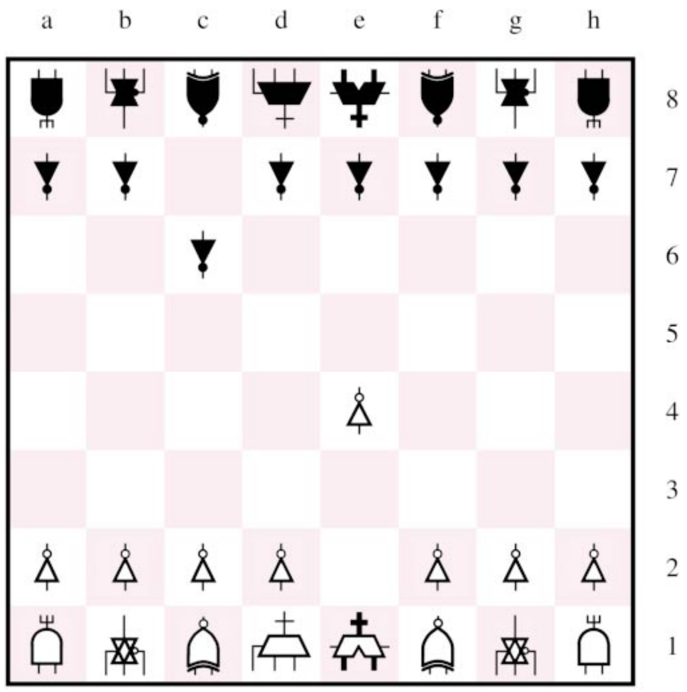
chapter

1

Design Concepts



1. e2–e4, c7–c6

This book is about logic circuits—the circuits from which computers are built. Proper understanding of logic circuits is vital for today's electrical and computer engineers. These circuits are the key ingredient of computers and are also used in many other applications. They are found in commonly used products, such as digital watches, various household appliances, CD players, and electronic games, as well as in large systems, such as the equipment for telephone and television networks.

The material in this book will introduce the reader to the many issues involved in the design of logic circuits. It explains the key ideas with simple examples and shows how complex circuits can be derived from elementary ones. We cover the classical theory used in the design of logic circuits in great depth because it provides the reader with an intuitive understanding of the nature of such circuits. But throughout the book we also illustrate the modern way of designing logic circuits, using sophisticated *computer aided design (CAD)* software tools. The CAD methodology adopted in the book is based on the industry-standard design language called VHDL. Design with VHDL is first introduced in Chapter 2, and usage of VHDL and CAD tools is an integral part of each chapter in the book.

Logic circuits are implemented electronically, using transistors on an integrated circuit chip. With modern technology it is possible to fabricate chips that contain tens of millions of transistors, as in the case of computer processors. The basic building blocks for such circuits are easy to understand, but there is nothing simple about a circuit that contains tens of millions of transistors. The complexity that comes with the large size of logic circuits can be handled successfully only by using highly organized design techniques. We introduce these techniques in this chapter, but first we briefly describe the hardware technology used to build logic circuits.

1.1 Digital Hardware

Logic circuits are used to build computer hardware, as well as many other types of products. All such products are broadly classified as *digital hardware*. The reason that the name *digital* is used will become clear later in the book—it derives from the way in which information is represented in computers, as electronic signals that correspond to digits of information.

The technology used to build digital hardware has evolved dramatically over the past four decades. Until the 1960s logic circuits were constructed with bulky components, such as transistors and resistors that came as individual parts. The advent of integrated circuits made it possible to place a number of transistors, and thus an entire circuit, on a single chip. In the beginning these circuits had only a few transistors, but as the technology improved they became larger. Integrated circuit chips are manufactured on a silicon wafer, such as the one shown in Figure 1.1. The wafer is cut to produce the individual chips, which are then placed inside a special type of chip package. By 1970 it was possible to implement all circuitry needed to realize a microprocessor on a single chip. Although early microprocessors had modest computing capability by today's standards, they opened the door for the information processing revolution by providing the means for implementation of affordable personal computers. About 30 years ago Gordon Moore, chairman of Intel Corporation, observed that integrated circuit technology was progressing at an astounding rate, doubling the number of transistors that could be placed on a chip every 1.5 to 2 years.

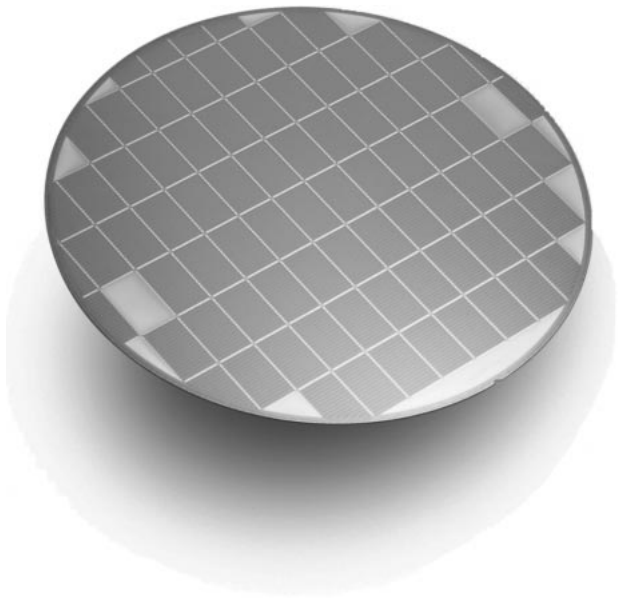


Figure 1.1 A silicon wafer (courtesy of Altera Corp.).

This phenomenon, informally known as *Moore's law*, continues to the present day. Thus in the early 1990s microprocessors could be manufactured with a few million transistors, and by the late 1990s it has become possible to fabricate chips that contain more than 10 million transistors.

Moore's law is expected to continue to hold true for at least the next decade. A consortium of integrated circuit manufacturers called the Semiconductor Industry Association (SIA) produces an estimate of how the technology is expected to evolve. Known as the *SIA Roadmap* [1], this estimate predicts the minimum size of a transistor that can be fabricated on an integrated circuit chip. The size of a transistor is measured by a parameter called its *gate length*, which we will discuss in Chapter 3. A sample of the SIA Roadmap is given in Table 1.1. In 1999 the minimum possible gate length that can be reliably manufactured is 0.14 μm. The first row of the table indicates that the minimum gate length is expected to reduce steadily to about 0.035 μm by the year 2012. The size of a transistor determines how many transistors can be placed in a given amount of chip area, with the current maximum being about 14 million transistors per cm^2. This number is expected to grow to 100 million transistors by the year 2012. The largest chip size is expected to be about 1300 mm^2 at that time; thus chips with up to 1.3 billion transistors will be possible! There is no doubt that this technology will have a huge impact on all aspects of people's lives.

The designer of digital hardware may be faced with designing logic circuits that can be implemented on a single chip or, more likely, designing circuits that involve a number of chips placed on a *printed circuit board (PCB)*. Frequently, some of the logic circuits can be realized in existing chips that are readily available. This situation simplifies the design task and shortens the time needed to develop the final product. Before we discuss the design

April 5, 1999 13:51 g02-ch1 Sheet number 4 Page number 4 black

Table 1.1 A sample of the SIA Roadmap

	Year					
	1999	**2001**	**2003**	**2006**	**2009**	**2012**
Transistor gate length	0.14 μm	0.12 μm	0.10 μm	0.07 μm	0.05 μm	0.035 μm
Transistors per cm^2	14 million	16 million	24 million	40 million	64 million	100 million
Chip size	800 mm^2	850 mm^2	900 mm^2	1000 mm^2	1100 mm^2	1300 mm^2

process in more detail, we should introduce the different types of integrated circuit chips that may be used.

There exists a large variety of chips that implement various functions that are useful in the design of digital hardware. The chips range from very simple chips with low functionality to extremely complex chips. For example, a digital hardware product may require a microprocessor to perform some arithmetic operations, memory chips to provide storage capability, and interface chips that allow easy connection to input and output devices. Such chips are available from various vendors.

For most digital hardware products, it is also necessary to design and build some logic circuits from scratch. For implementing these circuits, three main types of chips may be used: standard chips, programmable logic devices, and custom chips. These are discussed next.

1.1.1 Standard Chips

Numerous chips are available that realize some commonly used logic circuits. We will refer to these as *standard chips*, because they usually conform to an agreed-upon standard in terms of functionality and physical configuration. Each standard chip contains a small amount of circuitry (usually involving fewer than 100 transistors) and performs a simple function. To build a logic circuit, the designer chooses the chips that perform whatever functions are needed and then defines how these chips should be interconnected to realize a larger logic circuit.

Standard chips were popular for building logic circuits until the early 1980s. However, as integrated circuit technology improved, it became inefficient to use valuable space on PCBs for chips with low functionality. Another drawback of standard chips is that the functionality of each chip is fixed and cannot be changed.

1.1.2 Programmable Logic Devices

In contrast to standard chips that have fixed functionality, it is possible to construct chips that contain circuitry that can be configured by the user to implement a wide range of different logic circuits. These chips have a very general structure and include a collec-

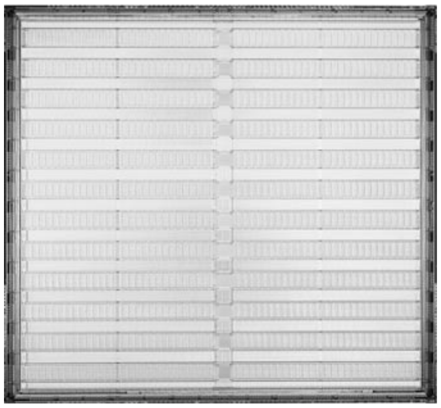


Figure 1.2 A field-programmable gate array chip (courtesy of Altera Corp.).

tion of *programmable switches* that allow the internal circuitry in the chip to be configured in many different ways. The designer can implement whatever functions are needed for a particular application by choosing an appropriate configuration of the switches. The switches are programmed by the end user, rather than when the chip is manufactured. Such chips are known as *programmable logic devices (PLDs)*. We will introduce them in Chapter 3.

Most types of PLDs can be programmed multiple times. This capability is advantageous because a designer who is developing a prototype of a product can program a PLD to perform some function, but later, when the prototype hardware is being tested, can make corrections by reprogramming the PLD. Reprogramming might be necessary, for instance, if a designed function is not quite as intended or if new functions are needed that were not contemplated in the original design.

PLDs are available in a wide range of sizes. They can be used to realize much larger logic circuits than a typical standard chip can realize. Because of their size and the fact that they can be tailored to meet the requirements of a specific application, PLDs are widely used today. One of the most sophisticated types of PLD is known as a *field-programmable gate array (FPGA)*. FPGAs that contain more than 100 million transistors will soon be available [2,3]. A photograph of an FPGA chip that has 10 million transistors is shown in Figure 1.2. The chip consists of a large number of small logic circuit elements, which can be connected together using the programmable switches. The logic circuit elements are arranged in a regular two-dimensional structure.

1.1.3 Custom-Designed Chips

PLDs are available as off-the-shelf components that can be purchased from different suppliers. Because they are programmable, they can be used to implement most logic circuits found in digital hardware. However, PLDs also have a drawback in that the programmable switches consume valuable chip area and limit the speed of operation of implemented circuits. Thus in some cases PLDs may not meet the desired performance or cost objectives. In such situations it is possible to design a chip from scratch; namely, the logic circuitry that must be included on the chip is designed first and then an appropriate technology is chosen to implement the chip. Finally, the chip is manufactured by a company that has the fabrication facilities. This approach is known as *custom* or *semi-custom design*, and such chips are called *custom* or *semi-custom chips*. Such chips are intended for use in specific applications and are sometimes called *application-specific integrated circuits (ASICs)*.

The main advantage of a custom chip is that its design can be optimized for a specific task; hence it usually leads to better performance. It is possible to include a larger amount of logic circuitry in a custom chip than would be possible in other types of chips. The cost of producing such chips is high, but if they are used in a product that is sold in large quantities, then the cost per chip, amortized over the total number of chips fabricated, may be lower than the total cost of off-the-shelf chips that would be needed to implement the same function(s). Moreover, if a single chip can be used instead of multiple chips to achieve the same goal, then a smaller area is needed on a PCB that houses the chips in the final product. This results in a further reduction in cost.

A disadvantage of the custom-design approach is that manufacturing a custom chip often takes a considerable amount of time, on the order of months. In contrast, if a PLD can be used instead, then the chips are programmed by the end user and no manufacturing delays are involved.

1.2 THE DESIGN PROCESS

The availability of computer-based tools has greatly influenced the design process in a wide variety of design environments. For example, designing an automobile is similar in the general approach to designing a furnace or a computer. Certain steps in the development cycle must be performed if the final product is to meet the specified objectives. We will start by introducing a typical development cycle in the most general terms. Then we will focus on the particular aspects that pertain to the design of logic circuits.

The flowchart in Figure 1.3 depicts a typical development process. We assume that the process is to develop a product that meets certain expectations. The most obvious requirements are that the product must function properly, that it must meet an expected level of performance, and that its cost should not exceed a given target.

The process begins with the definition of product specifications. The essential features of the product are identified, and an acceptable method of evaluating the implemented features in the final product is established. The specifications must be tight enough to ensure that the developed product will meet the general expectations, but should not be unnecessarily constraining (that is, the specifications should not prevent design choices that may lead to unforeseen advantages).

From a complete set of specifications, it is necessary to define the general structure of an initial design of the product. This step is difficult to automate. It is usually performed by a human designer because there is no clear-cut strategy for developing a product's overall structure—it requires considerable design experience and intuition.

After the general structure is established, CAD tools are used to work out the details. Many types of CAD tools are available, ranging from those that help with the design

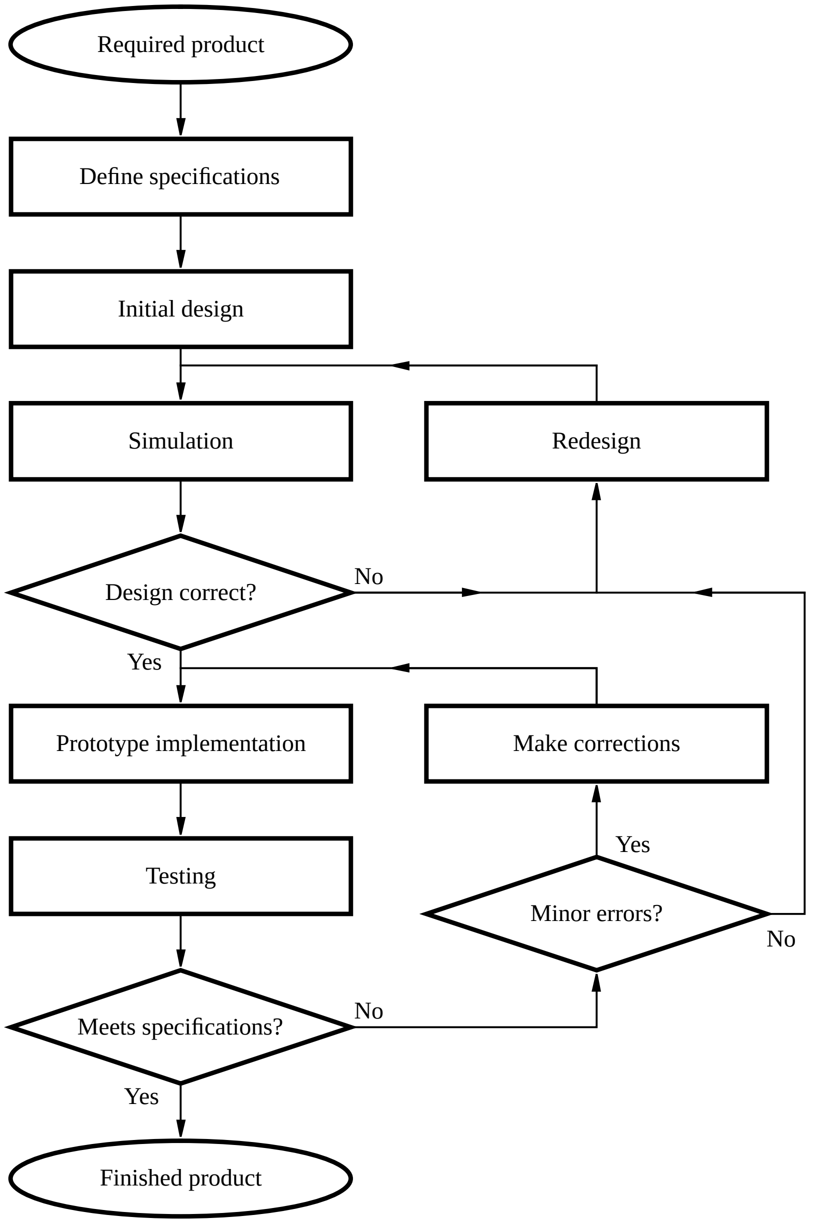


Figure 1.3 The development process.

of individual parts of the system to those that allow the entire system's structure to be represented in a computer. When the initial design is finished, the results must be verified against the original specifications. Traditionally, before the advent of CAD tools, this step involved constructing a physical model of the designed product, usually including just the key parts. Today it is seldom necessary to build a physical model. CAD tools enable designers to simulate the behavior of incredibly complex products, and such simulations are used to determine whether the obtained design meets the required specifications. If errors are found, then appropriate changes are made and the verification of the new design is repeated through simulation. Although some design flaws may escape detection via simulation, usually all but the most subtle problems are discovered in this way.

