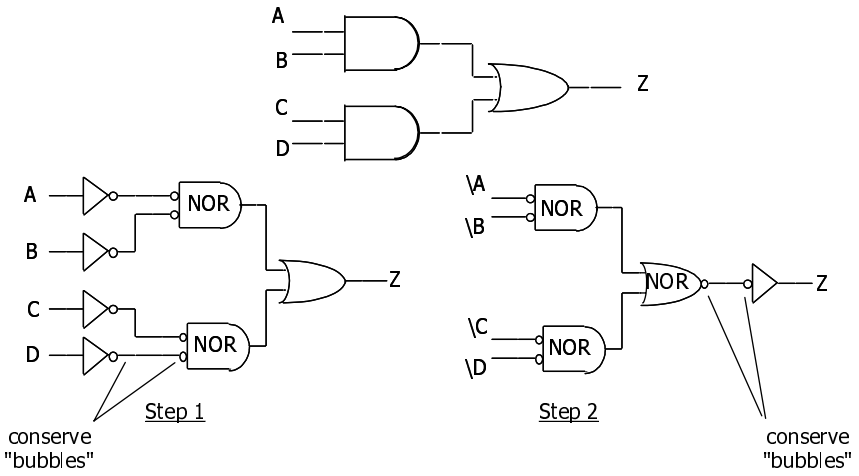


Figure 2.35 AND/OR conversion to NOR/NOR.

AND/OR Conversion to NOR/NOR Networks Suppose you are now restricted to mapping the AND/OR network into a NOR-only network. As a shortcut, you can simply replace the first-level AND gates with NOR gates (AND with inverted inputs) and the second-level OR gate with a NOR gate.

But this is not logically equivalent. Every time a new inversion is introduced, it must be balanced by a complementary inversion. We call this "conserving bubbles." To accomplish this, we introduce additional inverters at the inputs and the output, The two step process is illustrated in Figure 2.35.

To keep track of the need for inverted inputs to the first-level gates, we use the notation: *A*, *B*, *C*, and *D*. Since it is common to have both a Boolean variable and its complement available as circuit inputs, the conversion may not lead to additional inverters. To eliminate the inverter at the output, we may be able to use the complement of the function *Z*, *Z*, wherever in the rest of the circuit Z was needed.

OR/AND Conversion to NOR/NOR Network Now let's consider a gate implementation for a simple expression in product of sums form. We can map this expression into a NOR/NOR network simply by replacing the OR gates with NOR gates and the AND gate with a NOR gate (an AND gate with inverted inputs). You can see in Figure 2.36 that the inversions are appropriately conserved.

OR/AND Conversion to NAND/NAND Networks Implementing the expression using NAND-only logic introduces exactly the same problems we have already seen in Figure 2.35. The correct transformation replaces the OR gates with NAND gates (OR gates with inverted inputs) and the AND gate with a NAND gate. To maintain equivalence with the original function, we place inverters at the inputs and at the output. This is shown in Figure 2.37.

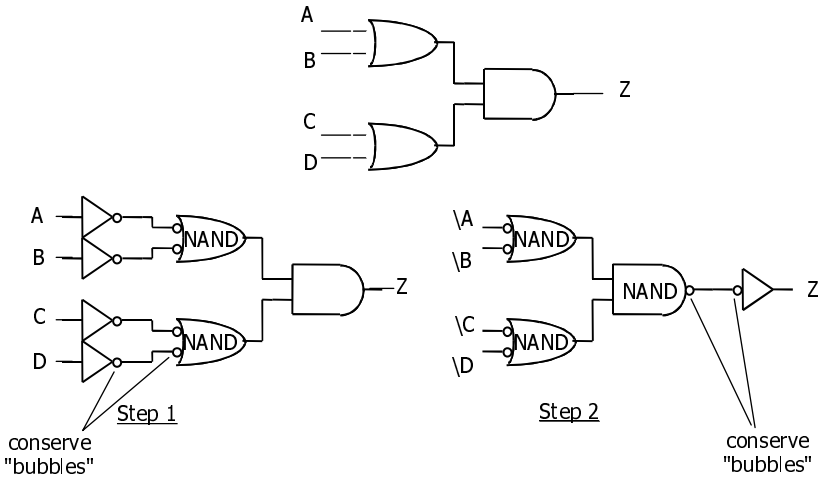


Figure 2.37 OR/AND conversion to NAND/NAND.

Generalization to Multilevel Circuits We can extend the transformation techniques to multilevel networks. Consider the function

$$F = A(B + CD) + B\overline{C}$$

Its implementation in AND/OR form is shown in Figure 2.38(a). You can see how we have arranged the logic into alternating levels of AND and OR gates. This makes it easier to observe the places where the conversion to NAND/NAND gates can take place. You simply replace each AND with a NAND and each OR with a NAND in its "alternative" form (OR with inverted inputs).

The application of this procedure is shown in Figure 2.38(b). We have grouped levels 1, 2 and 3, 4 into AND/OR circuits. These can be replaced by equivalent NAND/NAND networks directly.

Note that the literal *B* input to gate G3 must be inverted to preserve the original sense of the signal wire. Always remember to conserve the introduction of inversions. Any internal signal wires that undergo an odd number of inversions must have an additional inverter inserted in the path.

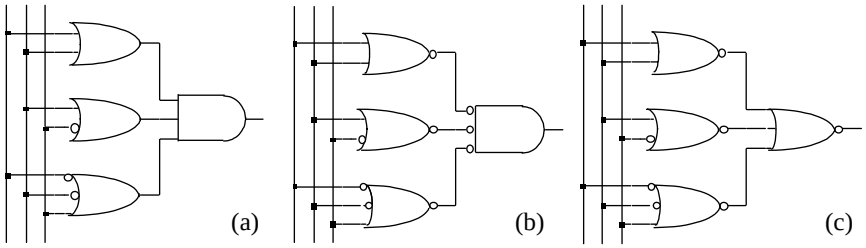


Figure 2.36 OR/AND conversion to NOR/NOR.

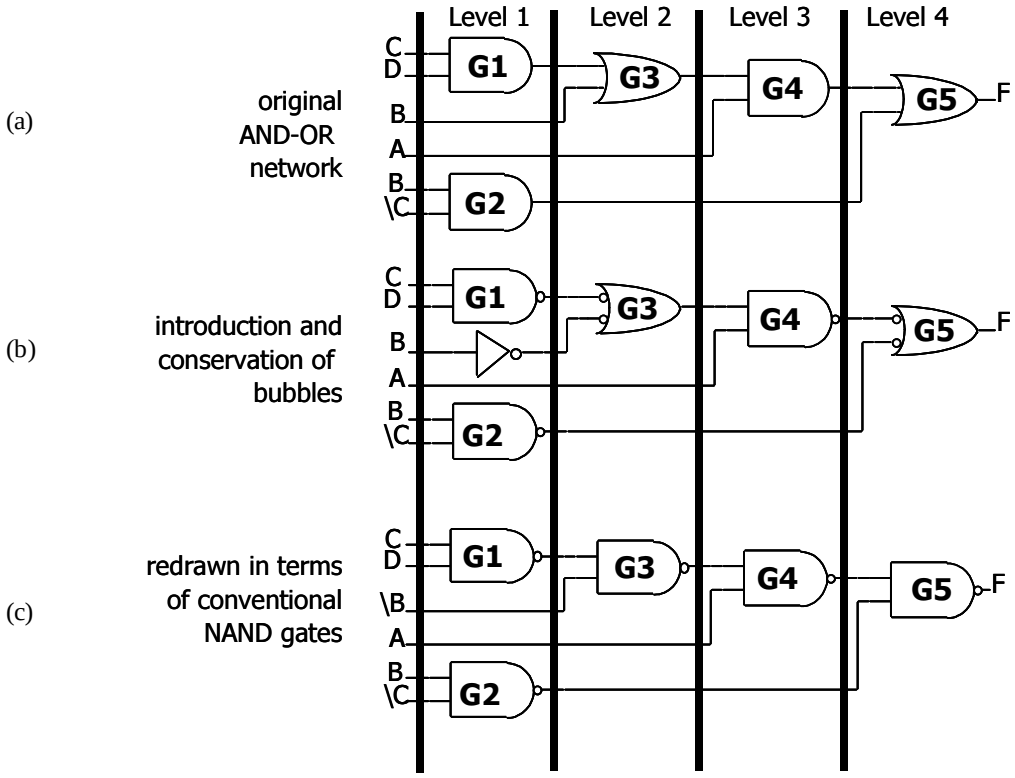


Figure 2.38 Multilevel conversion to NAND logic.

The final NAND-only network is shown in Figure 2.38(c). We have eliminated an inverter by replacing the *B* input to G3 with a connection to its complemented literal.

Suppose your target is a NOR-only network. You can take the same approach when the initial network is expressed as alternating OR and AND levels. You should place OR gates at the odd levels and AND gates at the even levels. You can immediately replace these by NOR gates. Any unmatched input bubbles should be corrected by inserting inverters or using the complemented literal where necessary.