When the simulation indicates that the design is correct, a complete physical prototype of the product is constructed. The prototype is thoroughly tested for conformance with the specifications. Any errors revealed in the testing must be fixed. The errors may be minor, and often they can be eliminated by making small corrections directly on the prototype of the product. In case of large errors, it is necessary to redesign the product and repeat the steps explained above. When the prototype passes all the tests, then the product is deemed to be successfully designed and it can go into production.

1.3 Design of Digital Hardware

Our previous discussion of the development process is relevant in a most general way. The steps outlined in Figure 1.3 are fully applicable in the development of digital hardware. Before we discuss the complete sequence of steps in this development environment, we should emphasize the iterative nature of the design process.

1.3.1 Basic Design Loop

Any design process comprises a basic sequence of tasks that are performed in various situations. This sequence is presented in Figure 1.4. Assuming that we have an initial concept about what should be achieved in the design process, the first step is to generate an initial design. This step often requires a lot of manual effort because most designs have some specific goals that can be reached only through the designer's knowledge, skill, and intuition. The next step is the simulation of the design at hand. There exist excellent CAD tools to assist in this step. To carry out the simulation successfully, it is necessary to have adequate input conditions that can be applied to the design that is being simulated and later to the final product that has to be tested. Applying these input conditions, the simulator tries to verify that the designed product will perform as required under the original product specifications. If the simulation reveals some errors, then the design must be changed to overcome the problems. The redesigned version is again simulated to determine whether the errors have disappeared. This loop is repeated until the simulation indicates a successful design. A prudent designer expends considerable effort to remedy errors during simulation because errors are typically much harder to fix if they are discovered late in the design process. Even so, some errors may not be detected during simulation, in which case they have to be dealt with in later stages of the development cycle.

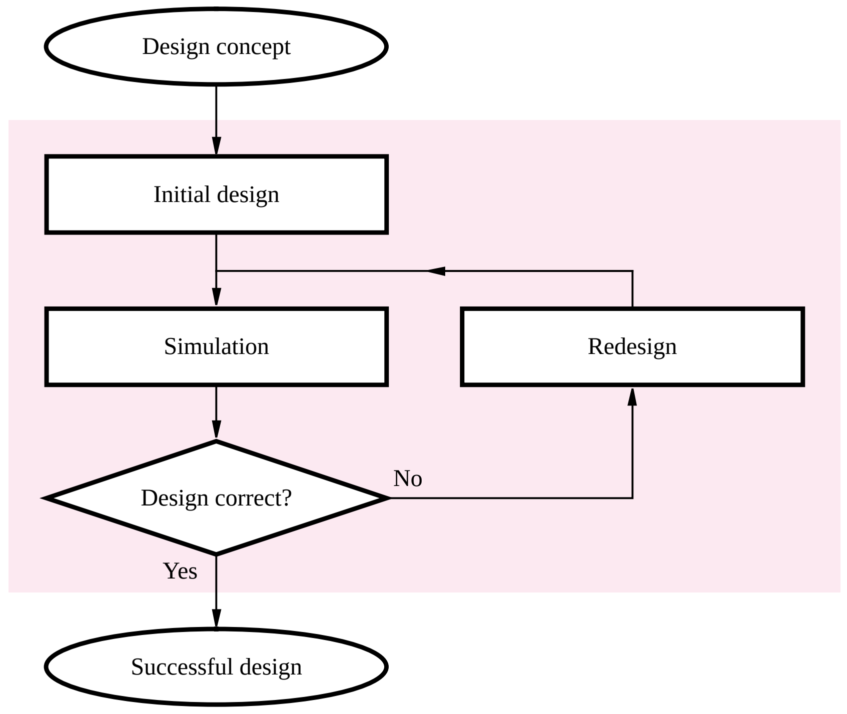


Figure 1.4 The basic design loop.

1.3.2 Design of a Digital Hardware Unit

Digital hardware products usually involve one or more PCBs that contain many chips and other components. Development of such products starts with the definition of the overall structure. Then the required integrated circuit chips are selected, and the PCBs that house and connect the chips together are designed. If the selected chips include PLDs or custom chips, then these chips must be designed before the PCB-level design is undertaken. Since the complexity of circuits implemented on individual chips and on the circuit boards is usually very high, it is essential to make use of good CAD tools.

An example of a PCB is given in Figure 1.5. The PCB is a part of a large computer system designed at the University of Toronto. This computer, called *NUMAchine* [4,5], is a *multiprocessor*, which means that it contains many processors that can be used together to work on a particular task. The PCB in the figure contains one processor chip and various memory and support chips. Complex logic circuits are needed to form the interface between the processor and the rest of the system. A number of PLDs are used to implement these logic circuits.

To illustrate the complete development cycle in more detail, we will consider the steps needed to produce a digital hardware unit that can be implemented on a PCB. This hardware

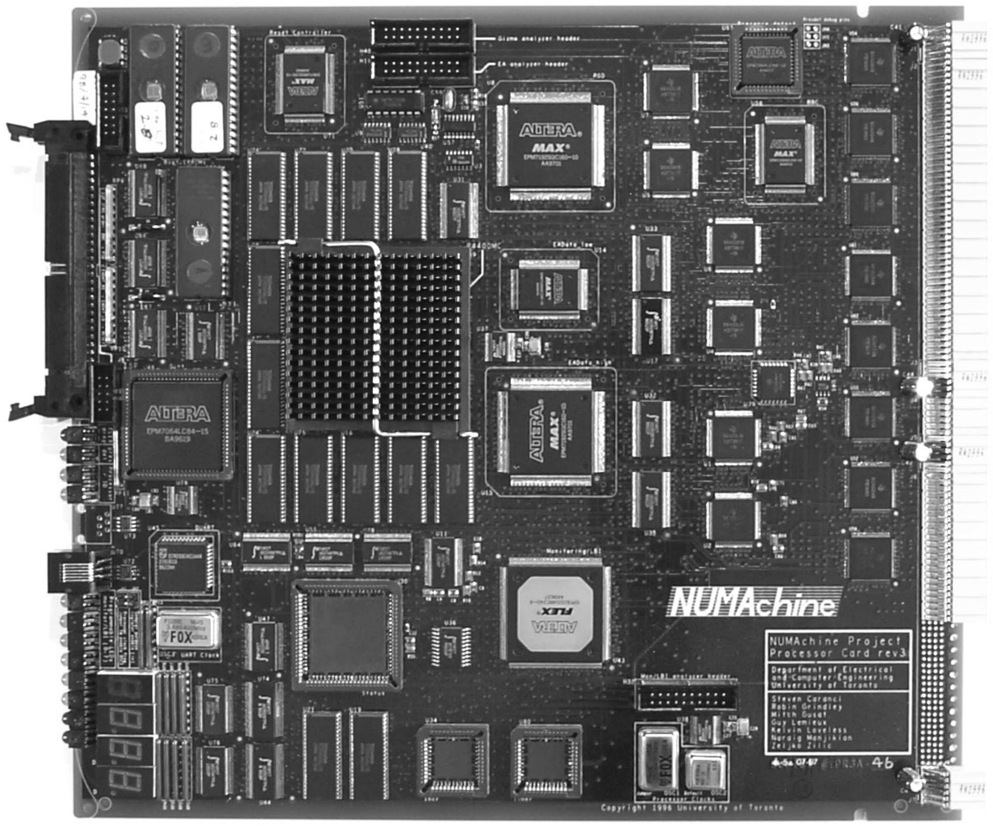

NUMAchine
NUMAchine Project
Processor Card rev3
Department of Electrical and Computer Engineering
University of Toronto
Copyright 1996 University of Toronto

Figure 1.5 A printed circuit board.

could be viewed as a very complex logic circuit that performs the functions defined by the product specifications. Figure 1.6 shows the design flow, assuming that we have a design concept that defines the expected behavior and characteristics of this large circuit.

An orderly way of dealing with the complexity involved is to partition the circuit into smaller blocks and then to design each block separately. Breaking down a large task into more manageable smaller parts is known as the divide-and-conquer approach. The design of each block follows the procedure outlined in Figure 1.4. The circuitry in each block is defined, and the chips needed to implement it are chosen. The operation of this circuitry is simulated, and any necessary corrections are made.

Having successfully designed all blocks, the interconnection between the blocks must be defined, which effectively combines these blocks into a single large circuit. Now it is necessary to simulate this complete circuit and correct any errors. Depending on the errors encountered, it may be necessary to go back to the previous steps as indicated by the paths A, B, and C in the flowchart. Some errors may be caused by incorrect connections

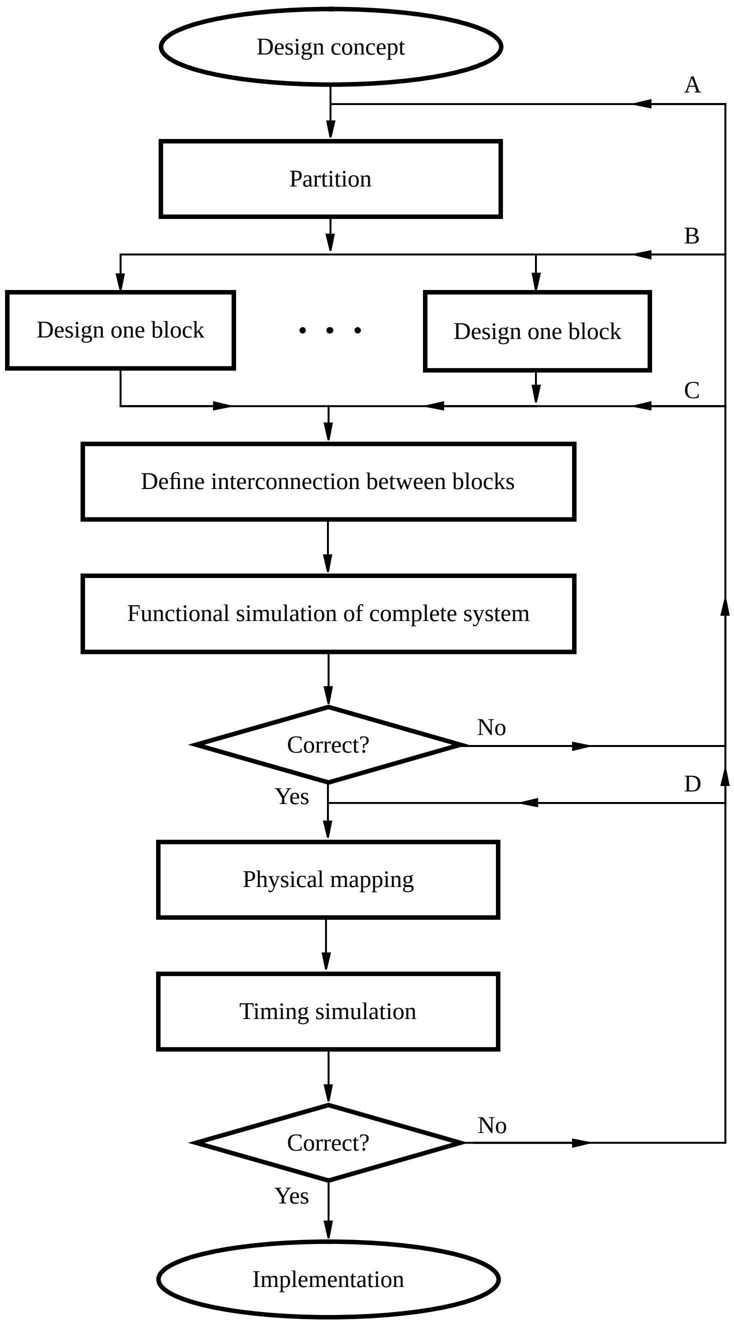


Figure 1.6 Design flow for logic circuits.

between the blocks, in which case these connections have to be redefined, following path C. Some blocks may not have been designed correctly, in which case path B is followed and the erroneous blocks are redesigned. Another possibility is that the very first step of partitioning the overall large circuit into blocks was not done well, in which case path A is followed. This may happen, for example, if none of the blocks implement some functionality needed in the complete circuit.

Successful completion of functional simulation suggests that the designed circuit will correctly perform all of its functions. The next step is to decide how to realize this circuit on a PCB. The physical location of each chip on the board has to be determined, and the wiring pattern needed to make connections between the chips has to be defined. We refer to this step as the *physical design* of the PCB. CAD tools are relied on heavily to perform this task automatically.

Once the placement of chips and the actual wire connections on the PCB have been established, it is desirable to see how this physical layout will affect the performance of the circuit on the finished board. It is reasonable to assume that if the previous functional simulation indicated that all functions will be performed correctly, then the CAD tools used in the physical design step will ensure that the required functional behavior will not be corrupted by placing the chips on the board and wiring them together to realize the final circuit. However, even though the functional behavior may be correct, the realized circuit may operate more slowly than desired and thus lead to inadequate performance. This condition occurs because the physical wiring on the PCB involves metal traces that present resistance and capacitance to electrical signals and thus may have a significant impact on the speed of operation. To distinguish between simulation that considers only the functionality of the circuit and simulation that also considers timing behavior, it is customary to use the terms *functional simulation* and *timing simulation*. A timing simulation may reveal potential performance problems, which can then be corrected by using the CAD tools to make changes in the physical design of the PCB.

Having completed the design process, the designed circuit is ready for physical implementation. The steps needed to implement a prototype board are indicated in Figure 1.7. A first version of the board is built and tested. Most minor errors that are detected can usually be corrected by making changes directly on the prototype board. This may involve changes in wiring or perhaps reprogramming some PLDs. Larger problems require a more substantial redesign. Depending on the nature of the problem, the designer may have to return to any of the points A, B, C, or D in the design process of Figure 1.6.

We have described the development process where the final circuit is implemented using many chips on a PCB. The material presented in this book is directly applicable to this type of design problem. However, for practical reasons the design examples that appear in the book are relatively small and can be realized in a single integrated circuit, either a custom-designed chip or a PLD. All the steps in Figure 1.6 are relevant in this case as well, with the understanding that the circuit blocks to be designed are on a smaller scale.

1.4 LOGIC CIRCUIT DESIGN IN THIS BOOK

In this book we use PLDs extensively to illustrate many aspects of logic circuit design. We selected this technology because it is widely used in real digital hardware products

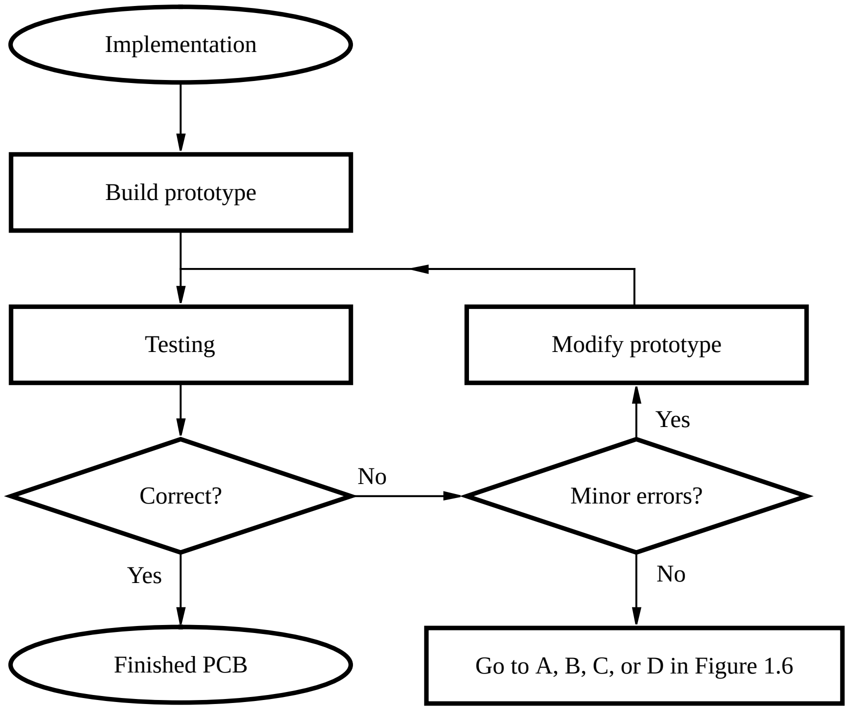


Figure 1.7 Completion of PCB development.

and because the chips are user programmable. PLD technology is particularly well suited for educational purposes because many readers have access to facilities for programming PLDs, which enables the reader to actually implement the sample circuits. To illustrate practical design issues, in this book we use two types of PLDs—they are the two types of devices that are widely used in digital hardware products today. One type is known as *complex programmable logic devices* (CPLDs) and the other as *field-programmable gate arrays* (FPGAs). These chips are introduced in Chapter 3.

We will illustrate the automated design of logic circuits using a sophisticated CAD system from Altera Corporation, one of the world's leading suppliers of PLDs. The system is called *MAX+plusII*. This industrial-quality software supports all phases of the design cycle and is powerful and easy to use. To allow the reader to obtain hands-on experience with the CAD tools, a CD-ROM containing the MAX+plusII software accompanies the book. The software is easily installed on a suitable personal computer, and we provide a sequence of complete step-by-step tutorials to illustrate the proper use of the CAD tools in concert with the book.