It is a little more complicated when transforming alternating AND/OR networks (OR/AND networks) into NOR-only circuits (NAND-only circuits). Nevertheless, you can still apply the same basic techniques.

For example, suppose you want to map the AND/OR/AND/OR network of Figure 2.39(a) into NOR gates. You should invert the inputs to the odd levels while inserting an extra inversion at the output of the even levels. The extra inversions between adjacent even and odd levels can be saved if they cancel each other. This is shown in Figure 2.39(b) between levels 2 and 3, for gates G3 and G4. The final NOR-only circuit is shown in Figure 2.39(c). All but one of the literals have

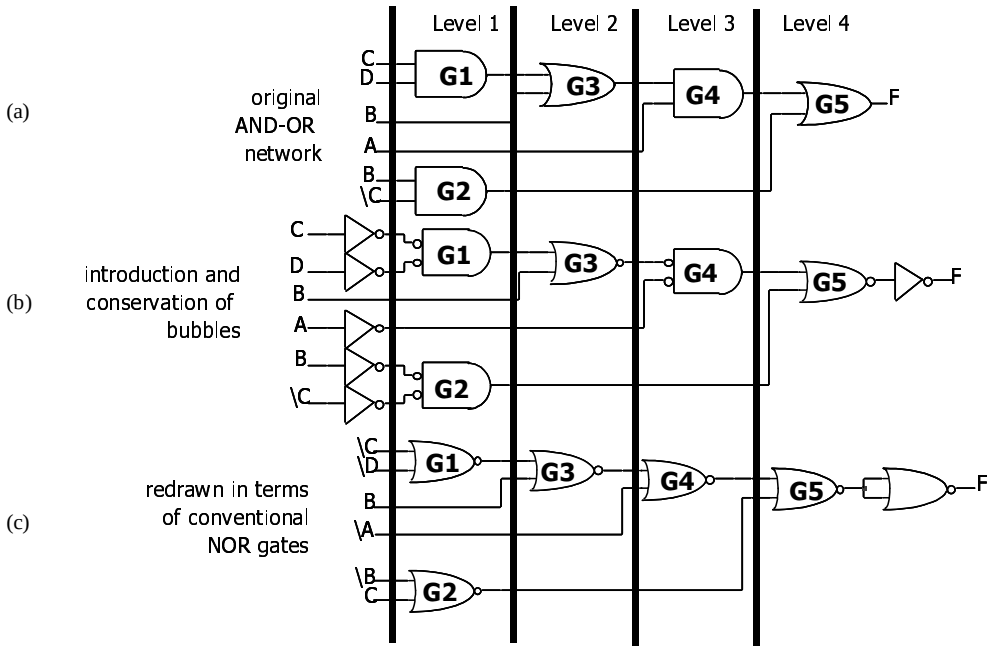


Figure 2.39 Multilevel conversion to NOR gates.

been inverted and an extra inversion has been inserted at the output. You can implement this last inversion by a NOR gate with both inputs tied to the same signal.

Figure 2.40 shows an example in which the circuit cannot be placed into a form that alternates between AND and OR gates. The multilevel function is

$$F = AX + X + D$$

$$X = BC$$

Figure 2.40(a) shows the initial AND/OR network. We begin by introducing double inversions at the inputs to the last-stage OR gate (Figure 2.40(b)). We propagate these back to the outputs of the two AND gates and the input *D* (Figure 2.40(c)). Note how the connection to *D* has been replaced by a connection to its complement to match the bubble on the OR's input.

There is still one problem with this circuit. The NAND gate computes the complement of the function *X*, not *X*. To conserve bubbles and the sense of the signals, the NAND output must be inverted before it can be input to the second-stage NAND gate. The final converted circuit is shown in Figure 2.40(d).

2.5.2 Building Blocks of Multi-Level Logic (Standard Cells)

Up to this point, we have described various kinds of logic gates with essentially no limits on their inputs. If you need a 20-input AND gate, we assumed that you would have one. Unfortunately, this is not really the case. Designers work with specific *logic families*, that is, specific gates and frequently encountered combinations of these, which constrain the primitive building blocks from which you can construct a design. The common combinations are pre-designed and are available as entries in a catalog or library of logic functions. These are often called *standard cells*. Not all of the possible logic gates will be available in a given family.

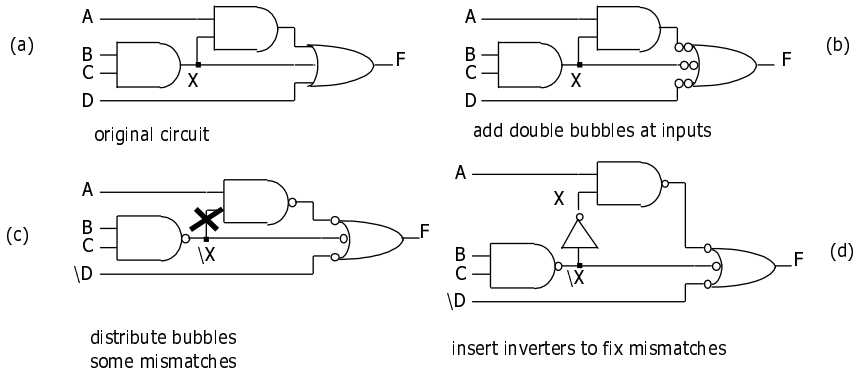


Figure 2.40 Another multilevel conversion example.

Transistor-Transistor Logic The TTL family of small- and medium-scale integration (SSI/MSI) is an old logic family, dating back to the 1960s, but it still in wide-spread use by logic designers. Furthermore, the catalog of TTL parts lives on as a model for the kinds of common building blocks found in more modern logic families.

Some of the kinds of building blocks found in the TTL family are:
Inverter Gates: 1-input;
NAND Gates: 2-, 3-, 4-, 5-. 8-, 12-, and 13-inputs;
AND Gates: 2-, 3-, and 4-inputs;
NOR Gates: 2-, 3-, and 4-inputs
OR Gates: 2- and 4-inputs;
XOR Gates: 2-inputs;
XNOR Gates: 2-inputs;

The TTL family also supports a variety of more exotic gate-level building blocks. An example is the AND-OR-Invert "gate" constructed from two 2-input AND gates at the first level connected to a 2-input NOR (OR-Invert) gate at the second level. Variations include more inputs at the first level (e.g., 3- or 4-inputs to the AND stage) and more inputs at the second level (e.g., four AND gates feeding the second stage NOR gate).

The TTL family also support even larger aggregations of logic gates that provide higher level functions. For example, the TTL catalog includes a 4-bit adder function and even a BCD-to-Seven Segment Display decoder! Catalog circuits of this complexity can easily make use of many tens of more primitive gates.

Programmable Read-Only Memories (PROM) and Programmable Array Logic (PALs) Any logic function can be expressed as a truth table, and it is possible to store or *program* the elements of a memory component to tabulate the relationship between the inputs, which form the address of the memory array, with the desired output, which is saved in the elements of the array. Each additional input doubles the size of the memory array, so there are some practical and economic limits, but this is not a bad way to implement a complicated function that is difficult to simplify.

A variation on the theme of programmable logic are devices called PALs. These consist of pre-designed AND and OR arrays where the only thing the designer needs to do is to specify the connections between the inputs and the AND gates, and the minterms formed by the AND gates and the OR gates. There are practical limits on the number of inputs to the AND and OR gates, but in general, they are much larger than are found in the basic TTL gate building blocks. On the order of eight inputs to the AND and OR gates is very commonly supported in PALs. Given the support for relatively large numbers of inputs, it does not make sense to use a PAL to implement a simple function of AND and OR gates. But it is usually cheaper in cost, design time, and circuit area to implement a complex set of logic equations using a PAL than discrete gates.

Field Programmable Gate Arrays FPGAs offer another approach to implementing logic systems. Xilinx is one popular manfacturer of such components, which they call *Logic Cell Arrays* (LCAs), but it illustrates the concept. An example of a basic building block is a *Combinational Function Block* (CFB) that can be programmed to implement any combinational logic function of up to five inputs. The CFB can also be configured to implement any two combinational functions of four inputs. This is accomplished by programming the appropriate truth tables and configuration bits into the CFB during the process of implementing a digital system that uses it as a component.

For example, a single CFB can implement a full adder bit slice, using the inputs X, Y, and Carry-in to generate the two outputs Sum and Carry-out. Five input AND, NAND, OR, NOR, XOR, XNOR, etc. gates are easily implemented in a CFB. Note that there might be advantages to simplifying a design to require gates of 4-inputs or less, since each CFB can implement two such gates. On the other hand, making the gates even simpler, such as 3- or 2- inputs, really buys the designed very little, since the number of CFBs to implement a given function is the same whether the gates are 4-, 3-, or 2- inputs.

This section illustrates just some of the practical issues related to how far to push a logic simplification. Fortunately, the more advanced logic families, like PALs and FPGAs, are supported by computer-aided design software to automatically map a specification of a collection of Boolean functions into an optimized design for the target family. The specification can take the form of Boolean algebra, gate-level drawings, or a description written in a high-level hardware description language. Such CAD tools are designed to make good use of the catalog of available building blocks for the particular logic family.

2.5.3 Metrics for Logic Minimization

We can obtain minimized two-level logic networks from the canonical sum of products or product of sums form, by applying Boolean algebra, as well as other reduction methods we will introduce in the next chapter. Signal propagations can be fast, because no signal has to travel through more than two gate levels (not counting zeroth-level inversions). The drawback is the potential for large gate fanins, which can reduce gate performance and increase circuit area in some technologies. In many technologies, gate-level building blocks with more than four inputs are rare. The list of catalog TTL parts in section 2.5.2 has few gates with more than 4 inputs. Most only have 2 or 3. Large fanin gates are found in programmable two-level logic but in a limited way, namely, to form product terms of many inputs. FPGAs also limit the number of inputs to a logic function for two reasons: to make the programmable logic blocks smaller and to use less wiring resources to interconnect signals between logic blocks. Thus, in virtually all practical gate-level realizations of a network, large fan-in gates must be replaced by a multiple-level network of smaller fan-in gates. This motivates much of the recent interest in automatic multilevel logic optimization.

An optimal two-level network uses the smallest number of product terms and literals to realize a given truth table. It is not so easy to define optimality for multilevel networks. Is the network optimal if it has the smallest number of gates? Or is it more critical to have the fewest literals in the resulting expression? Or is the number of inputs per gate a major consideration as in FPGAs?

To get a taste for how to simplify multilevel expressions, we will discuss a useful way to represent such expressions, and the kinds of operations that can be applied to them to achieve their simplification. The actual methods used by modern computer-aided design tools will be described in chapter 4.

Factored Form The standard way to represent a multilevel equation is in a so-called *factored form*. Simply stated, this is an expression that alternates between AND and OR operations, a kind of "sum of products of sums of products.…" The following function is in factored form:

$$X = (AB + \overline{B}C)[C + D(E + A\overline{C})] + (D + E)(FG)$$

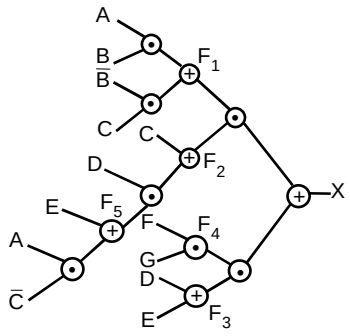


Figure 2.41 Graphical representation of a factored form with simple nodes.

Examining the expression, we see that no further subexpressions can be factored out. For the purpose of counting literals, the factored form for X can be rewritten as a sequence of two-level expressions:

$$X = F_1F_2 + F_3F_4 \qquad F_3 = D + E$$
$$F_1 = (AB + \overline{B}\overline{C}) \qquad F_4 = FG$$
$$F_2 = C + DF_5 \qquad F_5 = E + A\overline{C}$$

The structure of the expression is a little clearer if it is represented in the form of a graph or tree, where the "leaves" represent literals and the internal nodes represent either an AND or OR operation. This graphical representation is shown in Figure 2.41. For the most part, the ANDs and ORs alternate between adjacent nodes of the tree.

Criterion for Multilevel Simplification In modern logic families, designers have observed that gates (internal nodes of the graph of Figure 2.41) require relatively little circuit area but connections (edges of the graph) use significant area. Stated in a different way, the implementation complexity is strongly related to the number of wires used to construct a circuit. Because the number of internal connections scale with the number of literals, it is a reasonable strategy to attempt to minimize the number of literals. To count the number of literals in a multilevel expression, simply add up the number of literals found in its equivalent implementation in terms of two-level expressions. For the function depicted in Figure 2.41, for example, the literal count is 18. Note that when they are referenced in expressions, the intermediate functions F_1, F_2, F_3, F_4, and F_5 count as literals. Note that not all nodes represent an expression. One of the important choices to be made in multi-level optimization is how to group the nodes into intermediate expressions. This is usually based on the building blocks available in the technology being used for implementation and is thus a *technology-dependent* optimization step.

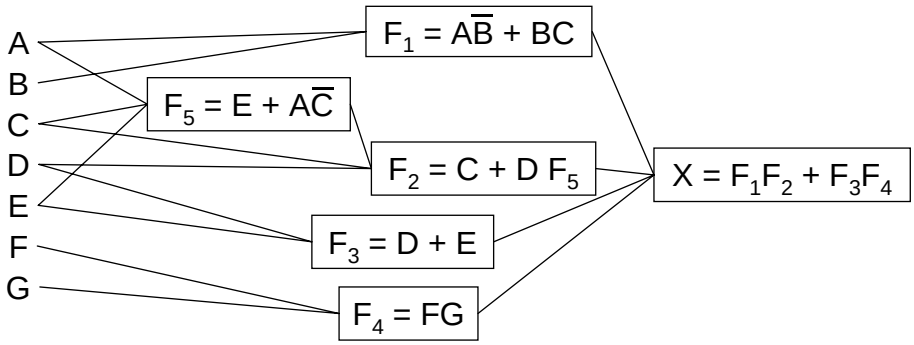


Figure 2.42 Graphical representation of a factored form with complex nodes.

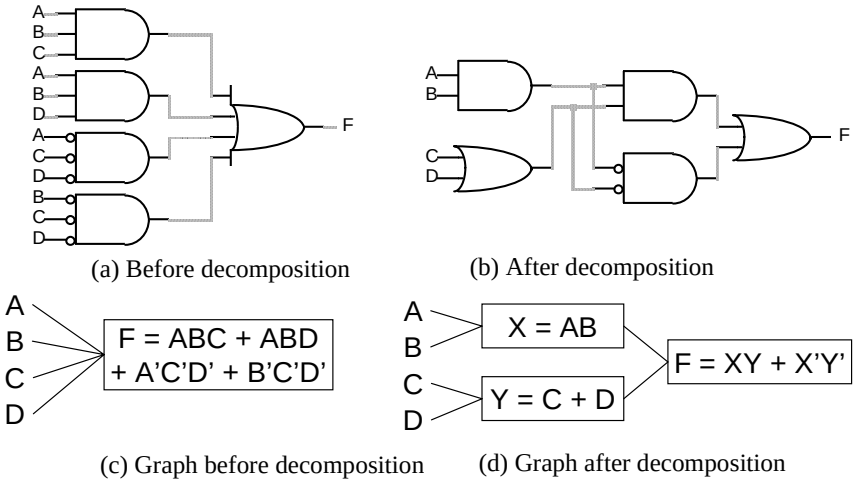
To represent these grouping of primitive functions, most tools use a more complex graph representation that places arbitrary functions in the nodes of the graph rather than just simple ANDs and ORs. Figure 2.42 shows the corresponding complex graph for the particular intermediate functions chosen above..

There are five basic operations for manipulating multilevel networks: (1) decomposition, (2) extraction, (3) factoring, (4) substitution, and (5) collapsing. In the following subsections, we will describe each of these operations and illustrate how each alters the multilevel expression with a simple example.

Decomposition Decomposition takes a single Boolean expression and replaces it by a collection of new expressions. Consider the function

$$F = ABC + ABD + \overline{A}\overline{C}\overline{D} + \overline{B}\overline{C}\overline{D}$$

This expression is in reduced sum of products form and has 12 literals. It requires nine gates, counting inverters, for implementation (see Figure 2.43(a)). it



(a) Before decomposition (b) After decomposition

(c) Graph before decomposition (d) Graph after decomposition

Figure 2.43 Effects of decomposition.

would be represented by a single node in our graph representation. However, the expression can be decomposed into three much simpler functions that will yield :

$$F = XY + \overline{X}\,\overline{Y} \qquad X = AB$$

$$Y = C + D$$

The resulting set of functions has eight literals and requires seven gates for implementation (see Figure 2.43(b))—nothing is free, though, and the number of gate levels has increased from two to three. These functions now require three nodes to be represented in our graph rather than one (see Figure 2.43(c,d)). As you would expect from the name, decomposition breaks a node into smaller simpler pieces.

Extraction Whereas you apply decomposition to a single function, you apply extraction to a collection of functions. This operation identifies common subexpressions in the collection of functions to which it is applied. Performing extraction requires that the functions be expressed in terms of their factors, and then the common factors must be "pulled out."

Let's look at an example of extraction. We start with three functions represented as three nodes

$$F = (A + B)CD + E$$

$$G = (A + B)\overline{E}$$

$$H = CDE$$

Note that the extraction operation does not require the functions to be in a two-level form.