For educational purposes Altera provides a laboratory development PCB, which is called the UP-1 board. This PCB, shown in Figure 1.8, contains both a CPLD and an FPGA chip and has an interface for connecting the board to a personal computer. Logic circuits can be designed on the computer using MAX+plusII and then *downloaded* into the PLDs, thus realizing the designed circuit. The reader is encouraged to obtain the board from Altera,

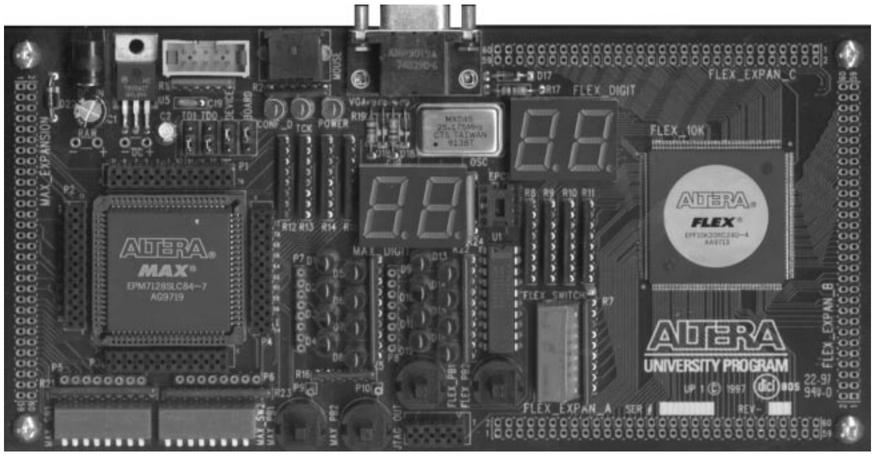


Figure 1.8 The Altera UP-1 laboratory development board.

which can be done by accessing the University Program part of Altera's Web site. All the examples of logic circuits presented in this book can be implemented on the UP-1 board.

1.5 Theory and Practice

Modern design of logic circuits depends heavily on CAD tools, but the discipline of logic design evolved long before CAD tools were invented. This chronology is quite obvious because the very first computers were built with logic circuits, and there certainly were no computers available on which to design them!

Numerous manual design techniques have been developed to deal with logic circuits. Boolean algebra, which we will introduce in Chapter 2, was adopted as a mathematical means for representing such circuits. An enormous amount of "theory" was developed, showing how certain design issues may be treated. To be successful, a designer had to apply this knowledge in practice.

CAD tools not only made it possible to design incredibly complex circuits but also made the design work much simpler in general. They perform many tasks automatically, which may suggest that today's designer need not understand the theoretical concepts used in the tasks performed by CAD tools. An obvious question would then be, Why should one study the theory that is no longer needed for manual design? Why not simply learn how to use the CAD tools?

There are three big reasons for learning the relevant theory. First, although the CAD tools perform the automatic tasks of optimizing a logic circuit to meet particular design objectives, the designer has to give the original description of the logic circuit. If the designer specifies a circuit that has inherently bad properties, then the final circuit will also be of poor quality. Second, the algebraic rules and theorems for design and manipulation of logic circuits are directly implemented in today's CAD tools. It is not possible for a user of the tools to understand what the tools do without grasping the underlying theory. Third, CAD tools offer many optional processing steps that a user can invoke when working on a design. The designer chooses which options to use by examining the resulting circuit produced by the CAD tools and deciding whether it meets the required objectives. The only way that the designer can know whether or not to apply a particular option in a given situation is to know what the CAD tools will do if that option is invoked—again, this implies that the designer must be familiar with the underlying theory. We discuss the classical logic circuit theory extensively in this book, because it is not possible to become an effective logic circuit designer without understanding the fundamental concepts.

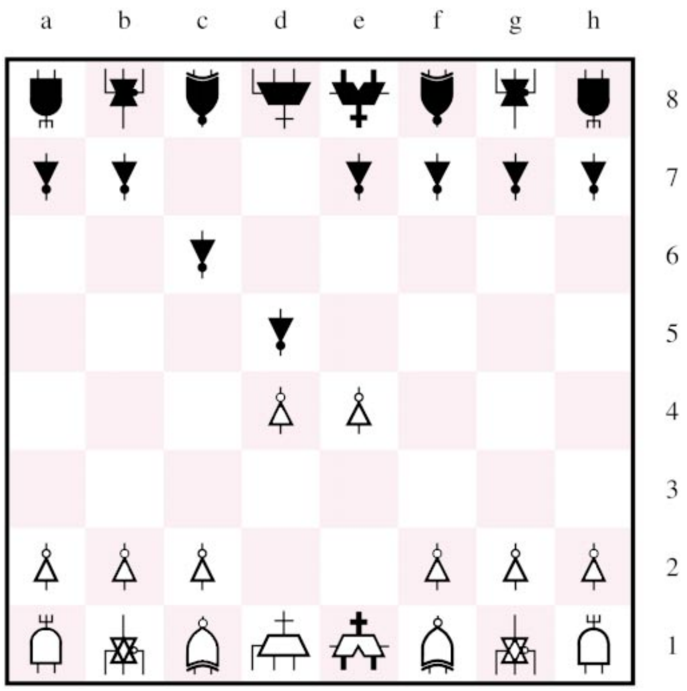
On a final note, there is another good reason to learn some logic circuit theory even if it were not required for CAD tools. Simply put, it is interesting and intellectually challenging. In the modern world filled with sophisticated automatic machinery, it is tempting to rely on tools as a substitute for thinking. However, in logic circuit design, as in any type of design process, computer-based tools are not a substitute for human intuition and innovation. Computer-based tools can produce good digital hardware designs only when employed by a designer who thoroughly understands the nature of logic circuits.

REFERENCES

1. Semiconductor Industry Association, "National Technology Roadmap for Semiconductors," http://www.semichips.org/
2. Altera Corporation, "APEX 20K Advance Information Brief," http://www.altera.com
3. Xilinx Corporation, "Virtex Field Programmable Gate Arrays," http://www.xilinx.com
4. S. Brown, N. Manjikian, Z. Vranesic, S. Caranci, A. Grbic, R. Grindley, M. Gusat, K. Loveless, Z. Zilic, and S. Srbljic, "Experience in Designing a Large-Scale Multiprocessor Using Field-Programmable Devices and Advanced CAD Tools," 33rd IEEE Design Automation Conference, Las Vegas, June 1996.
5. A. Grbic, S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic, " The Design and Implementation of the NUMAchine Multiprocessor," IEEE Design Automation Conference, San Francisco, June 1998.

chapter

2 Introduction to Logic Circuits



2. d2–d4, d7–d5

The study of logic circuits is motivated mostly by their use in digital computers. But such circuits also form the foundation of many other digital systems where performing arithmetic operations on numbers is not of primary interest. For example, in a myriad of control applications actions are determined by some simple logical operations on input information, without having to do extensive numerical computations.

Logic circuits perform operations on digital signals and are usually implemented as electronic circuits where the signal values are restricted to a few discrete values. In *binary* logic circuits there are only two values, 0 and 1. In *decimal* logic circuits there are 10 values, from 0 to 9. Since each signal value is naturally represented by a digit, such logic circuits are referred to as *digital circuits*. In contrast, there exist *analog circuits* where the signals may take on a continuous range of values between some minimum and maximum levels.

In this book we deal with binary circuits, which have the dominant role in digital technology. We hope to provide the reader with an understanding of how these circuits work, how are they represented in mathematical notation, and how are they designed using modern design automation techniques. We begin by introducing some basic concepts pertinent to the binary logic circuits.

2.1 Variables and Functions

The dominance of binary circuits in digital systems is a consequence of their simplicity, which results from constraining the signals to assume only two possible values. The simplest binary element is a switch that has two states. If a given switch is controlled by an input variable x, then we will say that the switch is open if $x = 0$ and closed if $x = 1$, as illustrated in Figure 2.1*a*. We will use the graphical symbol in Figure 2.1*b* to represent such switches in the diagrams that follow. Note that the control input x is shown explicitly in the symbol. In Chapter 3 we will explain how such switches are implemented with transistors.

Consider a simple application of a switch, where the switch turns a small lightbulb on or off. This action is accomplished with the circuit in Figure 2.2*a*. A battery provides the power source. The lightbulb glows when sufficient current passes through its filament, which is an electrical resistance. The current flows when the switch is closed, that is, when $x = 1$. In this example the input that causes changes in the behavior of the circuit is the

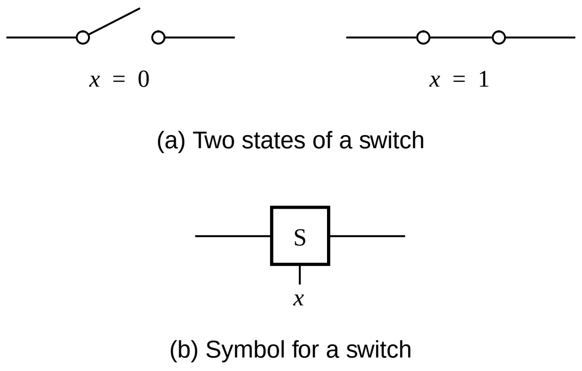


Figure 2.1 A binary switch.

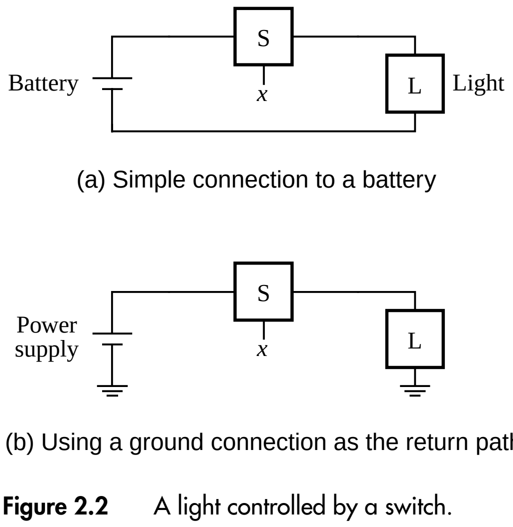

S
Battery
x
L
Light
(a) Simple connection to a battery
S
Power supply
x
L
(b) Using a ground connection as the return path

Figure 2.2 A light controlled by a switch.

switch control x. The output is defined as the state (or condition) of the light L. If the light is on, we will say that $L = 1$. If the the light is off, we will say that $L = 0$. Using this convention, we can describe the state of the light L as a function of the input variable x. Since $L = 1$ if $x = 1$ and $L = 0$ if $x = 0$, we can say that

$$L(x) = x$$

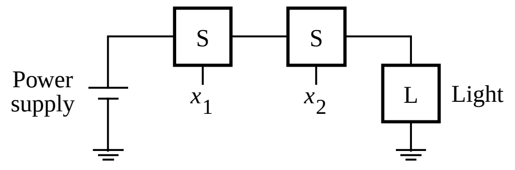
This simple *logic expression* describes the output as a function of the input. We say that $L(x) = x$ is a *logic function* and that x is an *input variable*.

The circuit in Figure 2.2*a* can be found in an ordinary flashlight, where the switch is a simple mechanical device. In an electronic circuit the switch is implemented as a transistor and the light may be a light-emitting diode (LED). An electronic circuit is powered by a power supply of a certain voltage, perhaps 5 volts. One side of the power supply is connected to ground, as shown in Figure 2.2*b*. The ground connection may also be used as the return path for the current, to close the loop, which is achieved by connecting one side of the light to ground as indicated in the figure. Of course, the light can also be connected by a wire directly to the grounded side of the power supply, as in Figure 2.2*a*.

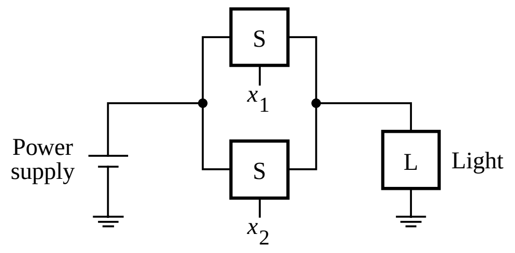
Consider now the possibility of using two switches to control the state of the light. Let x_1 and x_2 be the control inputs for these switches. The switches can be connected either in series or in parallel as shown in Figure 2.3. Using a series connection, the light will be turned on only if both switches are closed. If either switch is open, the light will be off. This behavior can be described by the expression

$$\begin{aligned} & L(x_1, x_2) = x_1 \cdot x_2 \\ \text{where} \quad & L = 1 \text{ if } x_1 = 1 \text{ and } x_2 = 1, \\ & L = 0 \text{ otherwise.} \end{aligned}$$

The "·" symbol is called the *AND operator*, and the circuit in Figure 2.3*a* is said to implement a *logical AND function*.



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 2.3 Two basic functions.

The parallel connection of two switches is given in Figure 2.3*b*. In this case the light will be on if either x_1 or x_2 switch is closed. The light will also be on if both switches are closed. The light will be off only if both switches are open. This behavior can be stated as

$$\begin{aligned} & L(x_1, x_2) = x_1 + x_2 \\ \text{where} \quad & L = 1 \text{ if } x_1 = 1 \text{ or } x_2 = 1 \text{ or if } x_1 = x_2 = 1, \\ & L = 0 \text{ if } x_1 = x_2 = 0. \end{aligned}$$

The + symbol is called the *OR operator*, and the circuit in Figure 2.3*b* is said to implement a *logical OR function*.

In the above expressions for AND and OR, the output $L(x_1, x_2)$ is a logic function with input variables x_1 and x_2. The AND and OR functions are two of the most important logic functions. Together with some other simple functions, they can be used as building blocks

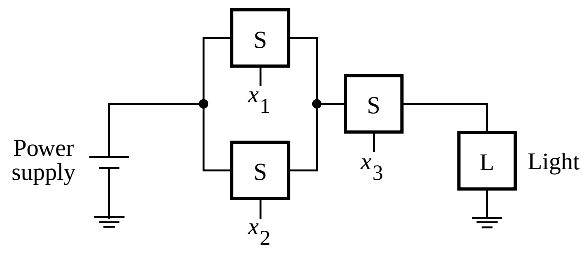


Figure 2.4 A series-parallel connection.

for the implementation of all logic circuits. Figure 2.4 illustrates how three switches can be used to control the light in a more complex way. This series-parallel connection of switches realizes the logic function

$$L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$$

The light is on if $x_3 = 1$ and, at the same time, at least one of the x_1 or x_2 inputs is equal to 1.

2.2 INVERSION

So far we have assumed that some positive action takes place when a switch is closed, such as turning the light on. It is equally interesting and useful to consider the possibility that a positive action takes place when a switch is opened. Suppose that we connect the light as shown in Figure 2.5. In this case the switch is connected in parallel with the light, rather than in series. Consequently, a closed switch will short-circuit the light and prevent the current from flowing through it. Note that we have included an extra resistor in this circuit to ensure that the closed switch does not short-circuit the power supply. The light will be turned on when the switch is opened. Formally, we express this functional behavior as

$$\begin{aligned} & L(x) = \overline{x} \\ \text{where} \quad & L = 1 \text{ if } x = 0, \\ & L = 0 \text{ if } x = 1 \end{aligned}$$

The value of this function is the inverse of the value of the input variable. Instead of using the word *inverse*, it is more common to use the term *complement*. Thus we say that $L(x)$ is a complement of x in this example. Another frequently used term for the same operation is the *NOT operation*. There are several commonly used notations for indicating the complementation. In the preceding expression we placed an overbar on top of x. This notation is probably the best from the visual point of view. However, when complements are needed in expressions that are typed using a computer keyboard, which is often done when using CAD tools, it is impractical to use overbars. Instead, either an apostrophe is

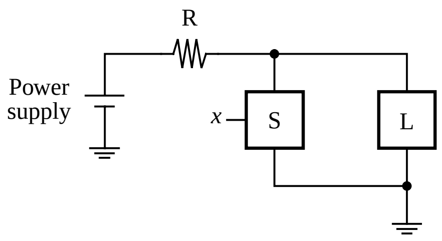


Figure 2.5 An inverting circuit.

placed after the variable, or the exclamation mark or the word NOT is placed in front of the variable to denote the complementation. Thus the following are equivalent:

$$\overline{x} = x' = !x = \text{NOT } x$$

The complement operation can be applied to a single variable or to more complex operations. For example, if

$$f(x_1, x_2) = x_1 + x_2$$

then the complement of f is

$$\overline{f}(x_1, x_2) = \overline{x_1 + x_2}$$

This expression yields the logic value 1 only when neither x_1 nor x_2 is equal to 1, that is, when $x_1 = x_2 = 0$. Again, the following notations are equivalent:

$$\overline{x_1 + x_2} = (x_1 + x_2)' = !(x_1 + x_2) = \text{NOT } (x_1 + x_2)$$

2.3 Truth Tables

We have introduced the three most basic logic operations—AND, OR, and complement—by relating them to simple circuits built with switches. This approach gives these operations a certain "physical meaning." The same operations can also be defined in the form of a table, called a *truth table*, as shown in Figure 2.6. The first two columns (to the left of the heavy vertical line) give all four possible combinations of logic values that the variables x_1 and x_2 can have. The next column defines the AND operation for each combination of values of x_1 and x_2, and the last column defines the OR operation. Because we will frequently need to refer to "combinations of logic values" applied to some variables, we will adopt a shorter term, *valuation*, to denote such a combination of logic values.