In this example, the extraction operation discovers that the subexpressions $X = (A + B)$ and $Y = (CD)$ are common to F, G, and F, H, respectively. These subexpressions are called primary divisors or, more technically, *kernels* and *cubes*. Reexpressed in these terms, the functions can be rewritten as

$$F = XY + E$$

$$G = X\overline{E}$$

$$H = YE$$

$$X = A + B$$

$$Y = CD$$

The original collection of functions contains 11 literals and requires eight gates for implementation. (In Figure 2.44(a), the bubble at the input of the gate that computes G counts as one inverter gate.) The revised set of functions, after extraction, still contains 11 literals but now needs only seven gates for its implementation. The graph now has five nodes, the modified versions of the three original ones plus the two new nodes to represent the new subexpressions (see

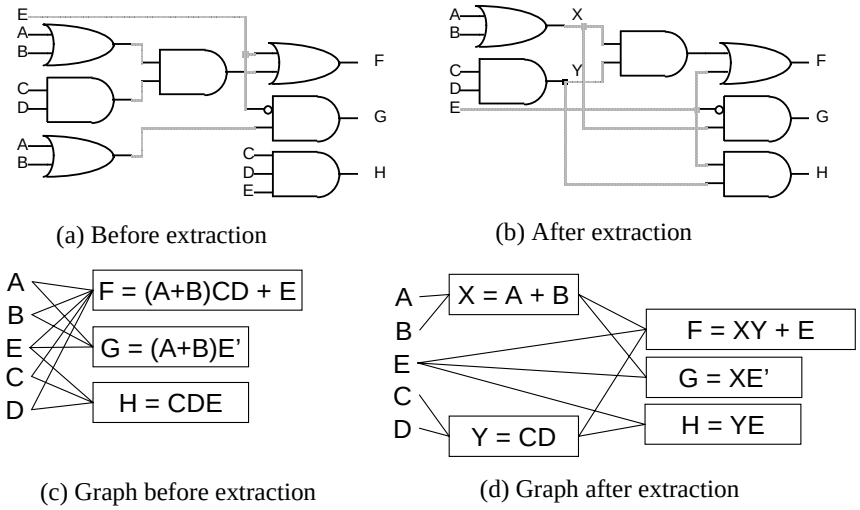


Figure 2.44 Effects of extraction operation.

Figure 2.44(c,d)). You can see in Figure 2.44(b) that the single-level implementation for *H* has been replaced by a two-level implementation after extraction. The number of gates is reduced, but the function *H* now probably incurs worse delay.

Factoring Factoring takes an expression in two-level form and reexpresses it as a multilevel function without introducing any intermediate subfunctions. Thus, factoring simply rewrites the expression within a node rather than changing the structure of the graph. It is used before extraction to identify potential common subexpressions.

As an example, let's consider the following function in sum of products form. It has nine literals and can be implemented with five gates (Figure 2.45(a)):

$$F = AC + AD + BC + BD + E$$

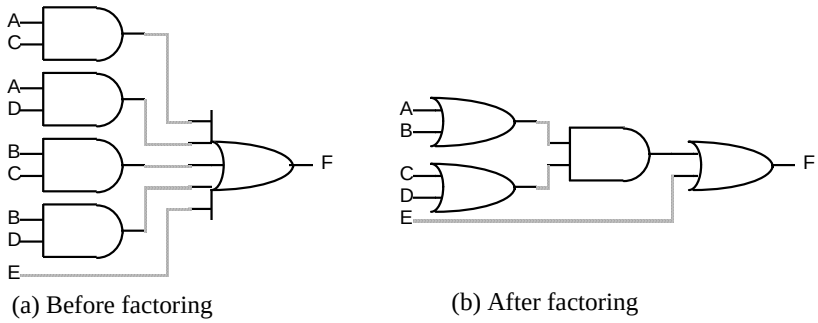


Figure 2.45 Effects of factoring operation.

After factoring, the number of literals is reduced to seven:

$$F = (A + B)(C + D) + E$$

This can be implemented with four gates (Figure 2.45(b)). Note that the graph representation for F does not change. It is still a single node with one input. Factoring only changes the representation of a function within a single node.

Substitution Substituting a function G into a function F reexpresses F in terms of G. For example, if $F = A + BC$, and $G = A + B$, then F can be rewritten in terms of G as follows:

$$\begin{aligned} F &= A + BC \\ &= G(A + C) \end{aligned}$$

Once common subexpressions have been identified, substitution can be used to reexpress functions as factored forms over the subexpressions. This operation also changes the structure of the graph by adding an arc from the node for G to the node for F and changing the definition of F.

Collapsing Collapsing is the reverse of substitution. It might be used to reduce the number of levels of logic to meet a timing constraint. As an example, we can collapse G back into F:

$$\begin{aligned} F &= G(A + C) \\ &= (A + B)(A + C) \\ &= AA + AC + AB + BC \\ &= A + BC \end{aligned}$$

The number of literals used to express F has been reduced from five to three. The graph is also modified in the opposite way as substitution in that an arc that existed between nodes G and F is now removed.

Polynomial Division and Multilevel Factoring All of the multilevel operations have strong analogies with the multiplication and division of polynomials. The strategy is to rewrite the expression for a function F in terms of the subexpressions P, Q, and R, which represent the divisor, quotient, and remainder, respectively. In generic terms, F is written as

$$F = PQ + R$$

As a more concrete example, given the two expressions:

$$X = AC + AD + BC + BD + E$$

$$Y = A + B$$

We could write X "divided" by Y as follows:

$$X = Y(C + D) + E$$

The divisor is $Y = (A + B)$, the quotient is $(C + D)$, and the remainder is E. Expanding the expression with the distributive law would yield the original equation for X.

Finding divisors is a rather difficult problem when the laws of Boolean algebra are considered. Multiplying Boolean expressions can yield unusual results because of the variety of simplification theorems you can apply. For example, consider the functions F and G:

$$F = AD + BCD + E$$

$$G = A + B$$

Under the normal rules of algebra, G does not divide into F. The so-called *algebraic* divisors of F are D and $(A + BC)$. That is, F can be divided by D, leaving the quotient $A + BC$ with remainder E. It can also be divided by $A + BC$, leaving D as the quotient and E as the remainder.

However, if we apply the rules of Boolean algebra, then G does divide into F. We can write the quotient of F divided by G as

$$F/G = (A + C)D$$

This is because F can be rewritten as

$$\begin{aligned} F &= [G(A + C)D] + E \\ &= (A + B)(A + C)D + E \\ &= (AA + AC + AB + BC)D + E \\ &= (A + BC)D + E \\ &= AD + BCD + E \end{aligned}$$

The existence of Boolean divisors greatly increases the number of potential factorings of a set of expressions. It is not uncommon to restrict the search to the easier-to-find algebraic factors.

It should be clear from this discussion that the challenge in multilevel logic simplification is to find good divisors. These lead to factored expressions with the greatest number of common subexpressions. By factoring these out, we minimize the number of literals needed to express a set of functions. The five graph opera-

tions discussed above must be applied in a particular order to guide the transformations of the circuit toward a result that meets our optimization criteria: smallest size, smallest gates, fewest levels, etc. We will return to these issues in chapter 4.

2.6 Time Response in Combinational Networks

As a hardware designer, it is important to visualize how a circuit behaves over time—that is, to be able to look at a circuit and see how signals move through it, recognizing asymmetric delays along paths that can lead to transitory behavior at the outputs. This is not an easy skill to acquire, even after extensive design experience. Fortunately, simulation tools can offer great assistance in visualizing the time-based behavior of circuits.

2.6.1 Gate Delays

Outputs in combinational logic are functions of the inputs and some delay. *Gate delay* is the amount of time it takes for a change at the gate input to cause a change at its output. Most circuit families define delays in terms of minimum (best case), typical (average), and maximum (worst case) times. A corollary to Murphy's law, well known to experienced digital designers, is that if a circuit can run at its worst-case delay, it will. Never assume that your design will be able to run with minimum delay.

What would happen if you depended on a portion of your design running with minimum delay? If its delay is longer than you designed for, you may examine its output too soon, incorrectly computing the final output of your overall system.

OBTAIN TIMING INFORMATION FOR MODERN XILINX AND ACTEL PARTS FAMILIES.

2.6.2 Timing Waveforms

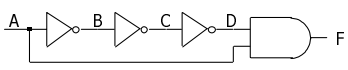


Figure 2.46 Pulse shaper circuit.

Let's consider the circuit shown in Figure 2.46. An input signal A passes through three inversions, leaving it in its inverted state, which is then ANDed with the original input. This appears to implement a rather useless function: $A \bullet \overline{A} = 0$. However, the timing diagram of Figure 2.47 tells us a different story. After the input A goes high, the output waveform goes high for a short time before going low. Such a circuit is called a *pulse shaper* because a change at its input causes a short-duration pulse at the output.

The circuit of Figure 2.46 operates as follows. Let's assume that the initial state has $A = 0$, $B = 1$, $C = 0$, $D = 1$, and $F = 0$, as shown in Figure 2.47 at time step 0. To make what is happening more clear in the figure, rather than our usual unit delay we assume that each gate has a propagation delay of 10 time units. When input A changes from 0 to 1 at time 10, it takes 10 time units, a gate delay, before B changes from 1 to 0 (time step 20). After a second gate delay, C changes from 0 to 1 (time step 30). D changes from 1 to 0 after a third gate delay (time step 40). However, between time 10 and time 40, both A and D are logic 1. If the

AND gate also has a 10-unit gate delay, the output *F* will be high between time steps 20 and 50. This is exactly what is shown in the timing diagram. In effect, the three inverters stretch the time during which *A* and *D* are both logic 1 after *A* changes from 0 to 1. Eventually, the change in *A* propagates to *D* as a 0, causing *F* to fall after another gate delay. It is no surprise that the pulse is exactly three inverter delays wide. If we increased the number of inverters to five, the width of the pulse would be five gate delays instead.