The truth table is a useful aid for depicting information involving logic functions. We will use it in this book to define specific functions and to show the validity of certain functional relations. Small truth tables are easy to deal with. However, they grow exponentially in size with the number of variables. A truth table for three input variables has eight rows because there are eight possible valuations of these variables. Such a table is given in Figure 2.7, which defines three-input AND and OR functions. For four-input variables the truth table has 16 rows, and so on.

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

Figure 2.6 A truth table for the AND and OR operations.

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 2.7 Three-input AND and OR operations.

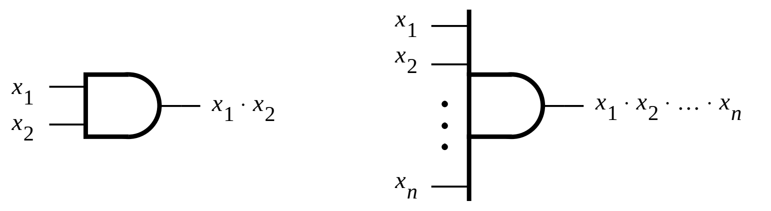
The AND and OR operations can be extended to n variables. An AND function of variables $x_1, x_2, \cdots, x_n$ has the value 1 only if all n variables are equal to 1. An OR function of variables $x_1, x_2, \cdots, x_n$ has the value 1 if at least one, or more, of the variables is equal to 1.

2.4 LOGIC GATES AND NETWORKS

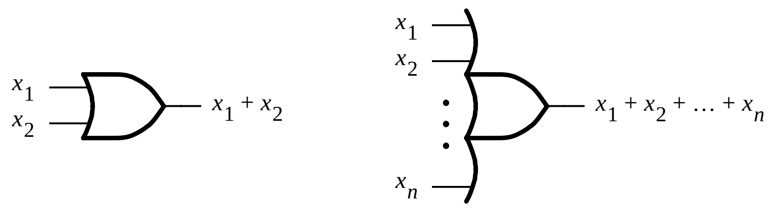
The three basic logic operations introduced in the previous sections can be used to implement logic functions of any complexity. A complex function may require many of these basic operations for its implementation. Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a *logic gate*. A logic gate has one or more inputs and one output that is a function of its inputs. It is often convenient to describe a logic circuit by drawing a circuit diagram, or *schematic*, consisting of graphical symbols representing the logic gates. The graphical symbols for the AND, OR, and NOT gates are shown in Figure 2.8. The figure indicates on the left side how the AND and OR gates are drawn when there are only a few inputs. On the right side it shows how the symbols are augmented to accommodate a greater number of inputs. We will show how logic gates are built using transistors in Chapter 3.

A larger circuit is implemented by a *network* of gates. For example, the logic function from Figure 2.4 can be implemented by the network in Figure 2.9. The complexity of a given network has a direct impact on its cost. Because it is always desirable to reduce the cost of any manufactured product, it is important to find ways for implementing logic circuits as inexpensively as possible. We will see shortly that a given logic function can be implemented with a number of different networks. Some of these networks are simpler than others, hence searching for the solutions that entail minimum cost is prudent.

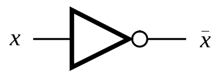
In technical jargon a network of gates is often called a *logic network* or simply a *logic circuit*. We will use these terms interchangeably.



(a) AND gates



(b) OR gates



(c) NOT gate

Figure 2.8 The basic gates.

2.4.1 Analysis of a Logic Network

A designer of digital systems is faced with two basic issues. For an existing logic network, it must be possible to determine the function performed by the network. This task is referred to as the *analysis* process. The reverse task of designing a new network that implements a desired functional behavior is referred to as the *synthesis* process. The analysis process is rather straightforward and much simpler than the synthesis process.

Figure 2.10*a* shows a simple network consisting of three gates. To determine its functional behavior, we can consider what happens if we apply all possible input signals to it. Suppose that we start by making $x_1 = x_2 = 0$. This forces the output of the NOT gate

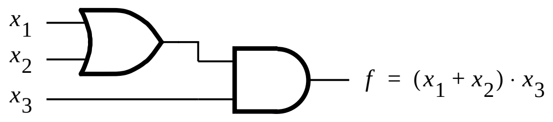
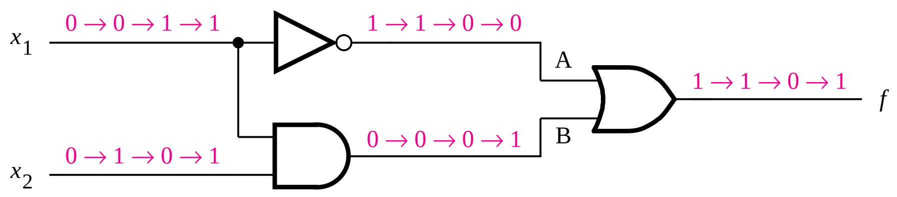


Figure 2.9 The function from Figure 2.4.

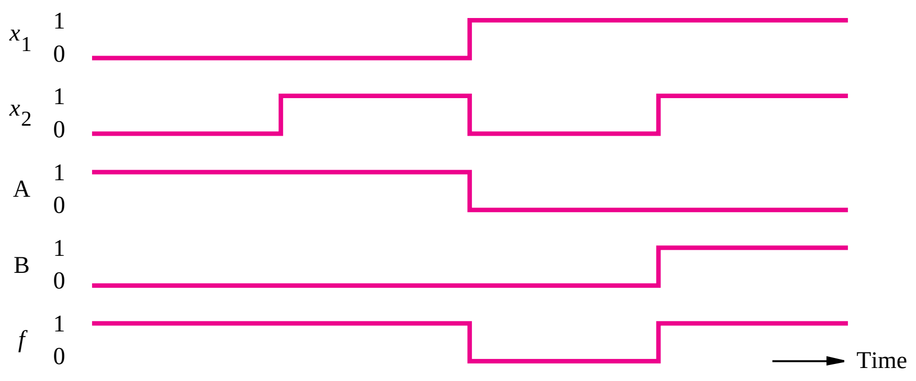


x_1
$0 \rightarrow 0 \rightarrow 1 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 0$
A
$1 \rightarrow 1 \rightarrow 0 \rightarrow 1$
f
$0 \rightarrow 0 \rightarrow 0 \rightarrow 1$
B
x_2
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$

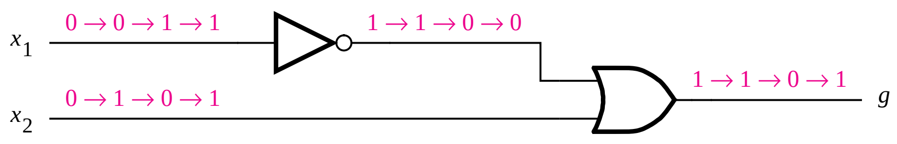
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

(b) Truth table for f



(c) Timing diagram



x_1
$0 \rightarrow 0 \rightarrow 1 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 0$
x_2
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 1$
g

(d) Network that implements $g = \bar{x}_1 + x_2$

Figure 2.10 An example of logic networks.

to be equal to 1 and the output of the AND gate to be 0. Because one of the inputs to the OR gate is 1, the output of this gate will be 1. Therefore, $f = 1$ if $x_1 = x_2 = 0$. If we let $x_1 = 0$ and $x_2 = 1$, then no change in the value of f will take place, because the outputs of the NOT and AND gates will still be 1 and 0, respectively. Next, if we apply $x_1 = 1$ and $x_2 = 0$, then the output of the NOT gate changes to 0 while the output of the AND gate remains at 0. Both inputs to the OR gate are then equal to 0; hence the value of f will be 0. Finally, let $x_1 = x_2 = 1$. Then the output of the AND gate goes to 1, which in turn causes f to be equal to 1. Our verbal explanation can be summarized in the form of the truth table shown in Figure 2.10*b*.

Timing Diagram

We have determined the behavior of the network in Figure 2.10*a* by considering the four possible valuations of the inputs x_1 and x_2. Suppose that the signals that correspond to these valuations are applied to the network in the order of our discussion; that is, $(x_1, x_2) = (0, 0)$ followed by $(0, 1)$, $(1, 0)$, and $(1, 1)$. Then changes in the signals at various points in the network would be as indicated in blue in the figure. The same information can be presented in graphical form, known as a *timing diagram*, as shown in Figure 2.10*c*. The time runs from left to right, and each input valuation is held for some fixed period. The figure shows the waveforms for the inputs and output of the network, as well as for the internal signals at the points labeled *A* and *B*.

Timing diagrams are used for many purposes. They depict the behavior of a logic circuit in a form that can be observed when the circuit is tested using instruments such as logic analyzers and oscilloscopes. Also, they are often generated by CAD tools to show the designer how a given circuit is expected to behave before it is actually implemented electronically. We will introduce the CAD tools later in this chapter and will make use of them throughout the book.

Functionally Equivalent Networks

Now consider the network in Figure 2.10*d*. Going through the same analysis procedure, we find that the output g changes in exactly the same way as f does in part (*a*) of the figure. Therefore, $g(x_1, x_2) = f(x_1, x_2)$, which indicates that the two networks are functionally equivalent; the output behavior of both networks is represented by the truth table in Figure 2.10*b*. Since both networks realize the same function, it makes sense to use the simpler one, which is less costly to implement.

In general, a logic function can be implemented with a variety of different networks, probably having different costs. This raises an important question: How does one find the best implementation for a given function? Many techniques exist for synthesizing logic functions. We will discuss the main approaches in Chapter 4. For now, we should note that some manipulation is needed to transform the more complex network in Figure 2.10*a* into the network in Figure 2.10*d*. Since $f(x_1, x_2) = \overline{x}_1 + x_1 \cdot x_2$ and $g(x_1, x_2) = \overline{x}_1 + x_2$, there must exist some rules that can be used to show the equivalence

$$\overline{x}_1 + x_1 \cdot x_2 = \overline{x}_1 + x_2$$

We have already established this equivalence through detailed analysis of the two circuits and construction of the truth table. But the same outcome can be achieved through algebraic manipulation of logic expressions. In the next section we will discuss a mathematical approach for dealing with logic functions, which provides the basis for modern design techniques.

2.5 Boolean Algebra

In 1849 George Boole published a scheme for the algebraic description of processes involved in logical thought and reasoning [1]. Subsequently, this scheme and its further refinements became known as *Boolean algebra*. It was almost 100 years later that this algebra found application in the engineering sense. In the late 1930s Claude Shannon showed that Boolean algebra provides an effective means of describing circuits built with switches [2]. The algebra can, therefore, be used to describe logic circuits. We will show that this algebra is a powerful tool that can be used for designing and analyzing logic circuits. The reader will come to appreciate that it provides the foundation for much of our modern digital technology.

Axioms of Boolean Algebra

Like any algebra, Boolean algebra is based on a set of rules that are derived from a small number of basic assumptions. These assumptions are called *axioms*. Let us assume that Boolean algebra B involves elements that take on one of two values, 0 and 1. Assume that the following axioms are true:

1*a*. $0 \cdot 0 = 0$

1*b*. $1 + 1 = 1$

2*a*. $1 \cdot 1 = 1$

2*b*. $0 + 0 = 0$

3*a*. $0 \cdot 1 = 1 \cdot 0 = 0$

3*b*. $1 + 0 = 0 + 1 = 1$

4*a*. If $x = 0$, then $\overline{x} = 1$

4*b*. If $x = 1$, then $\overline{x} = 0$

Single-Variable Theorems

From the axioms we can define some rules for dealing with single variables. These rules are often called *theorems*. If x is a variable in B, then the following theorems hold:

5*a*. $x \cdot 0 = 0$

5*b*. $x + 1 = 1$

6*a*. $x \cdot 1 = x$

6*b*. $x + 0 = x$

7*a*. $x \cdot x = x$

7*b*. $x + x = x$

8*a*. $x \cdot \overline{x} = 0$

8*b*. $x + \overline{x} = 1$

9. $\overline{\overline{x}} = x$

It is easy to prove the validity of these theorems by perfect induction, that is, by substituting the values $x = 0$ and $x = 1$ into the expressions and using the axioms given above. For example, in theorem 5*a*, if $x = 0$, then the theorem states that $0 \cdot 0 = 0$, which is true according to axiom 1*a*. Similarly, if $x = 1$, then theorem 5*a* states that $1 \cdot 0 = 0$, which is also true according to axiom 3*a*. The reader should verify that theorems 5*a* to 9 can be proven in this way.

Duality

Notice that we have listed the axioms and the single-variable theorems in pairs. This is done to reflect the important *principle of duality*. Given a logic expression, its *dual* is obtained by replacing all $+$ operators with $\cdot$ operators, and vice versa, and by replacing all 0s with 1s, and vice versa. The dual of any true statement (axiom or theorem) in Boolean algebra is also a true statement. At this point in the discussion, the reader will not appreciate why duality is a useful concept. However, this concept will become clear later in the chapter, when we will show that duality implies that at least two different ways exist to express every logic function with Boolean algebra. Often, one expression leads to a simpler physical implementation than the other and is thus preferable.

Two- and Three-Variable Properties

To enable us to deal with a number of variables, it is useful to define some two- and three-variable algebraic identities. For each identity, its dual version is also given. These identities are often referred to as *properties*. They are known by the names indicated below. If x, y, and z are the variables in B, then the following properties hold:

10*a*.	$x \cdot y = y \cdot x$	*Commutative*
10*b*.	$x + y = y + x$	
11*a*.	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	*Associative*
11*b*.	$x + (y + z) = (x + y) + z$	
12*a*.	$x \cdot (y + z) = x \cdot y + x \cdot z$	*Distributive*
12*b*.	$x + y \cdot z = (x + y) \cdot (x + z)$	
13*a*.	$x + x \cdot y = x$	*Absorption*
13*b*.	$x \cdot (x + y) = x$	
14*a*.	$x \cdot y + x \cdot \overline{y} = x$	*Combining*
14*b*.	$(x + y) \cdot (x + \overline{y}) = x$	
15*a*.	$\overline{x \cdot y} = \overline{x} + \overline{y}$	*DeMorgan's theorem*
15*b*.	$\overline{x + y} = \overline{x} \cdot \overline{y}$	
16*a*.	$x + \overline{x} \cdot y = x + y$	
16*b*.	$x \cdot (\overline{x} + y) = x \cdot y$	

Again, we can prove the validity of these properties either by perfect induction or by performing algebraic manipulation. Figure 2.11 illustrates how perfect induction can be used to prove DeMorgan's theorem, using the format of a truth table. The evaluation of left-hand and right-hand sides of the identity in 15*a* gives the same result.

We have listed a number of axioms, theorems, and properties. Not all of these are necessary to define Boolean algebra. For example, assuming that the $+$ and $\cdot$ operations are defined, it is sufficient to include theorems 5 and 8 and properties 10 and 12. These are sometimes referred to as Huntington's basic postulates [3]. The other identities can be derived from these postulates.

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\overline{x}$	$\overline{y}$	$\overline{x} + \overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0
		LHS		RHS		

Figure 2.11 Proof of DeMorgan's theorem in 15*a*.

The preceding axioms, theorems, and properties provide the information necessary for performing algebraic manipulation of more complex expressions.

Example 2.1

Let us prove the validity of the logic equation

$$(x_1 + x_3) \cdot (\overline{x}_1 + \overline{x}_3) = x_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_3$$

The left-hand side can be manipulated as follows. Using the distributive property, 12*a*, gives

$$\text{LHS} = (x_1 + x_3) \cdot \overline{x}_1 + (x_1 + x_3) \cdot \overline{x}_3$$

Applying the distributive property again yields

$$\text{LHS} = x_1 \cdot \overline{x}_1 + x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3 + x_3 \cdot \overline{x}_3$$

Note that the distributive property allows ANDing the terms in parenthesis in a way analogous to multiplication in ordinary algebra. Next, according to theorem 8*a*, the terms $x_1 \cdot \overline{x}_1$ and $x_3 \cdot \overline{x}_3$ are both equal to 0. Therefore,

$$\text{LHS} = 0 + x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3 + 0$$

From 6*b* it follows that

$$\text{LHS} = x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3$$

Finally, using the commutative property, 10*a* and 10*b*, this becomes

$$\text{LHS} = x_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_3$$

which is the same as the right-hand side of the initial equation.

Example 2.2

Consider the logic equation

$$x_1 \cdot \overline{x}_3 + \overline{x}_2 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot x_3 = \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot x_2 + x_1 \cdot \overline{x}_2$$

The left-hand side can be manipulated as follows

$$\begin{aligned} \text{LHS} &= x_1 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_2 \cdot x_3 && \text{using } 10b \\ &= x_1 \cdot (\overline{x}_3 + x_3) + \overline{x}_2 \cdot (\overline{x}_3 + x_3) && \text{using } 12a \end{aligned}$$

$$\begin{aligned} &= x_1 \cdot 1 + \overline{x}_2 \cdot 1 && \text{using } 8b \\ &= x_1 + \overline{x}_2 && \text{using } 6a \end{aligned}$$

The right-hand side can be manipulated as

$$\begin{aligned} \text{RHS} &= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot (x_2 + \overline{x}_2) && \text{using } 12a \\ &= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot 1 && \text{using } 8b \\ &= \overline{x}_1 \cdot \overline{x}_2 + x_1 && \text{using } 6a \\ &= x_1 + \overline{x}_1 \cdot \overline{x}_2 && \text{using } 10b \\ &= x_1 + \overline{x}_2 && \text{using } 16a \end{aligned}$$

Being able to manipulate both sides of the initial equation into identical expressions establishes the validity of the equation. Note that the same logic function is represented by either the left- or the right-hand side of the above equation; namely

$$\begin{aligned} f(x_1, x_2, x_3) &= x_1 \cdot \overline{x}_3 + \overline{x}_2 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot x_3 \\ &= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot x_2 + x_1 \cdot \overline{x}_2 \end{aligned}$$

As a result of manipulation, we have found a much simpler expression

$$f(x_1, x_2, x_3) = x_1 + \overline{x}_2$$

which also represents the same function. This simpler expression would result in a lower-cost logic circuit that could be used to implement the function.

Examples 2.1 and 2.2 illustrate the purpose of the axioms, theorems, and properties as a mechanism for algebraic manipulation. Even these simple examples suggest that it is impractical to deal with highly complex expressions in this way. However, these theorems and properties provide the basis for automating the synthesis of logic functions in CAD tools. To understand what can be achieved using these tools, the designer needs to be aware of the fundamental concepts.

2.5.1 THE VENN DIAGRAM

We have suggested that perfect induction can be used to verify the theorems and properties. This procedure is quite tedious and not very informative from the conceptual point of view. A simple visual aid that can be used for this purpose also exists. It is called the Venn diagram, and the reader is likely to find that it provides for a more intuitive understanding of how two expressions may be equivalent.

The Venn diagram has traditionally been used in mathematics to provide a graphical illustration of various operations and relations in the algebra of sets. A set s is a collection of elements that are said to be the members of s. In the Venn diagram the elements of a set are represented by the area enclosed by a contour such as a square, a circle, or an ellipse. For example, in a universe N of integers from 1 to 10, the set of even numbers is $E = \{2, 4, 6, 8, 10\}$. A contour representing E encloses the even numbers. The odd numbers form the complement of E; hence the area outside the contour represents $\overline{E} = \{1, 3, 5, 7, 9\}$.

Since in Boolean algebra there are only two values (elements) in the universe, $B = \{0, 1\}$, we will say that the area within a contour corresponding to a set s denotes that $s = 1$, while the area outside the contour denotes $s = 0$. In the diagram we will shade the area where $s = 1$. The concept of the Venn diagram is illustrated in Figure 2.12. The universe B is represented by a square. Then the constants 1 and 0 are represented as shown in parts (*a*) and (*b*) of the figure. A variable, say, x, is represented by a circle, such that the area inside the circle corresponds to $x = 1$, while the area outside the circle corresponds to $x = 0$. This is illustrated in part (*c*). An expression involving one or more variables is depicted by

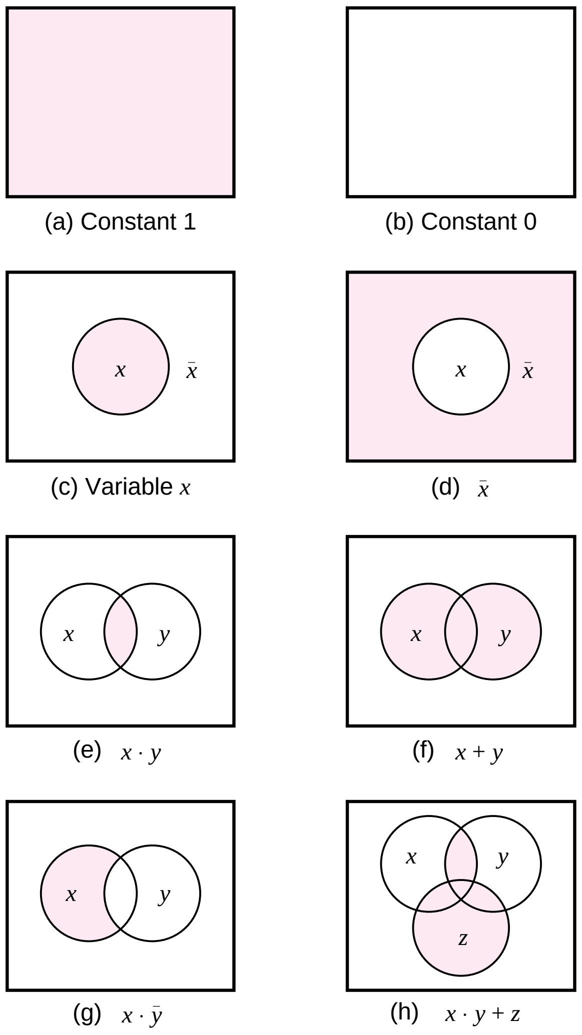


Figure 2.12 The Venn diagram representation.

shading the area where the value of the expression is equal to 1. Part (*d*) indicates how the complement of x is represented.

To represent two variables, x and y, we draw two overlapping circles. Then the area where the circles overlap represents the case where $x = y = 1$, namely, the AND of x and y, as shown in part (*e*). Since this common area consists of the intersecting portions of x and y, the AND operation is often referred to formally as the *intersection* of x and y. Part (*f*) illustrates the OR operation, where $x + y$ represents the total area within both circles, namely, where at least one of x or y is equal to 1. Since this combines the areas in the circles, the OR operation is formally often called the *union* of x and y.

Part (*g*) depicts the product term $x \cdot \overline{y}$, which is represented by the intersection of the area for x with that for $\overline{y}$. Part (*h*) gives a three-variable example; the expression $x \cdot y + z$ is the union of the area for z with that of the intersection of x and y.

To see how we can use Venn diagrams to verify the equivalence of two expressions, let us demonstrate the validity of the distributive property, 12*a*, in section 2.5. Figure 2.13 gives the construction of the left and right sides of the identity that defines the property

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

Part (*a*) shows the area where $x = 1$. Part (*b*) indicates the area for $y + z$. Part (*c*) gives the diagram for $x \cdot (y + z)$, the intersection of shaded areas in parts (*a*) and (*b*). The right-hand

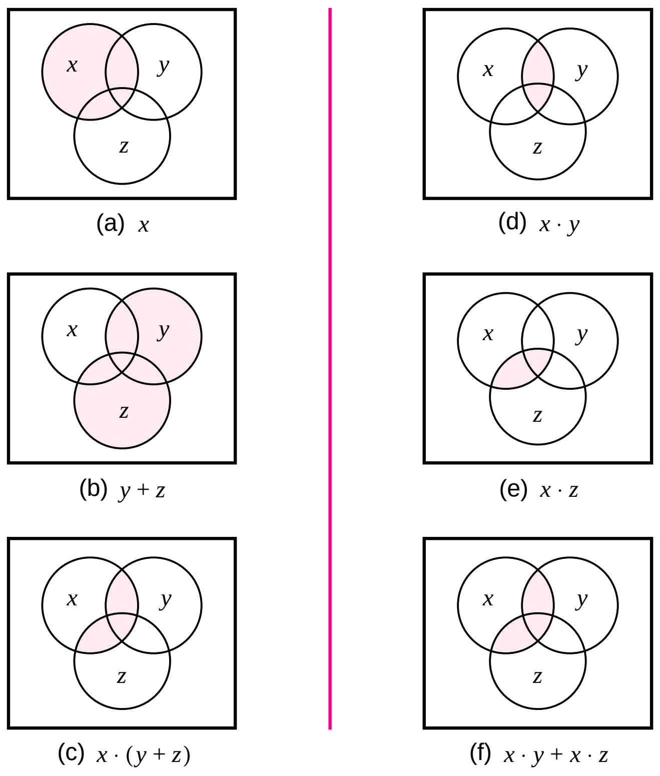


Figure 2.13 Verification of the distributive property $x \cdot (y + z) = x \cdot y + x \cdot z$.

side is constructed in parts (*d*), (*e*), and (*f*). Parts (*d*) and (*e*) describe the terms $x \cdot y$ and $x \cdot z$, respectively. The union of the shaded areas in these two diagrams then corresponds to the expression $x \cdot y + x \cdot z$, as seen in part (*f*). Since the shaded areas in parts (*c*) and (*f*) are identical, it follows that the distributive property is valid.

As another example, consider the identity

$$x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$$

which is illustrated in Figure 2.14. Notice that this identity states that the term $y \cdot z$ is fully covered by the terms $x \cdot y$ and $\overline{x} \cdot z$; therefore, this term can be omitted.

The reader should use the Venn diagram to prove some other identities. It is particularly instructive to prove the validity of DeMorgan's theorem in this way.

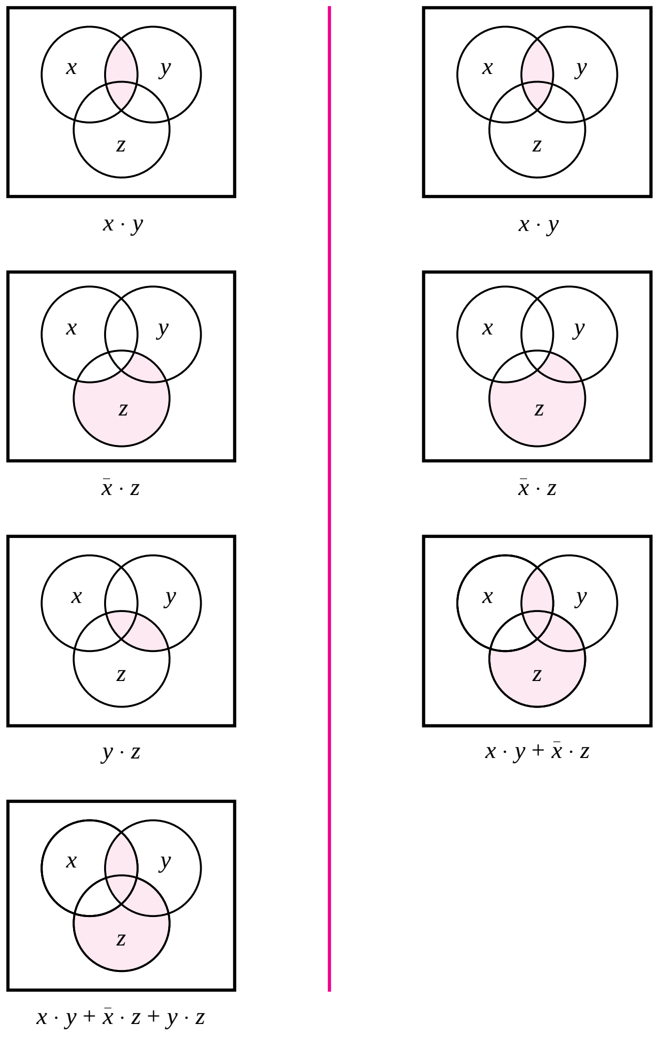


Figure 2.14 Verification of $x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$.

2.5.2 NOTATION AND TERMINOLOGY

Boolean algebra is based on the AND and OR operations. We have adopted the symbols $\cdot$ and $+$ to denote these operations. These are also the standard symbols for the familiar arithmetic multiplication and addition operations. Considerable similarity exists between the Boolean operations and the arithmetic operations, which is the main reason why the same symbols are used. In fact, when single digits are involved there is only one significant difference; the result of $1 + 1$ is equal to 2 in ordinary arithmetic, whereas it is equal to 1 in Boolean algebra as defined by theorem 7*b* in section 2.5.

When dealing with digital circuits, most of the time the $+$ symbol obviously represents the OR operation. However, when the task involves the design of logic circuits that perform arithmetic operations, some confusion may develop about the use of the $+$ symbol. To avoid such confusion, an alternative set of symbols exists for the AND and OR operations. It is quite common to use the $\wedge$ symbol to denote the AND operation, and the $\vee$ symbol for the OR operation. Thus, instead of $x_1 \cdot x_2$, we can write $x_1 \wedge x_2$, and instead of $x_1 + x_2$, we can write $x_1 \vee x_2$.

Because of the similarity with the arithmetic addition and multiplication operations, the OR and AND operations are often called the *logical sum* and *product* operations. Thus $x_1 + x_2$ is the logical sum of x_1 and x_2, and $x_1 \cdot x_2$ is the logical product of x_1 and x_2. Instead of saying "logical product" and "logical sum," it is customary to say simply "product" and "sum." Thus we say that the expression

$$x_1 \cdot \overline{x}_2 \cdot x_3 + \overline{x}_1 \cdot x_4 + x_2 \cdot x_3 \cdot \overline{x}_4$$

is a sum of three product terms, whereas the expression

$$(\overline{x}_1 + x_3) \cdot (x_1 + \overline{x}_3) \cdot (\overline{x}_2 + x_3 + x_4)$$

is a product of three sum terms.

2.5.3 PRECEDENCE OF OPERATIONS

Using the three basic operations—AND, OR, and NOT—it is possible to construct an infinite number of logic expressions. Parentheses can be used to indicate the order in which the operations should be performed. However, to avoid an excessive use of parentheses, another convention defines the precedence of the basic operations. It states that in the absence of parentheses, operations in a logic expression must be performed in the order: NOT, AND, and then OR. Thus in the expression

$$x_1 \cdot x_2 + \overline{x}_1 \cdot \overline{x}_2$$

it is first necessary to generate the complements of x_1 and x_2. Then the product terms $x_1 \cdot x_2$ and $\overline{x}_1 \cdot \overline{x}_2$ are formed, followed by the sum of the two product terms. Observe that in the absence of this convention, we would have to use parentheses to achieve the same effect as follows:

$$(x_1 \cdot x_2) + ((\overline{x}_1) \cdot (\overline{x}_2))$$

Finally, to simplify the appearance of logic expressions, it is customary to omit the $\cdot$ operator when there is no ambiguity. Therefore, the preceding expression can be written as

$$x_1x_2 + \overline{x}_1\overline{x}_2$$

We will use this style throughout the book.

2.6 Synthesis Using AND, OR, and NOT Gates

Armed with some basic ideas, we can now try to implement arbitrary functions using the AND, OR, and NOT gates. Suppose that we wish to design a logic circuit with two inputs, x_1 and x_2. Assume that x_1 and x_2 represent the states of two switches, either of which may be open (0) or closed (1). The function of the circuit is to continuously monitor the state of the switches and to produce an output logic value 1 whenever the switches (x_1, x_2) are in states $(0, 0)$, $(0, 1)$, or $(1, 1)$. If the state of the switches is $(1, 0)$, the output should be 0. Another way of stating the required functional behavior of this circuit is that the output must be equal to 0 if the switch x_1 is closed and x_2 is open; otherwise, the output must be 1. We can express the required behavior using a truth table, as shown in Figure 2.15.

A possible procedure for designing a logic circuit that implements the truth table is to create a product term that has a value of 1 for each valuation for which the output function f has to be 1. Then we can take a logical sum of these product terms to realize f. Let us begin with the fourth row of the truth table, which corresponds to $x_1 = x_2 = 1$. The product term that is equal to 1 for this valuation is $x_1 \cdot x_2$, which is just the AND of x_1 and x_2. Next consider the first row of the table, for which $x_1 = x_2 = 0$. For this valuation the value 1 is produced by the product term $\overline{x}_1 \cdot \overline{x}_2$. Similarly, the second row leads to the term $\overline{x}_1 \cdot x_2$. Thus f may be realized as

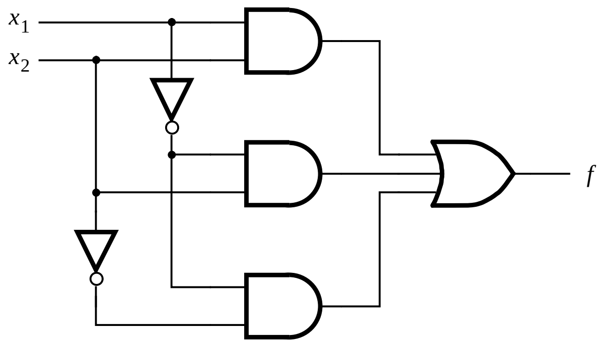
$$f(x_1, x_2) = x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2$$

The logic network that corresponds to this expression is shown in Figure 2.16*a*.

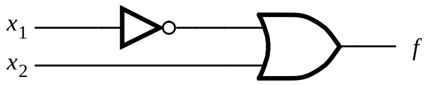
Although this network implements f correctly, it is not the simplest such network. To find a simpler network, we can manipulate the obtained expression using the theorems and

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

Figure 2.15 A function to be synthesized.



(a) Canonical sum-of-products



(b) Minimal-cost realization

Figure 2.16 Two implementations of the function in Figure 2.15.

properties from section 2.5. According to theorem 7*b*, we can replicate any term in a logical sum expression. Replicating the third product term, the above expression becomes

$$f(x_1, x_2) = x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + \overline{x}_1x_2$$

Using the commutative property 10*b* to interchange the second and third product terms gives

$$f(x_1, x_2) = x_1x_2 + \overline{x}_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2$$

Now the distributive property 12*a* allows us to write

$$f(x_1, x_2) = (x_1 + \overline{x}_1)x_2 + \overline{x}_1(\overline{x}_2 + x_2)$$

Applying theorem 8*b* we get

$$f(x_1, x_2) = 1 \cdot x_2 + \overline{x}_1 \cdot 1$$

Finally, theorem 6*a* leads to

$$f(x_1, x_2) = x_2 + \overline{x}_1$$

The network described by this expression is given in Figure 2.16*b*. Obviously, the cost of this network is much less than the cost of the network in part (*a*) of the figure.

This simple example illustrates two things. First, a straightforward implementation of a function can be obtained by using a product term (AND gate) for each row of the truth table for which the function is equal to 1. Each product term contains all input variables, and it is formed such that if the input variable x_i is equal to 1 in the given row, then x_i is entered in the term; if $x_i = 0$, then $\overline{x}_i$ is entered. The sum of these product terms realizes the desired function. Second, there are many different networks that can realize a given function. Some of these networks may be simpler than others. Algebraic manipulation can be used to derive simplified logic expressions and thus lower-cost networks.