A pulse shaper circuit exploits the propagation asymmetries in signal paths with the explicit purpose of creating short-duration changes at the output. It generates a periodic waveform that could be used, for example, as a clock in a digital system. It operates much like a stopwatch. With its switch in one position, the circuit does nothing. In the second position, the circuit generates a periodic sequence of pulses.

2.6.3 Application: Analysis of a Pulse Shaper Circuit

In this section, we will analyze the operation of the simple pulse shaper circuit of Figure 2.48. The circuit has a single input *A* that is connected to a logic 1 when the switch is open and to a logic 0 when the switch is closed. This is because the path to ground has lower resistance than the path to the power supply when the

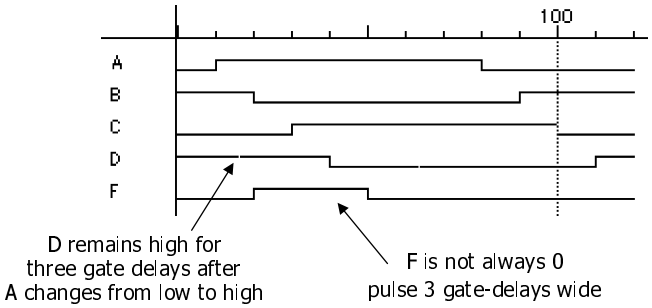


Figure 2.47 Pulse shaper waveform.

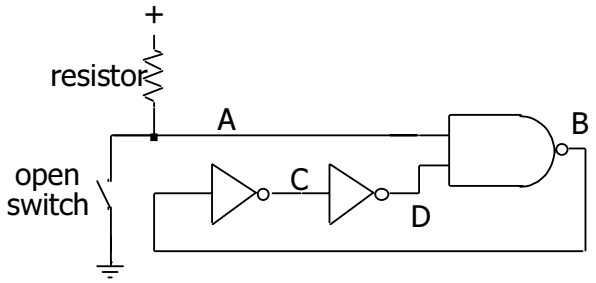


Figure 2.48 Pulse shaper example.

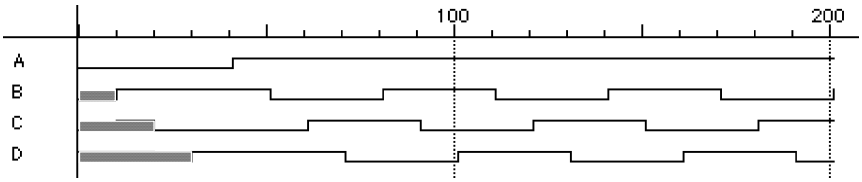


Figure 2.49 Timing waveform for pulse-shaping circuit.

switch is closed (switches are discussed in more detail in section 5.3). We will assume that the propagation delay of all gates is 10 time units.

Let's suppose that at time step 0, the switch has just been closed. We begin by determining the initial value for each of the circuit's nets. *A* goes to 0 instantly. Since a NAND gate will output a 1 whenever one of its inputs is 0, *B* goes to 1, but after a gate delay of 10 time units. So we say that *B* goes to 1 at time step 10.

C is set to the complement of *B*, but once again only after an inverter propagation delay. Thus *C* goes to 0 at time step 20. *D* becomes the complement of *C* after another inverter delay. So it goes to 1 at time step 30. Since *A* is 0 and *D* is 1, the output of the NAND gate stays at 0. The circuit is said to be in a *steady state*.

What happens if the switch opens at time step 40? The input *A* immediately goes to 1. Now both inputs to the NAND gate are 1, so after a gate propagation of 10 time units, *B* will go low. This happens at time step 50.

The change in *B* propagates to *C* after another inverter delay. Thus at time step 60, *C* goes to 1. In a similar fashion, *D* goes to 0 at time step 70. Now the NAND gate has one of its inputs at 0, so at time step 80 *B* will go to 1.

Note that *B* first goes low at time step 50 and goes high at time step 80—a difference of 30 time units. This is exactly three gate delays: the delay through the NAND gate and the two inverter gates on the path from signal *B* to *D*.

Now that *B* is at 1, *C* will go to 0 at time step 90, *D* will go to 1 at time step 100, and *B* will return to 0 at time step 110. The circuit is no longer in steady state. It now oscillates with *B*, *C*, and *D* varying between 1 and 0, staying at each value for three gate delays (30 time units). The behavior of the circuit is summarized in the timing diagram of Figure 2.49.

2.7 Hardware Description Languages

Up to this point, we have used Boolean expressions and schematic diagrams to describe our logic circuits. As you might imagine, it can become quite difficult to comprehend a large diagram or long Boolean expression, or for that matter, even to draw or write one. Hierarchy helps with this problem in that it lets us use smaller pieces to describe larger entities. We have seen this when we decompose a large Boolean expression into smaller subexpressions and we have seen it in schematic

diagrams such as using two half-adders to form the full-adder of Figure 2.15. But hierarchy is not enough. Diagrams take a lot of time to draw legibly. Computer-aided schematic editors make the job easier but a small change in a logic expression can cause a large change in its corresponding schematic drawing. Boolean expressions can be difficult to read and do not provide an easy way to comprehend the function being described. For example, consider the ease of reading the C program versus the sum-of-products expression for the leap month calculation of Chapter 1.

Hardware description languages were developed to deal with these issues. They provide a way for designers to textually describe logic circuits while exploiting some of the advantage of software languages, namely, variables, sub-procedures, and conditional and interative statements.

However, the most valuable use of hardware description languages is that they also permit a designer to exercise their design without having to physically build it first. Descriptions in these languages can be *executed*. That is, they are run like a software program. A program that emulates the actual behavior of the circuit as faithfully as it can. Of course, it does not match reality precisely. Electrical effects require large amounts of computation to model in high-accuracy detail. A balance must be struck between fidelity and performance in the simulator.

Hardware description languages have been around for a long time. The first were designed to help processor designers describe their new designs and actually execute programs on them. A language called ISP, for instruction set processor, was developed at Carnegie-Mellon University in the mid-70s. It radically shortened the amount of time needed to get a new design to work correctly by allowing designers to find most of the bugs in simulation rather than after expensive hardware had already been built.

The success of these efforts was followed in the early 1980s by HDLs to describe arbitrary logic rather than just processor elements. These were mostly developed by industry (such as ABEL by Data-I/O and Verilog by Gateway) attesting to their utility. VHDL, developed under the leadership of Department of Defense, was developed to help in modeling of even more aspects of complex systems that are composed of hardware as well as software. Both Verilog and VHDL are now IEEE standards.

By the early 1990s, with most designers using hardware description languages, attention turned to *compilers*, known as synthesis tools, that could automatically turn HDL descriptions into circuit implementations. More recently, the focus of research is on system-level description languages that can be used to describe complex distributed systems and synthesis tools to generate the code and hardware that will make it all work.

In this text, we will use Verilog simply because it makes it much easier for beginners to get started and it has a syntax similar to the C programming language. Verilog and VHDL almost evenly divide the users of HDLs.

Hardware description languages appear quite similar to programming languages at first glance, but we'll see, in this and later chapters, that there are some fundamental differences. One of things they do share, however, is the notion of hierarchy. With HDLs, we define modules and then compose these into larger

designs. A very basic module description in Verilog might look something like this:

```
module xor_gate (a, b, z);
    input    a, b;
    output   z;

    <module internals>

endmodule
```

The module has a name, in this case, xor_gate, and three wires: a, b, and z. a and b are inputs, while z is an output. We can think of a and b as the input parameters of the module and z as its output parameter or return value. There are two principal ways of describing the internals of the module: structure or behavior.

2.7.1 Describing Structure

A structural description is simply a textual version of schematic diagram. For OR gate using the schematic of Figure 2.50, then the module description would be completed as follows:

```
module xor_gate (a, b, z);
    input    a, b;
    output   z;
    wire     abar, bbar, t1, t2;

    inverter invA(a, abar);
    inverter invB(b, bbar);
    and_gate and1(a, bbar, t1);
    and_gate and2(abar, b, t2);
    or_gate or1(out, t1, t2);

endmodule
```

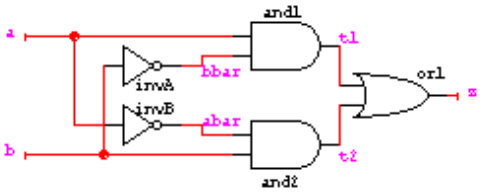


Figure 2.50 .Schematic for an exclusive-or gate implementation

The module description lists all of the five gates of Figure 2.50 and uses variable names for the connecting wires. Each gate is an *instance* of another module. In this case, we use three different types of modules: inverter, and_gate, and or_gate. Descriptions of these modules would also be in the complete circuit description. However, just as in software, there will most likely be libraries of primitive elements so that descriptions can be made more concise. The five gates are given unique names (invA, invB, and1, and2, and or1) so that they can be referred to in later simulation and debugging. Finally, note that abar, bbar, t1, and t2 had to be declared as internal variables to the module as they are wires that are neither inputs nor outputs.

2.7.2 Describing Behavior

Rather than describing in detail how the function will be realized (by describing the gates to be used and how they will be interconnected), a behavioral description simply describes the function of the module without spelling out a specific implementation. A behavioral description of our xor_gate module may look as follows:

```
module xor_gate (a, b, z);
    input    a, b;
    output   z;
    reg      z;

    always @(a or b) begin
        z = a ^ b;
    end

endmodule
```

The module description contains an *always* block. This specifies *when* the values of the outputs of a module need to be updated and *how.* In this case, whenever the value of a or the value of b changes, then z should be assigned the value of the exclusive-or of a and b. The *sensitivity list* is specified within the parentheses after the @ symbol. The statement to be executed to determine how to update the value of z is within the begin-end block. The ^ symbol is used to signify the exclusive-or operation in Verilog. Make sure not to confuse the “or” between a and b for a logical operation. This or means that if a “or” b change in value then the statement within the always block should be executed. This is quite different than in programming languages where statements are executed sequentially. In this case, a statement is executed because of a change in the value of a wire. HDLs do this, because this is how real circuits operate and they are trying have as similar semantics as possible. The additional declaration for z makes this clear to the simulator (by indicating that z will have new values assigned to it continously and a *register* should be created to keep track of when the statements that assign a new value to z need to be re-evaluated).

We can also include more complex statements inside of always blocks that include iteration and conditionals. For example, we can re-write the always block for our xor_gate as follows:

```
always @(a or b) begin
    if (a) then z = ~b else z = b;
end
```

The if statement also realizes the exclusive-or of a and b and assigns a value to z. The ~ symbol is used to complement values in Verilog.

These types of always blocks are so common in Verilog, that there is a special short-hand notation that is exactly equivalent:

```
module xor_gate (a, b, z);
    input   a, b;
    output  z;
    reg     z;

    assign z = a ^ b;

endmodule
```

The assign statement specifies that z should *continuously* be assigned a value that is the excluive-or of a and b. In fact, this is referred to as a continous assignment statement in Verilog. It is quite different than an assignment statement in a programming language. It is not executed once, but continously. What this means in practice, is that the statement is re-evaluated whenever the value of a or b changes so that we can ensure that z always has the correct value. In effect, the assign statement takes the burden from the designer of having to spell out the sensitivity list. The statement is re-evaluated whenever any variable used on the right side of the assignment changes value.

The additional declaration for z makes this clear to the simulator (by indicating that z will have new values assigned to it continously and a *register* should be created to keep track of when the statements that assign a new value to z need to be re-evaluated).

Of course, for a design to be complete, all its modules will require complete structural descriptions. However, there are several reasons for wanting behavioral descriptions as well. Early in the design process, designers may not want to put a lot of effort into a schematic that may have to be changed later. It is easier to simply state the function that will be required and move on to simulating and debugging the overall design. Moreover, automatic synthesis tools can be used to compile a structural description given a behavioral description. As you can imagine, these tools have radically improved the productivity of designers. Finally, some modules may never be realized in circuitry but are simply present to provide a simulation context for the design. We'll see an example of this shortly.

2.7.3 Delay

Sensitivity lists emulate how real logic circuitry works. Another aspect of real hardware that needs to be emulated is its delay characteristics. Verilog provides for this with a delay statement. For example, if we want our xor_gate to have a delay of 6 time unit, then we can add a delay to its behavioral description as follows:

```
module xor_gate (a, b, z);
    input   a, b;
    output  z;
    reg     z;

    assign #6 z = a ^ b;

endmodule
```

```
module xor_gate (a, b, z)
    input   a, b;
    output  z;
    reg     z;

    always @(a or b) begin
        #6 z = a ^ b;
    end

endmodule
```

The effect of the delay statement is to postpone the assignment of a new value to z for 6 time units so that a change in z will occur 6 time units than the change in a or b that caused it to happen. Note that delay statements only make sense within a behavioral description. In a structural description it is the responsibility of sub-modules to have the appropriate delay between their inputs and outputs.

2.7.4 Event-Driven Simulation

We have mentioned simulation several times in this discussion of HDLs. A simulator is a tool that can read hardware descriptions written in languages such as Verilog and execute them for us. Most HDL simulator are *event-driven*, that is, they are based on the concept of an event occurence. An event is simply the change in the logic value carried on a wire. It has an associated value and a time of occurence.

Simulators are constructed to propagate events through the modules that consistute a design. In our example, any changes in the values of wires a and b consistute events, these are propagated through the xor_gate module and a new event on z may be generated. That, in turn, may affect other modules and cause them to generated other events. This continues until there either no more events or the designer stops the simulation. Delay statements are used to advance time in event-driven simulators. Without delay, all events would occur in the same virtual instance. By modeling delay, we separate events in time in the same way they would be in a real circuit. Event-driven simulators take the burden from designers of having to manually check event propagation. This makes them invaluable and very popular tools for logic design.

Event-driven simulators need events to start propagating. These may be generated by a human designer who “drives” the simulation by changing values on wires through the simulator’s user interface. Alternatively, we can include modules

in our simulated design whose function is simply to generate event or react to events generated by the circuit. Of course, these modules would not be present in a realization of the circuit. Suppose we would like to exercise our xor_gate so that it was "stimulated" with the four combinations of possible input values. We could create a module that cycled through these four and connect it our xor_gate as follows:

```
module stimulus (x, y);
    output   x, y;
    reg      x, y;

    intial begin
        x = 0; y = 0;
        #10
        x = 0; y = 1;
        #10
        x = 1; y = 0;
        #10
        x = 1; y = 1;
        #10
        $finish
    end

endmodule
```

```
module both_together(z);
    output     z;
    wire       w1, w2;

    stimulus stim1(w1, w2);
    xor_gate xor1(w1, w2, z);

    always @(z) begin
        $diplay("At time: %d with inputs:
            %b and %b, the output is: %b",
            $time, w1, w2, z);
    end

endmodule
```

The module on the left uses delay statements to change the values of its x and y every 10 time units. The *initial* block is only executed once at the start of a simulation. The *$finish* statement halts the simulator. Note that the module has no inputs. The module on the right, then connected this stimulus generator to an xor_gate. It has a single output z which will change at time 6, 16, 26, and 36. The simulator will halt at time 40. The *$display* statement is used as part of the user interface of the simulator. It is similar to a printf statement in C. By enclosing it in an always block, we will cause it print its string every time z changes value.

Of course, there are many more features to Verilog. Unfortunately, there is not enough space here to provide all the details. One last example will serve to highlight some very useful elements of the language.

```
module stimulus (x, y);
    output   x, y;
    reg[1:0] cnt;

    intial begin
        cnt = 0;
        repeat (3) begin
            #10 cnt = cnt +1;
        end
        #10
        $finish;
    end

    assign x = cnt[1];
    assign y = cnt[0];

endmodule
```

This module does precisely the same thing as the previous verison of “stimulus”. It accomplish it using a repeat loop and a two-bit variable called cnt. The outputs x and y are continously assigned the respective bits of cnt. The high-order bit is connected to x and the low-order bit to y using the two continous assignment statements at the bottom. The outputs x and y change whenever cnt is incremented.

Chapter Review

In this chapter, we have introduced a variety of primitive logic building blocks: the Inverter, AND, OR, NAND, NOR, XOR, and XNOR gates, with which we can implement any Boolean function. We have also presented the two primary canonical forms for describing a Boolean function: sum of products and product of sums. A function may have many equivalent Boolean expressions but only one representation in a canonical form. Logic minimization, to be introduced in the next chapter, seeks to find the equivalent expression with the minimum number of terms and the fewest literals per term. Don’t-care conditions on the inputs can be used to simplify the expression substantially. Reduced expressions lead to realizations of circuits with the fewest gates and the fewest inputs, at least if the target is a two-level circuit realization.

We build on top of simple gate logic to show the process of conversion of AND/OR and OR/AND networks into NOR-only and NAND-only logic. Under certain conditions, an additional inverter must be added at the output, so these conversions change two-level logic to a three-level form.

We examined the concept of multilevel logic and introduced its advantages in terms of reduced literal counts and simplified wiring complexity. Of course, multilevel logic introduces the possibility of increased circuit delay by placing additional levels of gates between inputs and outputs. In general, two-level logic yields the fastest implementations and multilevel logic results in circuits with fewer wires requiring less area for implementation,

We introduced the concept of time response in combinational logic network, and showed how the dynamic behavior of a combinational logic circuit in response to input changes can be different from its steady state behavior.