The process whereby we begin with a description of the desired functional behavior and then generate a circuit that realizes this behavior is called *synthesis*. Thus we can say that we "synthesized" the networks in Figure 2.16 from the truth table in Figure 2.15. Generation of AND-OR expressions from a truth table is just one of many types of synthesis techniques that we will encounter in this book.

2.6.1 Sum-of-Products and Product-of-Sums Forms

Having introduced the synthesis process by means of a very simple example, we will now present it in more formal terms using the terminology that is encountered in the technical literature. We will also show how the principle of duality, which was introduced in section 2.5, applies broadly in the synthesis process.

If a function f is specified in the form of a truth table, then an expression that realizes f can be obtained by considering either the rows in the table for which $f = 1$, as we have already done, or by considering the rows for which $f = 0$, as we will explain shortly.

Minterms

For a function of n variables, a product term in which each of the n variables appears once is called a *minterm*. The variables may appear in a minterm either in uncomplemented or complemented form. For a given row of the truth table, the minterm is formed by including x_i if $x_i = 1$ and by including $\overline{x}_i$ if $x_i = 0$.

To illustrate this concept, consider the truth table in Figure 2.17. We have numbered the rows of the table from 0 to 7, so that we can refer to them easily. (The reader who is already familiar with the binary number representation will realize that the row numbers chosen are just the numbers represented by the bit patterns of variables x_1, x_2, and x_3; we will discuss number representation in Chapter 5.) The figure shows all minterms for the three-variable table. For example, in the first row the variables have the values $x_1 = x_2 = x_3 = 0$, which leads to the minterm $\overline{x}_1\overline{x}_2\overline{x}_3$. In the second row $x_1 = x_2 = 0$ and $x_3 = 1$, which gives the minterm $\overline{x}_1\overline{x}_2x_3$, and so on. To be able to refer to the individual minterms easily, it is convenient to identify each minterm by an index that corresponds to the row numbers shown in the figure. We will use the notation m_i to denote the minterm for row number i. Thus $m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$, $m_1 = \overline{x}_1\overline{x}_2x_3$, and so on.

Sum-of-Products Form

A function f can be represented by an expression that is a sum of minterms, where each minterm is ANDed with the value of f for the corresponding valuation of input variables. For example, the two-variable minterms are $m_0 = \overline{x}_1\overline{x}_2$, $m_1 = \overline{x}_1x_2$, $m_2 = x_1\overline{x}_2$, and $m_3 = x_1x_2$. The function in Figure 2.15 can be represented as

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x}_1\overline{x}_2 x_3$	$M_1 = x_1 + x_2 + \overline{x}_3$
2	0	1	0	$m_2 = \overline{x}_1 x_2\overline{x}_3$	$M_2 = x_1 + \overline{x}_2 + x_3$
3	0	1	1	$m_3 = \overline{x}_1 x_2 x_3$	$M_3 = x_1 + \overline{x}_2 + \overline{x}_3$
4	1	0	0	$m_4 = x_1\overline{x}_2\overline{x}_3$	$M_4 = \overline{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\overline{x}_2 x_3$	$M_5 = \overline{x}_1 + x_2 + \overline{x}_3$
6	1	1	0	$m_6 = x_1 x_2\overline{x}_3$	$M_6 = \overline{x}_1 + \overline{x}_2 + x_3$
7	1	1	1	$m_7 = x_1 x_2 x_3$	$M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$

Figure 2.17 Three-variable minterms and maxterms.

$$\begin{aligned} f &= m_0 \cdot 1 + m_1 \cdot 1 + m_2 \cdot 0 + m_3 \cdot 1 \\ &= m_0 + m_1 + m_3 \\ &= \overline{x}_1\overline{x}_2 + \overline{x}_1 x_2 + x_1 x_2 \end{aligned}$$

which is the form that we derived in the previous section using an intuitive approach. Only the minterms that correspond to the rows for which $f = 1$ appear in the resulting expression.

Any function f can be represented by a sum of minterms that correspond to the rows in the truth table for which $f = 1$. The resulting implementation is functionally correct and unique, but it is not necessarily the lowest-cost implementation of f. A logic expression consisting of product (AND) terms that are summed (ORed) is said to be of the *sum-of-products* form. If each product term is a minterm, then the expression is called a *canonical sum-of-products* for the function f. As we have seen in the example of Figure 2.16, the first step in the synthesis process is to derive a canonical sum-of-products expression for the given function. Then we can manipulate this expression, using the theorems and properties of section 2.5, with the goal of finding a functionally equivalent sum-of-products expression that has a lower cost.

As another example, consider the three-variable function $f(x_1, x_2, x_3)$, specified by the truth table in Figure 2.18. To synthesize this function, we have to include the minterms m_1, m_4, m_5, and m_6. Copying these minterms from Figure 2.17 leads to the following canonical sum-of-products expression for f

$$f(x_1, x_2, x_3) = \overline{x}_1\overline{x}_2 x_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2 x_3 + x_1 x_2\overline{x}_3$$

This expression can be manipulated as follows

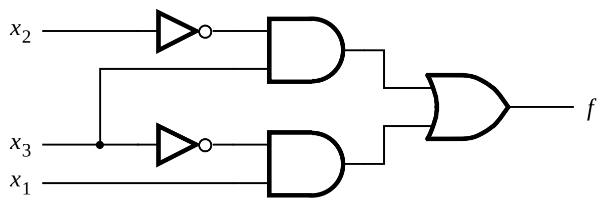
$$\begin{aligned} f &= (\overline{x}_1 + x_1)\overline{x}_2 x_3 + x_1(\overline{x}_2 + x_2)\overline{x}_3 \\ &= 1 \cdot \overline{x}_2 x_3 + x_1 \cdot 1 \cdot \overline{x}_3 \\ &= \overline{x}_2 x_3 + x_1\overline{x}_3 \end{aligned}$$

This is the minimum-cost sum-of-products expression for f. It describes the circuit shown in Figure 2.19*a*. A good indication of the *cost* of a logic circuit is the total number of gates

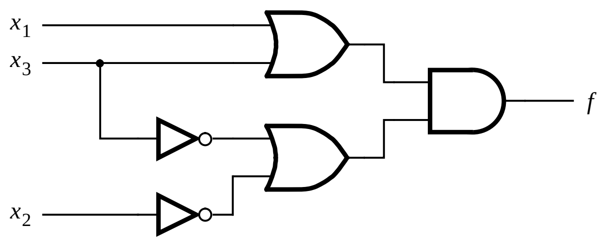
Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 2.18 A three-variable function.

plus the total number of inputs to all gates in the circuit. Using this measure, the cost of the network in Figure 2.19*a* is 13, because there are five gates and eight inputs to the gates. By comparison, the network implemented on the basis of the canonical sum-of-products would have a cost of 27; from the preceding expression, the OR gate has four inputs, each of the four AND gates has three inputs, and each of the three NOT gates has one input.



(a) A minimal sum-of-products realization



(b) A minimal product-of-sums realization

Figure 2.19 Two realizations of the function in Figure 2.18.

Minterms, with their row-number subscripts, can also be used to specify a given function in a more concise form. For example, the function in Figure 2.18 can be specified as

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6)$$

or even more simply as

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

The $\sum$ sign denotes the logical sum operation. This shorthand notation is often used in practice.

Maxterms

The principle of duality suggests that if it is possible to synthesize a function f by considering the rows in the truth table for which $f = 1$, then it should also be possible to synthesize f by considering the rows for which $f = 0$. This alternative approach uses the complements of minterms, which are called *maxterms*. All possible maxterms for three-variable functions are listed in Figure 2.17. We will refer to a maxterm M_j by the same row number as its corresponding minterm m_j as shown in the figure.

Product-of-Sums Form

If a given function f is specified by a truth table, then its complement $\overline{f}$ can be represented by a sum of minterms for which $\overline{f} = 1$, which are the rows where $f = 0$. For example, for the function in Figure 2.15

$$\begin{aligned}\overline{f}(x_1, x_2) &= m_2\\ &= x_1\overline{x}_2\end{aligned}$$

If we complement this expression using DeMorgan's theorem, the result is

$$\begin{aligned}\overline{\overline{f}} = f &= \overline{x_1\overline{x}_2}\\ &= \overline{x}_1 + x_2\end{aligned}$$

Note that we obtained this expression previously by algebraic manipulation of the canonical sum-of-products form for the function f. The key point here is that

$$f = \overline{m}_2 = M_2$$

where M_2 is the maxterm for row 2 in the truth table.

As another example, consider again the function in Figure 2.18. The complement of this function can be represented as

$$\begin{aligned}\overline{f}(x_1, x_2, x_3) &= m_0 + m_2 + m_3 + m_7\\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1x_2x_3\end{aligned}$$

Then f can be expressed as

$$\begin{aligned}f &= \overline{m_0 + m_2 + m_3 + m_7}\\ &= \overline{m}_0 \cdot \overline{m}_2 \cdot \overline{m}_3 \cdot \overline{m}_7\end{aligned}$$

$$= M_0 \cdot M_2 \cdot M_3 \cdot M_7$$
$$= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$$

This expression represents f as a product of maxterms.

A logic expression consisting of sum (OR) terms that are the factors of a logical product (AND) is said to be of the *product-of-sums* form. If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function. Any function f can be synthesized by finding its canonical product-of-sums. This involves taking the maxterm for each row in the truth table for which $f = 0$ and forming a product of these maxterms.

Returning to the preceding example, we can attempt to reduce the complexity of the derived expression that comprises a product of maxterms. Using the commutative property 10*b* and the associative property 11*b* from section 2.5, this expression can be written as

$$f = ((x_1 + x_3) + x_2)((x_1 + x_3) + \overline{x}_2)(x_1 + (\overline{x}_2 + \overline{x}_3))(\overline{x}_1 + (\overline{x}_2 + \overline{x}_3))$$

Then, using the combining property 14*b*, the expression reduces to

$$f = (x_1 + x_3)(\overline{x}_2 + \overline{x}_3)$$

The corresponding network is given in Figure 2.19*b*. The cost of this network is 13. While this cost happens to be the same as the cost of the sum-of-products version in Figure 2.19*a*, the reader should not assume that the cost of a network derived in the sum-of-products form will in general be equal to the cost of a corresponding circuit derived in the product-of-sums form.

Using the shorthand notation, an alternative way of specifying our sample function is

$$f(x_1, x_2, x_3) = \Pi(M_0, M_2, M_3, M_7)$$

or more simply

$$f(x_1, x_2, x_3) = \Pi M(0, 2, 3, 7)$$

The Π sign denotes the logical product operation.

The preceding discussion has shown how logic functions can be realized in the form of logic circuits, consisting of networks of gates that implement basic functions. A given function may be realized with circuits of a different structure, which usually implies a difference in cost. An important objective for a designer is to minimize the cost of the designed circuit. We will discuss the most important techniques for finding minimum-cost implementations in Chapter 4.

2.7 Design Examples

Logic circuits provide a solution to a problem. They implement functions that are needed to carry out specific tasks. Within the framework of a computer, logic circuits provide complete capability for execution of programs and processing of data. Such circuits are complex and difficult to design. But regardless of the complexity of a given circuit, a designer of logic circuits is always confronted with the same basic issues. First, it is necessary to specify the desired behavior of the circuit. Second, the circuit has to be synthesized and implemented.

Finally, the implemented circuit has to be tested to verify that it meets the specifications. The desired behavior is often initially described in words, which then must be turned into a formal specification. In this section we give two simple examples of design.

2.7.1 THREE-WAY LIGHT CONTROL

Assume that a large room has three doors and that a switch near each door controls a light in the room. It has to be possible to turn the light on or off by changing the state of any one of the switches.

As a first step, let us turn this word statement into a formal specification using a truth table. Let x_1, x_2, and x_3 be the input variables that denote the state of each switch. Assume that the light is off if all switches are open. Closing any one of the switches will turn the light on. Then turning on a second switch will have to turn off the light. Thus the light will be on if exactly one switch is closed, and it will be off if two (or no) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch. If $f(x_1, x_2, x_3)$ represents the state of the light, then the required functional behavior can be specified as shown in the truth table in Figure 2.20. The canonical sum-of-products expression for the specified function is

$$\begin{aligned} f &= m_1 + m_2 + m_4 + m_7 \\ &= \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_1x_2x_3 \end{aligned}$$

This expression cannot be simplified into a lower-cost sum-of-products expression. The resulting circuit is shown in Figure 2.21*a*.

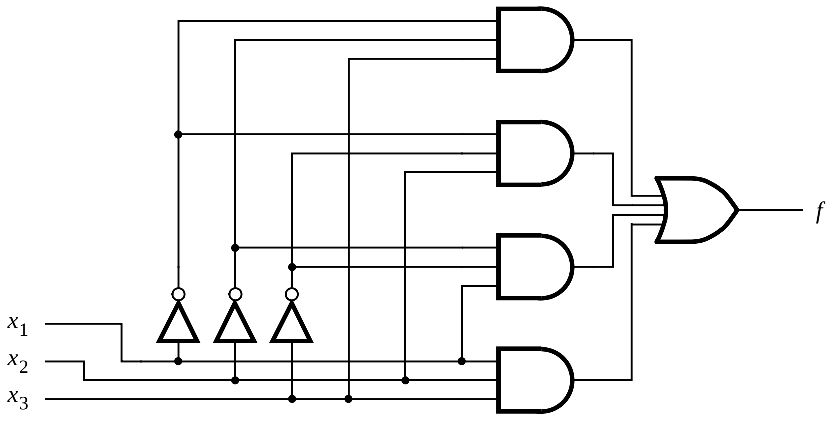
An alternative realization for this function is in the product-of-sums forms. The canonical expression of this type is

$$\begin{aligned} f &= M_0 \cdot M_3 \cdot M_5 \cdot M_6 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + x_3) \end{aligned}$$

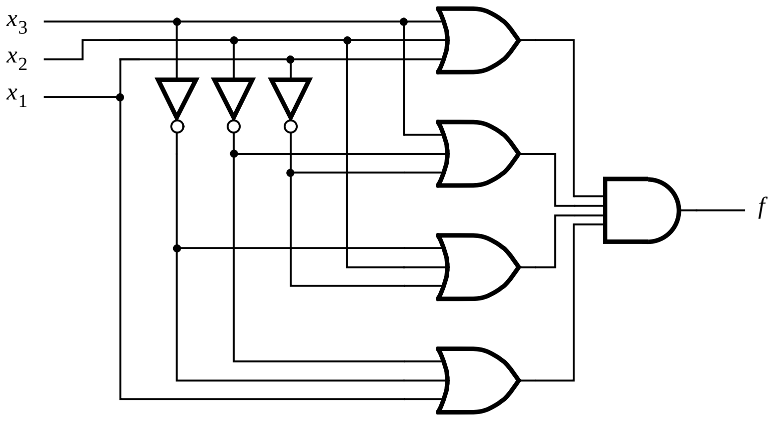
The resulting circuit is depicted in Figure 2.21*b*. It has the same cost as the circuit in part (*a*) of the figure.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2.20 Truth table for the three-way light control.



(a) Sum-of-products realization



(b) Product-of-sums realization

Figure 2.21 Implementation of the function in Figure 2.20.

When the designed circuit is implemented, it can be tested by applying the various input valuations to the circuit and checking whether the output corresponds to the values specified in the truth table. A straightforward approach is to check that the correct output is produced for all eight possible input valuations.

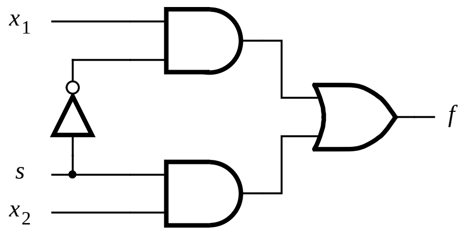
2.7.2 MULTIPLEXER CIRCUIT

In computer systems it is often necessary to choose data from exactly one of a number of possible sources. Suppose that there are two sources of data, provided as input signals x_1 and x_2. The values of these signals change in time, perhaps at regular intervals. Thus sequences of 0s and 1s are applied on each of the inputs x_1 and x_2. We want to design a circuit that produces an output that has the same value as either x_1 or x_2, dependent on the value of a selection control signal s. Therefore, the circuit should have three inputs: x_1, x_2, and s. Assume that the output of the circuit will be the same as the value of input x_1 if $s = 0$, and it will be the same as x_2 if $s = 1$.

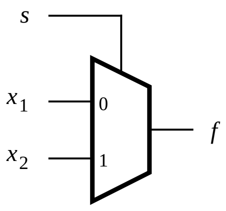
Based on these requirements, we can specify the desired circuit in the form of a truth table given in Figure 2.22*a*. From the truth table, we derive the canonical sum of products

$s\ x_1 x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

(a) Truth table



(b) Circuit



(c) Graphical symbol

s	$f(s, x_1, x_2)$
0	x_1
1	x_2

(d) More compact truth-table representation

Figure 2.22 Implementation of a multiplexer.

$$f(s, x_1, x_2) = \bar{s}x_1\bar{x}_2 + \bar{s}x_1x_2 + s\bar{x}_1x_2 + sx_1x_2$$

Using the distributive property, this expression can be written as

$$f = \bar{s}x_1(\bar{x}_2 + x_2) + s(\bar{x}_1 + x_1)x_2$$

Applying theorem 8*b* yields

$$f = \bar{s}x_1 \cdot 1 + s \cdot 1 \cdot x_2$$

Finally, theorem 6*a* gives

$$f = \bar{s}x_1 + sx_2$$

A circuit that implements this function is shown in Figure 2.22*b*. Circuits of this type are used so extensively that they are given a special name. A circuit that generates an output that exactly reflects the state of one of a number of data inputs, based on the value of one or more selection control inputs, is called a *multiplexer*. We say that a multiplexer circuit "multiplexes" input signals onto a single output.