The final section dealt with hardware description languages and how we can describe circuit modules in both structural and behavioral styles. We also highlighted the use of HDLs in simulating designs early in the design process so that designers can logically debug their designs before actually building their design either manually or with the aid of automatic tools.

Further Reading

George Boole's original work was published in the middle of the 19th century. Obviously, he did not have computer hardware in mind at the time. Instead, he attempted to develop a mathematical basis for logic. The basic axioms that we presented in Section 2.1 are called Huntington's axioms. These were published by E. V. Huntington in a paper entitled "Sets of Independent Postulates for the Algebra of Logic," in *Transactions of the American Mathematical Society* (Volume 5) in 1904. C. E. Shannon was the first to show how Boolean algebra could be applied to digital design in his landmark paper "A Symbolic Analysis of Relay and Switching Circuits," in *Transactions of the AIEE*, 57, 713–723, 1938.

For a detailed presentation of multilevel logic optimization techniques, the following papers are highly recommended: K. Bartlett, W. Cohen, A. DeGeus, G. Hachtel, "Synthesis and Optimization of Multi-Level Logic under Timing Constraints," *IEEE Transactions on Computer-Aided Design*, CAD-5, 4, 582–596 (October 1986), and R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A. R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on Computer-Aided Design*, CAD-6, 6, 1062–1081 (November 1987).

All digital design textbooks describe Boolean simplification in one form or another. For alternative approaches, see J. F. Wakerly, *Digital Design Principles and Practices*, Prentice-Hall, Englewood Cliffs, NJ, 1990; S. H. Unger, *The Essence of Logic Circuits*, Prentice-Hall, Englewood Cliffs, NJ, 1989; and C. H. Roth, Jr., *Fundamentals of Logic Design*, West Publishing Co., St. Paul, MN, 1985.

There are many texts on Hardware Description Languages. For a more complete treatment of Verilog, see D. Thomas, P. Moorby, The Verilog Hardware Description Language, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991, for VHDL, see P. Ashenden, The Student's Guide to VHDL, Morgan-Kaufmann, San Francisco, CA, 1998. Several also treat the subject or writing HDL descriptions targetting the use of automated synthesis tools to realize the design: E.

Sternheim, R. Singh, R. Madhavan, Y. Trivedi, Digital Design and Synthesis with Verilog HDL, Automata Publishing, San Jose, CA, 1993 or D. Smith, P. Franzon, Verilog Styles for Synthesis of Digital Systems, Prentice-Hall, Upper Saddle River, NJ, 2000. Finally, a text with good side-to-side comparisons of Verilog and VHDL is D. Smith, HDL Chip Design, Doone Publications, Madison, AL, 1996.

Exercises

Nothing should be made by man's labor which is not worth making, or which must be made by labor degrading to the makers.

—William Morris

2.1 (*Gate Logic*) Draw schematics for the following functions in terms of AND, OR, and inverter gates.

a. $X\ (Y + Z)$

b. $X\ Y + X\ Z$

c. $\overline{X(Y + Z)}$

d. $\overline{X} + \overline{Y}\overline{Z}$

e. $W\ (X + Y\ Z)$

2.2 (*Gate Logic*) Draw the schematics for the following functions using NOR gates and inverters only:

a. $\overline{[\overline{X} + (\overline{Y + Z})]}$

b. $\overline{[(\overline{\overline{X} + \overline{Y}}) + (\overline{X + \overline{Z}})]}$

2.3 (*Gate Logic*) Draw the schematics for the following functions using NAND gates and inverters only:

a. $\overline{[X(\overline{Y}\overline{Z})]}$

b. $XY + XZ$

2.4 (*Gate Logic*) Draw switch networks for the XOR and XNOR Boolean operators.

2.5 (*Gate Logic*) Design a hall light circuit to the following specification. There is a switch at either end of a hall that controls a single light. If the light is off, changing the position of either switch causes the light to turn on. Similarly, if the light is on, changing the position of either switch causes the light to turn off. Write your assumptions, derive a truth table, and describe how to implement this function in terms of logic gates or switching networks.

2.6 (*Laws and Theorems of Boolean Algebra*) Use switching diagrams to demonstrate the validity of the following simplification theorems:

a. $X + X\ Y = X$

b. $(X + \overline{Y})Y\ =\ XY$

c. $\overline{(X + Y)}\ =\ \overline{X} \bullet \overline{Y}$

2.7 (*Laws and Theorems of Boolean Algebra*) Prove the following simplification theorems using the first eight laws of Boolean algebra:

a. $(X + Y)(X + \overline{Y}) = X$

b. $X\,(X + Y) = X$

c. $(X + \overline{Y})Y = XY$

d. $(X + Y)(\overline{X} + Z) = XZ + X\overline{Y}$

2.8 (*Laws and Theorems of Boolean Algebra*) Verify that

a. OR and AND are duals of each other.

b. NOR and NAND are duals of each other.

c. XNOR and XOR are duals of each other.

d. XNOR is the complement of XOR:

$$\overline{(X\overline{Y} + \overline{X}Y)} = \overline{X}\overline{Y} + XY$$

2.9 (*Laws and Theorems of Boolean Algebra*) Prove, using truth tables, that

$$XY + YZ + \overline{X}Z = XY + \overline{X}Z$$

2.10 (*Laws and Theorems of Boolean Algebra*) Use DeMorgan's theorem to compute the complement of the following Boolean expressions:

a. $A\,(B + CD)$

b. $ABC + B(\overline{C} + \overline{D})$

c. $\overline{X} + \overline{Y}$

d. $X + Y\overline{Z}$

e. $(X + Y)Z$

f. $X + \overline{YZ}$

g. $X(Y + Z\overline{W} + \overline{V}S)$

2.11 (*Laws and Theorems of Boolean Algebra*) Form the complement of the following functions:

a. $f(A,B,C,D) = [A + \overline{BCD}][\overline{AD} + B(\overline{C} + A)]$

b. $f(A,B,C,D) = \overline{A}BC + (\overline{A} + B + D)(AB\overline{D} + \overline{B})$

2.12 (*Laws and Theorems of Boolean Algebra*) Using Boolean algebra, verify that the schematic of Figure Ex2.12 implements an XOR function.

2.13 (*Boolean Simplification*) Simplify the following functions using the theorems of Boolean algebra. Write the particular law or theorem you are using in each step. For each simplified function you derive, how many literals does it have?

a. $f(X,Y) = XY + X\overline{Y}$

b. $f(X,Y) = (X + Y)(X + \overline{Y})$

c. $f(X,Y,Z) = Y\overline{Z} + \overline{X}YZ + XYZ$

d. $f(X,Y,Z) = (X + Y)(\overline{X} + Y + Z)(\overline{X} + Y + \overline{Z})$

e. $f(W,X,Y,Z) = X + XYZ + \overline{X}YZ + \overline{X}Y + WX + \overline{W}X$

2.14 (*Boolean Simplification*) Consider the function $f(A,B,C,D) = (AD + \overline{A}C)[\overline{B}(C + B\overline{D})]$.

a. Draw its schematic using AND, OR, and inverter gates.

b. Using Boolean algebra, put the function into its minimized form and draw the resulting schematic.

2.15 (*Canonical Forms*) Consider the function $f(A,B,C,D) = \Sigma\, m(0,1,2,7,8,9,10,15)$.

a. Write this as a Boolean expression in canonical minterm form.

b. Rewrite the expression in canonical maxterm form.

c. Write the complement of f in "little m" notation and as a canonical minterm expression.

d. Write the complement of f in "big M" notation and as a canonical maxterm expression.

2.16 (*Canonical Forms and Boolean Simplification*) Given the following function in product of sums form, not necessarily minimized:

$$F(W,X,Y,Z) = (W + \overline{X} + \overline{Y})(\overline{W} + \overline{Z})(W + Y)$$

a. Express the function in the canonical sum of products form. Use "little m" notation.

b. Reexpress the function in minimized sum of products form.

c. Express $\overline{F}$ in minimized sum of products form.

d. Reexpress $\overline{F}$ in minimized product of sums form.

2.17 (*Combinational Logic Design*) Consider a five-input Boolean function that is asserted whenever exactly two of its inputs are asserted.

a. Construct its truth table.

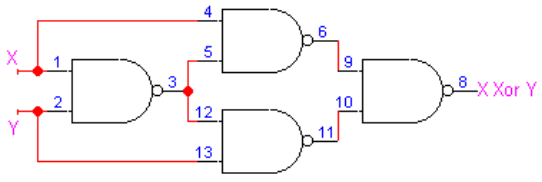


Figure Ex2.12 XOR implemented by NAND gates.

b. What is the function in sum of products form, using "little *m*" notation?

c. What is the function in product of sums form, using "big *M*" notation?

2.18 (*Combinational Logic Design*) In this chapter, we've examined the BCD increment by 1 function. Now consider a binary increment by 1 function defined over the 4-bit binary numbers 0000 through 1111.

a. Fill in the truth table for the function.

2.19 (*Combinational Logic Design*) In this chapter, we've examined a 2-bit binary adder circuit. Now consider a 2-bit binary subtractor, defined as follows. The inputs *A*, *B* and *C*, *D* form the two 2-bit numbers N_1 and N_2. The circuit will form the difference $N_1 - N_2$ on the output bits *F* (most significant) and *G* (least significant). Assume that the circuit never sees an input combination in which N_1 is less than N_2. The output bits are don't cares in these cases.

a. Fill in the four-variable truth table for *F* and *G*.

2.20 (*Combinational Logic Design*) Consider a four-input function that outputs a 1 whenever an odd number of its inputs are 1.

a. Fill in the truth table for the function.

2.21 (*Combinational Logic Design*) Design a combinational circuit with three data inputs D_2, D_1, D_0, two control inputs C_1, C_0, and two outputs R_1, R_0. R_1 and R_0 should be the remainder after dividing the binary number formed from D_2, D_1, D_0 by the number formed by C_1, C_0. For example, if D_2, D_1, $D_0 = 111$ and C_1, $C_0 = 10$, then R_1, $R_0 = 01$ (that is, the remainder of 7 divided by 2 is 1). Note that division by zero will never be requested.

a. Fill in truth tables for the combinational logic functions R_1 and R_0.

2.22 (*Conversion Between Forms*) Use Boolean algebra to verify the following:

a. The AND-OR expression of Figure 2.34 is equivalent to the NAND/NAND expression of that figure.

b. The AND-OR expression of Figure 2.35 is equivalent to the NOR/NOR expression of that figure.

c. The OR/AND expression of Figure 2.36 is equivalent to the NOR/NOR expression of that figure.

d. The OR/AND expression of Figure 2.37 is equivalent to the NAND/NAND expression of that figure.

2.23 (*AND-OR/NAND-NAND Mappings*) Draw schematics for the following expressions, mapped into NAND-only networks. You may assume that literals and their complements are available:

a. $A\overline{B}\overline{C} + \overline{A}C + \overline{A}B$

b. $(\overline{A} + \overline{B} + \overline{C})(\overline{A} + \overline{B})(\overline{A} + \overline{C})$

c. $\overline{A}B + A + \overline{C} + \overline{D}$

d. $(\overline{AB})(\overline{\overline{A}\overline{C}})$

e. $\overline{AB + \overline{A}\overline{C}}$

2.24 (*OR-AND/NOR-NOR Mappings*) Draw schematics for the following expressions, mapped into NOR-only networks. You may assume that literals and their complements are available:

a. $(A + B)(\overline{A} + C)$

b. $\overline{(A + B) \bullet (\overline{A} + C)}$

c. $\overline{(\overline{A + B}) \bullet (\overline{\overline{A} + C})}$

d. $(A + B) \bullet (\overline{A} + C + D) \bullet (\overline{A} + \overline{C})$

e. $(A + B) \bullet (\overline{B} \bullet C) \bullet (\overline{A} + \overline{C})$

2.25 (*Multilevel Network Mappings*) Draw schematics for the following expressions, using mixed NAND and NOR gates only:

a. $(AB + CD)E + F$

b. $(AB + C)E + DG$

c. $\{A + [(B + C)(D + E)]\}\{[(F + G)(\overline{B} + \overline{E})] + \overline{A}\}$

d. $(A + B)(C + D) + EF$

e. $A\overline{B}(\overline{B} + C)\overline{D} + \overline{A}$

2.26 (*Canonical Forms*) Given the following function in sum of products form (not necessarily minimized):

$$F(A, B, C, D) = A\overline{B}C + AD + AC$$

Reexpress the function in:

a. Canonical product of sums form. Use $\Pi\, M$ notation.

b. Minimized product of sums form.

c. $\overline{F}$ in minimized product of sums form.

d. $\overline{F}$ in minimized sum of products form.

e. Implement F and $\overline{F}$ using NAND gates only. You may assume that literals and their complements are available.

f. Implement F and $\overline{F}$ using NOR gates only. You may assume that literals and their complements are available.

2.27 (*Multilevel Logic*) Factor the following sum of products expressions:

a. $ABCD + ABDE$

b. $ACD + BC + ABE + BD$

c. $AC + ADE + BC + BDE$

d. $AD + AE + BD + BE + CD + CE + AF$

e. $ACE + ACF + ADE + ADF + BCE + BCF + BDE + BDF$

2.28 (*Multilevel Logic*) Write down the function represented by the circuit network in Figure Ex2.28 in a multilevel factored form using AND, OR, and NOT operations only—that is, no NAND or NOR operations:

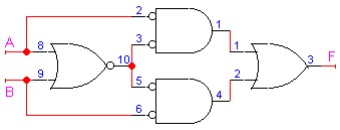


Figure Ex2.28

Derive the simplest Boolean expression (minimum number of literals and fewest gates) for the function represented by this schematic.

2.29 (*Multilevel Logic*) Using Boolean algebra or truth tables, verify that the multilevel forms for the full adder *Sum* and *CO* (carry-out) obtained in Section X are logically equivalent to the two-level forms.

2.30 (*Multilevel Logic*) Using Boolean algebra or truth tables, verify that the multilevel forms for the 2-bit binary adder outputs, *X*, *Y*, and *Z*, of Section X are logically equivalent to the two-level forms.

2.31 (*Multilevel Logic*) Using Boolean algebra or truth tables, verify that the multilevel forms for the BCD increment by 1 outputs, *W*, *X*, *Y*, and *Z*, of Section X are logically equivalent to the two-level forms.

2.32 (*Time Response*) Consider the circuit in Figure Ex2.32(a). Write down its functions in minimized form. Given that XOR/XNOR gates have twice the delay of the NAND gates, what is the circuit's output response to the input waveforms in Figure Ex2.32(b)? (Each 5-time-unit division represents one NAND gate delay.)

2.33 (*Time Response*) Consider the circuit with a single input in Figure Ex2.33(a). At time t_0 the switch is moved to the closed (connected) position, and at time t_1 the switch is returned to its original open (disconnected) position. Fill in a timing diagram showing the behavior of the internal signals *B* and *C*, and the output signal, in response to this input waveform. Assume all gates have an identical propagation delay T_{pd}, which corresponds to a single division on the chart in Figure Ex2.33(b).

2.34 (*Time Response*) Construct a timing diagram for the behavior of the circuit schematic in Figure Ex2.34.

a. Start by finding a nonoscillating starting condition for the circuit with switch S in position 1 (up) as shown. Fill in the timing waveform with an initial steady-state condition for the circuit nodes labeled *A*, *B*, *C*,

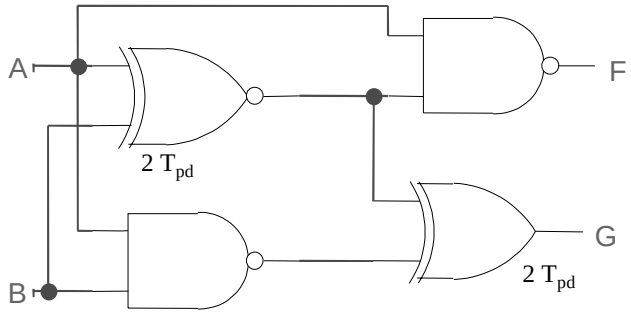


Figure Ex2.32

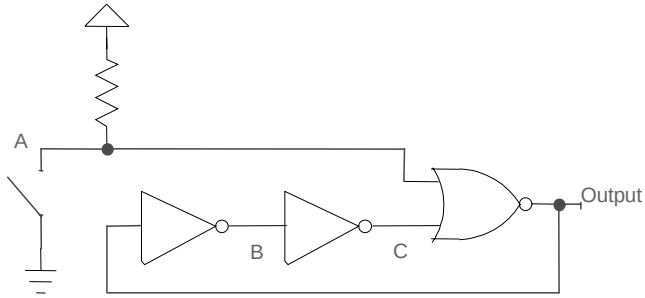


Figure Ex2.33

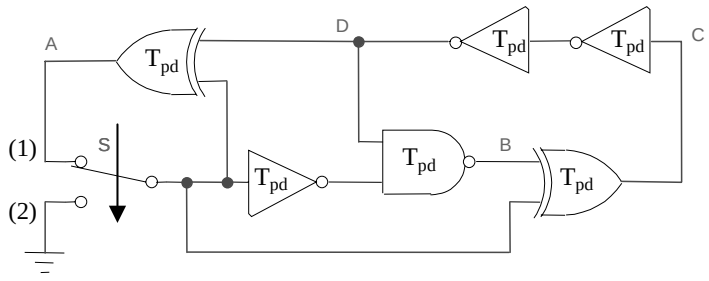


Figure Ex2.34

and *D*. *Warning:* It is very easy to choose an initial configuration that oscillates. A unique nonoscillating configuration does exist. Start your reasoning with the tightest loop, or make an educated guess and verify that the assumed state is indeed nonoscillating.

b. At time *T*, the switch is moved from position 1 to position 2 (down). Fill in the rest of the timing diagram with the logic values of the signals at points *A*, *B*, *C*, and *D* in the given circuit.