In this example we derived a multiplexer with two data inputs, which is referred to as a "2-to-1 multiplexer." A commonly used graphical symbol for the 2-to-1 multiplexer is shown in Figure 2.22*c*. The same idea can be extended to larger circuits. A 4-to-1 multiplexer has four data inputs and one output. In this case two selection control inputs are needed to choose one of the four data inputs that is transmitted as the output signal. An 8-to-1 multiplexer needs eight data inputs and three selection control inputs, and so on.

Note that the statement "$f = x_1$ if $s = 0$, and $f = x_2$ if $s = 1$" can be presented in a more compact form of a truth table, as indicated in Figure 2.22*d*. In later chapters we will have occasion to use such representation.

We showed how a multiplexer can be built using AND, OR, and NOT gates. In Chapter 3 we will show other possibilities for constructing multiplexers. In Chapter 6 we will discuss the use of multiplexers in considerable detail.

Designers of logic circuits rely heavily on CAD tools. We want to encourage the reader to become familiar with the CAD tool support provided with this book as soon as possible. We have reached a point where an introduction to these tools is useful. The next section presents some basic concepts that are needed to use these tools. We will also introduce, in section 2.9, a special language for describing logic circuits, called VHDL. This language is used to describe the circuits as an input to the CAD tools, which then proceed to derive a suitable implementation.

2.8 INTRODUCTION TO CAD TOOLS

The preceding sections introduced a basic approach for synthesis of logic circuits. A designer could use this approach manually for small circuits. However, logic circuits found in complex systems, such as today's computers, cannot be designed manually—they are designed using sophisticated CAD tools that automatically implement the synthesis techniques.

To design a logic circuit, a number of CAD tools are needed. They are usually packaged together into a *CAD system*, which typically includes tools for the following tasks: design entry, synthesis and optimization, simulation, and physical design. We will introduce some of these tools in this section and will provide additional discussion in later chapters.

2.8.1 DESIGN ENTRY

The starting point in the process of designing a logic circuit is the conception of what the circuit is supposed to do and the formulation of its general structure. This step is done manually by the designer because it requires design experience and intuition. The rest of the design process is done with the aid of CAD tools. The first stage of this process involves entering into the CAD system a description of the circuit being designed. This stage is called *design entry*. We will describe three design entry methods: using truth tables, using schematic capture, and writing source code in a hardware description language.

Design Entry with Truth Tables

We have already seen that any logic function of a few variables can be described conveniently by a truth table. Many CAD systems allow design entry using truth tables, where the table is specified as a plain text file. Alternatively, it may also be possible to specify a truth table as a set of waveforms in a timing diagram. We illustrated the equivalence of these two ways of representing truth tables in the discussion of Figure 2.10. The CAD system provided with this book supports both methods of using truth tables for design entry. Figure 2.23 shows an example in which the *Waveform Editor* is used to draw the timing diagram in Figure 2.10. The CAD system is capable of transforming this timing diagram automatically into a network of logic gates equivalent to that shown in Figure 2.10*d*.

Because truth tables are practical only for functions with a small number of variables, this design entry method is not appropriate for large circuits. It can, however, be applied for a small logic function that is part of a larger circuit. In this case the truth table becomes a subcircuit that can be interconnected to other subcircuits and logic gates. The most commonly used type of CAD tool for interconnecting such circuit elements is called a *schematic capture* tool. The word *schematic* refers to a diagram of a circuit in which circuit elements, such as logic gates, are depicted as graphical symbols and connections between circuit elements are drawn as lines.

Schematic Capture

A schematic capture tool uses the graphics capabilities of a computer and a computer mouse to allow the user to draw a schematic diagram. To facilitate inclusion of basic gates in the schematic, the tool provides a collection of graphical symbols that represent gates

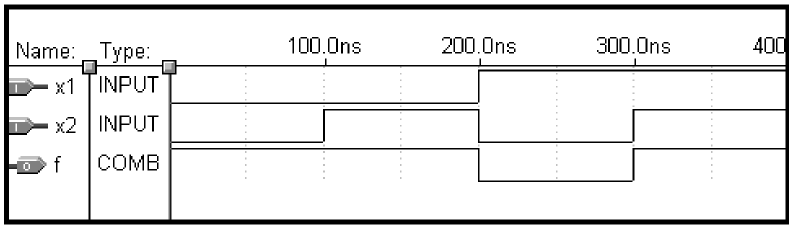


Figure 2.23 Screen capture of the Waveform Editor.

of various types with different numbers of inputs. This collection of symbols is called a *library*. The gates in the library can be imported into the user's schematic, and the tool provides a graphical way of interconnecting the gates to create a logic network.

Any subcircuits that have been previously created, using either different design entry methods or the schematic capture tool itself, can be represented as graphical symbols and included in the schematic. In practice it is common for a CAD system user to create a circuit that includes within it other smaller circuits. This methodology is known as *hierarchical design* and provides a good way of dealing with the complexities of large circuits.

Figure 2.24 gives an example of a hierarchical design created with the schematic capture tool, provided with the CAD system, called the *Graphic Editor*. The circuit includes a subcircuit represented as a rectangular graphical symbol. This subcircuit represents the logic function entered by way of the timing diagram in Figure 2.23. Note that the complete circuit implements the function $f = \overline{x}_1 + x_2\overline{x}_3$.

In comparison to design entry with truth tables, the schematic-capture facility is more amenable for dealing with larger circuits. A disadvantage of using schematic capture is that every commercial tool of this type has a unique user interface and functionality. Therefore, extensive training is often required for a designer to learn how to use such a tool, and this training must be repeated if the designer switches to another tool at a later date. Another drawback is that the graphical user interface for schematic capture becomes awkward to use when the circuit being designed is large. A useful method for dealing with large circuits is to write source code using a hardware description language to represent the circuit.

Hardware Description Languages

A *hardware description language (HDL)* is similar to a typical computer programming language except that an HDL is used to describe hardware rather than a program to be executed on a computer. Many commercial HDLs are available. Some are proprietary, meaning that they are provided by a particular company and can be used to implement circuits only in the technology provided by that company. We will not discuss the proprietary HDLs in this book. Instead, we will focus on a language that is supported by virtually all vendors that provide digital hardware technology and is officially endorsed as an *Institute of Electrical and Electronics Engineers (IEEE)* standard. The IEEE is a worldwide organization that promotes technical activities to the benefit of society in general. One of its activities involves the development of standards that define how certain technological concepts can be used in a way that is suitable for a large body of users.

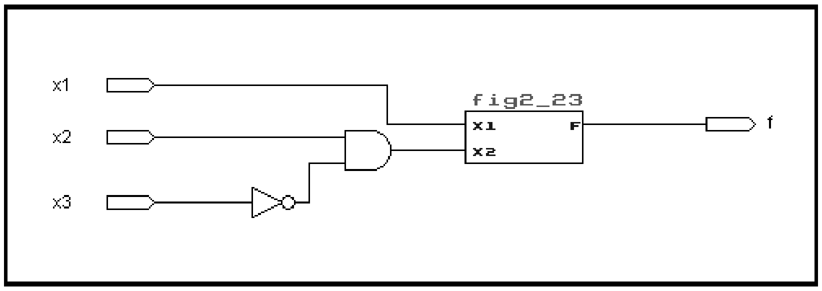


Figure 2.24 Screen capture of the Graphic Editor.

Two HDLs are IEEE standards: *VHDL (Very High Speed Integrated Circuit Hardware Description Language)* and *Verilog HDL*. Both languages are in widespread use in the industry. We use VHDL in this book because it is more popular than Verilog HDL. Although the two languages differ in many ways, the choice of using one or the other when studying logic circuits is not particularly important, because both offer similar features. Concepts illustrated in this book using VHDL can be directly applied when using Verilog HDL.

In comparison to performing schematic capture, using VHDL offers a number of advantages. Because it is supported by most companies that offer digital hardware technology, VHDL provides design *portability*. A circuit specified in VHDL can be implemented in different types of chips and with CAD tools provided by different companies, without having to change the VHDL specification. Design portability is an important advantage because digital circuit technology changes rapidly. By using a standard language, the designer can focus on the required functionality of the desired circuit without being overly concerned about the details of the technology that will eventually be used for implementation.

Design entry of a logic circuit is done by writing VHDL code. Signals in the circuit are represented as variables in the source code, and logic functions are expressed by assigning values to these variables. VHDL source code is plain text, which makes it easy for the designer to include within the code documentation that explains how the circuit works. This feature, coupled with the fact that VHDL is widely used, encourages sharing and reuse of VHDL-described circuits. This allows faster development of new products in cases where existing VHDL code can be adapted for use in the design of new circuits.

Similar to the way in which large circuits are handled in schematic capture, VHDL code can be written in a modular way that facilitates hierarchical design. Both small and large logic circuit designs can be efficiently represented in VHDL code. VHDL has been used to define circuits such as microprocessors with millions of transistors.

VHDL design entry can be combined with other methods. For example, a schematic-capture tool can be used in which a subcircuit in the schematic is described using VHDL. We will introduce VHDL in section 2.9.

2.8.2 SYNTHESIS

In section 2.4.1 we said that synthesis is the process of generating a logic circuit from a truth table. Synthesis CAD tools perform this process automatically. However, the synthesis tools also handle many other tasks. The process of *translating*, or *compiling*, VHDL code into a network of logic gates is part of synthesis.

When the VHDL code representing a circuit is passed through initial synthesis tools, the output is a lower-level description of the circuit. For simplicity we will assume that this process produces a set of logic expressions that describe the logic functions needed to realize the circuit. These expressions are then manipulated further by the synthesis tools. If the design entry is performed using schematic capture, then the synthesis tools produce a set of logic equations representing the circuit from the schematic diagram. Similarly, if truth tables are used for design entry, then the synthesis tools generate expressions for the logic functions represented by the truth tables.

Regardless of what type of design entry is used, the initial logic expressions produced by the synthesis tools are not likely to be in an optimal form. Because these expressions reflect the designer's input to the CAD tools, it is difficult for a designer to manually produce optimal results, especially for large circuits. One of the most important tasks of the synthesis tools is to manipulate the user's design to automatically produce an equivalent but better circuit. This step of synthesis is called *logic synthesis*, or *logic optimization*.

The measure of what makes one circuit better than another depends on the particular needs of a design project and the technology chosen for implementation. In section 2.6 we suggested that a good circuit might be one that has the lowest cost. There are other possible optimization goals, which are motivated by the type of hardware technology used for implementation of the circuit. We will discuss implementation technologies in Chapter 3 and return to the issue of optimization goals in Chapter 4.

After logic synthesis the optimized circuit is still represented in the form of logic equations. The final task in the synthesis process is to determine exactly how the circuit will be realized in a specific hardware technology. This task involves deciding how each logic function, represented by an expression, should be implemented using whatever physical resources are available in the technology. The task involves two steps called *technology mapping*, followed by *layout synthesis*, or *physical design*. We will discuss these steps in detail in Chapter 4.

2.8.3 FUNCTIONAL SIMULATION

Once the design entry and synthesis are complete, it is useful to verify that the designed circuit functions as expected. The tool that performs this task is called a *functional simulator*, and it uses two types of information. First, the user's initial design is represented by the logic equations generated during synthesis. Second, the user specifies valuations of the circuit's inputs that should be applied to these equations during simulation. For each valuation, the simulator evaluates the outputs produced by the equations. The output of the simulation is provided either in truth-table form or as a timing diagram. The user examines this output to verify that the circuit operates as required.

The logic equations used by the simulator are those produced by the synthesis tools before any optimizations are applied during logic synthesis. There would be no advantage in using the optimized form of the equations, because the intent is to evaluate the basic functionality of the design, which does not change as a result of optimization. The functional simulator assumes that the time needed for signals to propagate through the logic gates is negligible. In real logic gates this assumption is not realistic, regardless of the hardware technology chosen for implementation of the circuit. However, the functional simulation provides a first step in validating the basic operation of a design without concern for the effects of implementation technology. Accurate simulations that account for the timing details related to technology can be obtained by using a *timing simulator*. We will discuss timing simulation in Chapter 4.

2.8.4 SUMMARY

The CAD tools discussed in this section form a part of a CAD system. A typical design flow that the user follows is illustrated in Figure 2.25. After the design entry, initial synthesis tools perform various steps. For a function described by a truth table, the synthesis approach

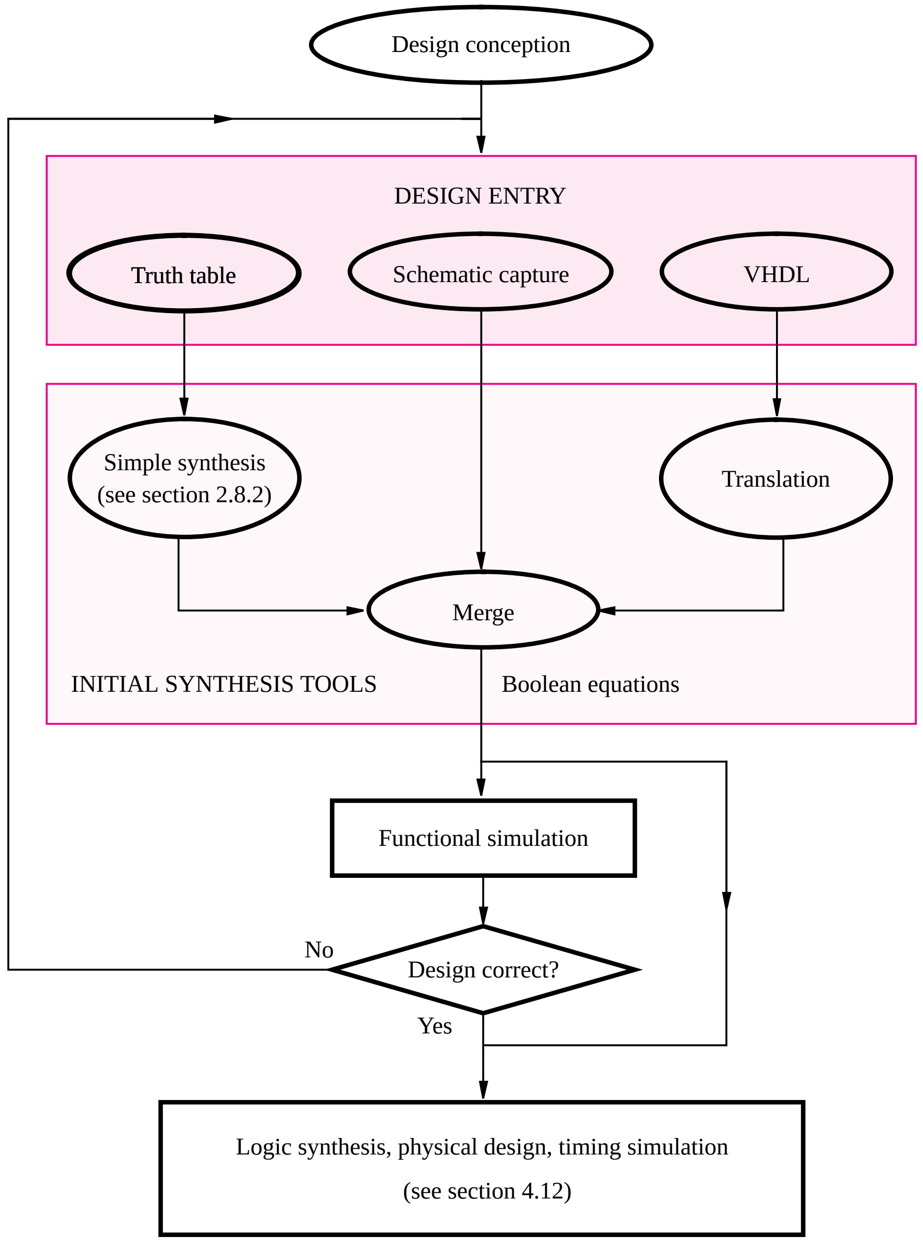


Figure 2.25 The first stages of a typical CAD system.

discussed in section 2.6 is applied to produce a logic expression for the function. For VHDL the translation process turns the VHDL source code into logic functions, which can be represented as logic expressions. As mentioned earlier, the designer can use a mixture of design entry methods. In Figure 2.25 this flexibility is reflected by the step labeled Merge, in which the components produced using any of the design entry methods are automatically merged into a single design. At this point the circuit is represented in the CAD system as a set of logic equations.

After the initial synthesis the correct operation of the designed circuit can be verified by using functional simulation. As shown in Figure 2.25, this step is not a requirement in the CAD flow and can be skipped at the designer's discretion. In practice, however, it is wise to verify that the designed circuit works as expected as early in the design process as possible. Any problems discovered during the simulation are fixed by returning to the design entry stage. Once errors are no longer apparent, the designer proceeds with the remaining tools in the CAD flow. These include logic synthesis, layout synthesis, timing simulation, and others. We have mentioned these tools only briefly thus far. The remaining CAD steps will be described in Chapter 4.

At this point the reader should have some appreciation for what is involved when using CAD tools. However, the tools can be fully appreciated only when they are used firsthand. In Appendexes B to D, we provide step-by-step tutorials that illustrate how to use the MAX+plusII CAD system, which is included with this book. The tutorial in Appendix B covers design entry with both schematic capture and VHDL, as well as functional simulation. We strongly encourage the reader to work through the hands-on material. Because the tutorial uses VHDL for design entry, we provide an introduction to VHDL in the following section.

2.9 Introduction to VHDL

In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. VHDL was developed as a part of that effort. VHDL has become the industry standard language for describing digital circuits, largely because it is an official IEEE standard. The original standard for VHDL was adopted in 1987 and called IEEE 1076. A revised standard was adopted in 1993 and called IEEE 1164.

VHDL was originally intended to serve two main purposes. First, it was used as a documentation language for describing the structure of complex digital circuits. As an official IEEE standard, VHDL provided a common way of documenting circuits designed by numerous designers. Second, VHDL provided features for modeling the behavior of a digital circuit, which allowed its use as input to software programs that were then used to simulate the circuit's operation.

In recent years, in addition to its use for documentation and simulation, VHDL has also become popular for use in design entry in CAD systems. The CAD tools are used to synthesize the VHDL code into a hardware implementation of the described circuit. In this book our main use of VHDL will be for synthesis.

VHDL is an extremely complex, sophisticated language. Learning all of its features is a daunting task. However, for use in synthesis only a subset of these features is important. To avoid confusion in learning this complex language, we will discuss only the features of VHDL that are actually used in the examples in the book. The material presented should be sufficient to allow the reader to design a wide range of circuits. The reader who wishes to learn the complete VHDL language can refer to one of the specialized texts [4–8].

To further simplify the task of learning VHDL, we will introduce the language in several stages throughout the book. Our general approach will be to introduce particular features only when they are relevant to the design topics covered in that part of the text. For convenience, in Appendix A we provide a complete listing of the VHDL features covered in the book. The reader may wish to refer to that material from time to time. In the remainder of this section, we discuss the most basic concepts needed to write simple VHDL code.

2.9.1 Representation of Digital Signals in VHDL

When using CAD tools to synthesize a logic circuit, the designer can provide the initial description of the circuit in several different ways, as we explained in section 2.8.1. One convenient way is to write this description in the form of VHDL source code. The VHDL compiler translates this code into a logic circuit. Each logic signal in the circuit is represented in VHDL code as a data object. Just as the variables declared in any high-level programming language have associated types, such as integers or characters, data objects in VHDL can be of various types. The original VHDL standard, IEEE 1076, includes a data type called *BIT*. An object of this type is well suited for representing digital signals because BIT objects can have only two values, 0 and 1. In this chapter all signals in our examples will be of type BIT. Other data types are introduced in section 4.11 and are listed in Appendix A.

2.9.2 Writing Simple VHDL Code

We will use an example to illustrate how to write simple VHDL source code. Consider the logic circuit in Figure 2.26. If we wish to write VHDL code to represent this circuit, the first step is to declare the input and output signals. This is done using a construct called an *entity*. An appropriate entity for this example appears in Figure 2.27. An entity must

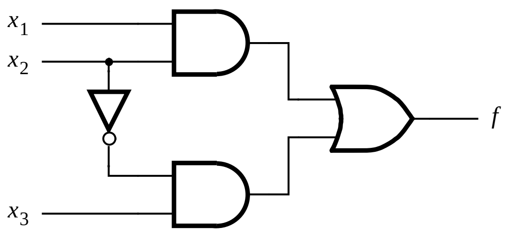


Figure 2.26 A simple logic function.

```
ENTITY example1 IS
    PORT ( x1, x2, x3  : IN    BIT ;
               f       : OUT   BIT ) ;
END example1 ;
```

Figure 2.27 VHDL entity declaration for the circuit in Figure 2.26.

be assigned a name; we have chosen the name *example1* for this first example. The input and output signals for the entity are called its *ports*, and they are identified by the keyword PORT. This name derives from the electrical jargon in which the word *port* refers to an input or output connection to an electronic circuit. Each port has an associated *mode* that specifies whether it is an input (IN) to the entity or an output (OUT) from the entity. Each port represents a signal, hence it has an associated type. The entity *example1* has four ports in total. The first three, x_1, x_2, and x_3, are input signals of type BIT. The port named f is an output of type BIT.

In Figure 2.27 we have used simple signal names *x1*, *x2*, *x3*, and *f* for the entity's ports. Similar to most computer programming languages, VHDL has rules that specify which characters are allowed in signal names. A simple guideline is that signal names can include any letter or number, as well as the underscore character '_'. There are two caveats: a signal name must begin with a letter, and a signal name cannot be a VHDL keyword.

An entity specifies the input and output signals for a circuit, but it does not give any details as to what the circuit represents. The circuit's functionality must be specified with a VHDL construct called an *architecture*. An architecture for our example appears in Figure 2.28. It must be given a name, and we have chosen the name *LogicFunc*. Although the name can be any text string, it is sensible to assign a name that is meaningful to the designer. In this case we have chosen the name *LogicFunc* because the architecture specifies the functionality of the design using a logic expression. VHDL has built-in support for the following Boolean operators: AND, OR, NOT, NAND, NOR, XOR, and XNOR. (So far we have introduced only AND, OR, and NOT operators; the others will be presented in Chapter 3.) Following the BEGIN keyword, our architecture specifies, using the VHDL signal assignment operator <=, that output *f* should be assigned the result of the logic expression on the right-hand side of the operator. Because VHDL does not assume any precedence of logic operators, parentheses are used in the expression. One might expect that an assignment statement such as

$$f <= x1 \text{ AND } x2 \text{ OR NOT } x2 \text{ AND } x3$$

would have implied parentheses

$$f <= (x1 \text{ AND } x2) \text{ OR } ((\text{NOT } x2) \text{ AND } x3)$$

But for VHDL code this assumption is not true. In fact, without the parentheses the VHDL compiler would produce a compile-time error for this expression.

Complete VHDL code for our example is given in Figure 2.29. This example has illustrated that a VHDL source code file has two main sections: an entity and an architecture.

```
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Figure 2.28 VHDL architecture for the entity in Figure 2.27.

```
ENTITY example1 IS
      PORT ( x1, x2, x3   : IN      BIT ;
                 f                : OUT  BIT ) ;
END example1 ;

ARCHITECTURE LogicFunc OF example1 IS
BEGIN
      f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Figure 2.29 Complete VHDL code for the circuit in Figure 2.26.

A simple analogy for what each section represents is that the entity is equivalent to a symbol in a schematic diagram and the architecture specifies the logic circuitry inside the symbol.

A second example of VHDL code is given in Figure 2.30. This circuit has four input signals, called $x1$, $x2$, $x3$, and $x4$, and two output signals, named f and g. A logic expression is assigned to each output. A logic circuit produced by the VHDL compiler for this example is shown in Figure 2.31.

The preceding two examples indicate that one way to assign a value to a signal in VHDL code is by means of a logic expression. In VHDL terminology a logic expression is called a *simple assignment statement*. We will see later that VHDL also supports several other types of assignment statements and many other features that are useful for describing circuits that are much more complex.

2.9.3 How *not* to Write VHDL Code

When learning how to use VHDL or other hardware description languages, the tendency for the novice is to write code that resembles a computer program, containing many variables

```
ENTITY example2 IS
      PORT ( x1, x2, x3, x4   : IN      BIT ;
                 f, g                  : OUT  BIT ) ;
END example2 ;

ARCHITECTURE LogicFunc OF example2 IS
BEGIN
      f <=  (x1 AND x3) OR (NOT x3 AND x2) ;
      g <= (NOT x3 OR x1) AND (NOT x3 OR x4) ;
END LogicFunc ;
```

Figure 2.30 VHDL code for a four-input function.

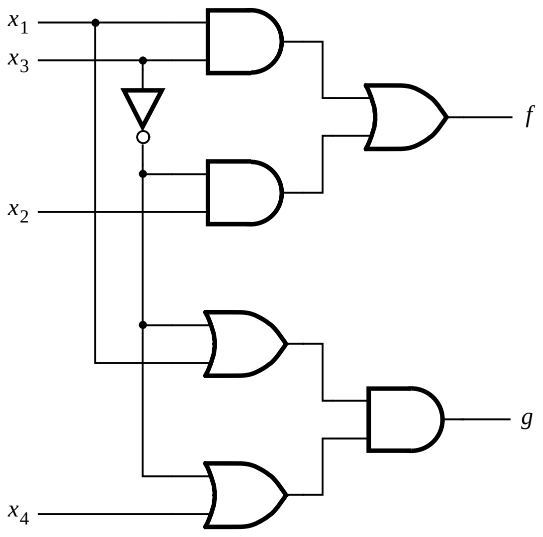


Figure 2.31 Logic circuit for the code in Figure 2.30.

and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesizing such code. This book contains more than 100 examples of complete VHDL code that represent a wide range of logic circuits. In these examples the code is easily related to the described logic circuit. The reader is advised to adopt the same style of code. A good general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the VHDL code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to describe.

Once complete VHDL code is written for a particular design, the reader is encouraged to analyze the resulting circuit synthesized by the CAD tools. Much can be learned about VHDL, logic circuits, and logic synthesis by studying the circuits that are produced automatically by the CAD tools.

2.10 CONCLUDING REMARKS

In this chapter we introduced the concept of logic circuits. We showed that such circuits can be implemented using logic gates and that they can be described using a mathematical model called Boolean algebra. Because practical logic circuits are often large, it is important to have good CAD tools to help the designer. This book is accompanied by the MAX+PlusII software, which is a CAD tool provided by Altera Corporation. We introduced a few basic features of this tool and urge the reader to start using this software as soon as possible.

Our discussion so far has been quite elementary. We will deal with both the logic circuits and the CAD tools in much more depth in the chapters that follow. But first, in

Chapter 3 we will examine the most important electronic technologies used to construct logic circuits. This material will give the reader an appreciation of practical constraints that a designer of logic circuits must face.

PROBLEMS

2.1 Use algebraic manipulation to prove that $x + yz = (x + y) \cdot (x + z)$. Note that this is the distributive rule, as stated in identity 12*b* in section 2.5.

2.2 Use algebraic manipulation to prove that $(x + y) \cdot (x + \overline{y}) = x$.

2.3 Use the Venn diagram to prove the identity in problem 1.

2.4 Use the Venn diagram to prove DeMorgan's theorem, as given in expressions 15*a* and 15*b* in section 2.5.

2.5 Use the Venn diagram to prove

$$(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \overline{x}_3) = x_1 + x_2$$

2.6 Determine whether or not the following expressions are valid, i.e., whether the left- and right-hand sides represent same function.
(a) $\overline{x}_1x_3 + x_1x_2\overline{x}_3 + \overline{x}_1x_2 + x_1\overline{x}_2 = \overline{x}_2x_3 + x_1\overline{x}_3 + x_2\overline{x}_3 + \overline{x}_1x_2x_3$
(b) $x_1\overline{x}_3 + x_2x_3 + \overline{x}_2\overline{x}_3 = (x_1 + \overline{x}_2 + x_3)(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)$
(c) $(x_1 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2) = (x_1 + x_2)(x_2 + x_3)(\overline{x}_1 + \overline{x}_3)$

2.7 Draw a timing diagram for the circuit in Figure 2.19*a*. Show the waveforms that can be observed on all wires in the circuit.

2.8 Repeat problem 2.7 for the circuit in Figure 2.19*b*.

2.9 Use algebraic manipulation to show that for three input variables x_1, x_2, and x_3

$$\sum m(1, 2, 3, 4, 5, 6, 7) = x_1 + x_2 + x_3$$

2.10 Use algebraic manipulation to show that for three input variables x_1, x_2, and x_3

$$\Pi \text{ M}(0, 1, 2, 3, 4, 5, 6) = x_1x_2x_3$$

2.11 Use algebraic manipulation to find the minimum sum-of-products expression for the function $f = x_1x_3 + x_1\overline{x}_2 + \overline{x}_1x_2x_3 + \overline{x}_1\overline{x}_2\overline{x}_3$.

2.12 Use algebraic manipulation to find the minimum sum-of-products expression for the function $f = x_1\overline{x}_2\overline{x}_3 + x_1x_2x_4 + x_1\overline{x}_2x_3\overline{x}_4$.

2.13 Use algebraic manipulation to find the minimum product-of-sums expression for the function $f = (x_1 + x_3 + x_4) \cdot (x_1 + \overline{x}_2 + x_3) \cdot (x_1 + \overline{x}_2 + \overline{x}_3 + x_4)$.

2.14 Use algebraic manipulation to find the minimum product-of-sums expression for the function $f = (x_1 + x_2 + x_3) \cdot (x_1 + \overline{x}_2 + x_3) \cdot (\overline{x}_1 + \overline{x}_2 + x_3) \cdot (x_1 + x_2 + \overline{x}_3)$.

2.15 (a) Show the location of all minterms in a three-variable Venn diagram.

(b) Show a separate Venn diagram for each product term in the function $f = x_1\overline{x}_2x_3 + x_1x_2 + \overline{x}_1x_3$. Use the Venn diagram to find the minimal sum-of-products form of f.

2.16 Represent the function in Figure 2.18 in the form of a Venn diagram and find its minimal sum-of-products form.

2.17 Figure P2.1 shows two attempts to draw a Venn diagram for four variables. For parts (*a*) and (*b*) of the figure, explain why the Venn diagram is not correct. (Hint: the Venn diagram must be able to represent all 16 minterms of the four variables.)

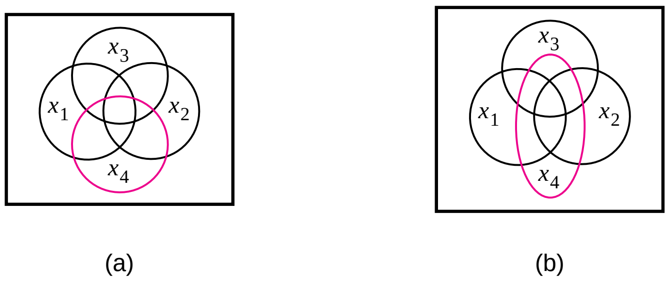


Figure P2.1 Two attempts to draw a four-variable Venn diagram.

2.18 Figure P2.2 gives a representation of a four-variable Venn diagram and shows the location of minterms m_0, m_1, and m_2. Show the location of the other minterms in the diagram. Represent the function $f = \overline{x}_1\overline{x}_2x_3\overline{x}_4 + x_1x_2x_3x_4 + \overline{x}_1x_2$ on this diagram.

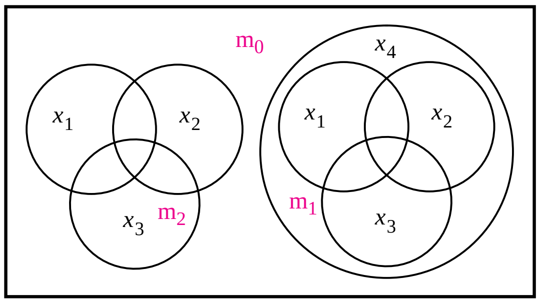


Figure P2.2 A four-variable Venn diagram.

2.19 Design the simplest sum-of-products circuit that implements the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$.

2.20 Design the simplest sum-of-products circuit that implements the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$.

2.21 Design the simplest product-of-sums circuit that implements the function $f(x_1, x_2, x_3) = \Pi\, M\,(0, 2, 5)$.

2.22 Design the simplest product-of-sums expression for the function $f(x_1, x_2, x_3) = \Pi\, M\,(0, 1, 5, 7)$.

2.23 Design the simplest circuit that has three inputs, x_1, x_2, and x_3, which produces an output value of 1 whenever two or more of the input variables have the value 1; otherwise, the output has to be 0.

2.24 For the timing diagram in Figure P2.3, synthesize the function $f(x_1, x_2, x_3)$ in the simplest sum-of-products form.

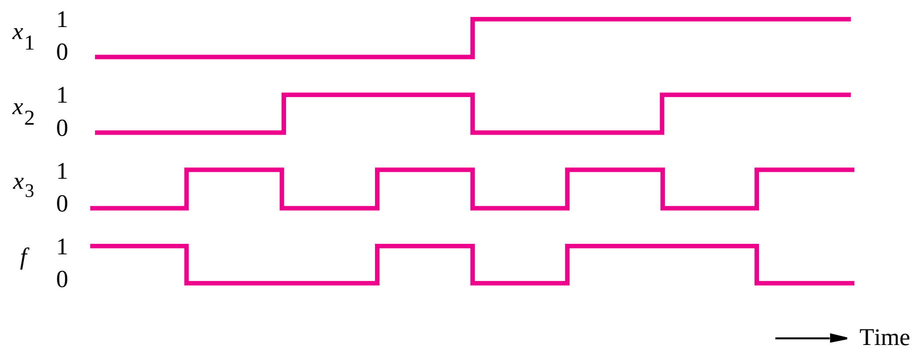


Figure P2.3 A timing diagram representing a logic function.

2.25 For the timing diagram in Figure P2.4, synthesize the function $f(x_1, x_2, x_3)$ in the simplest sum-of-products form.

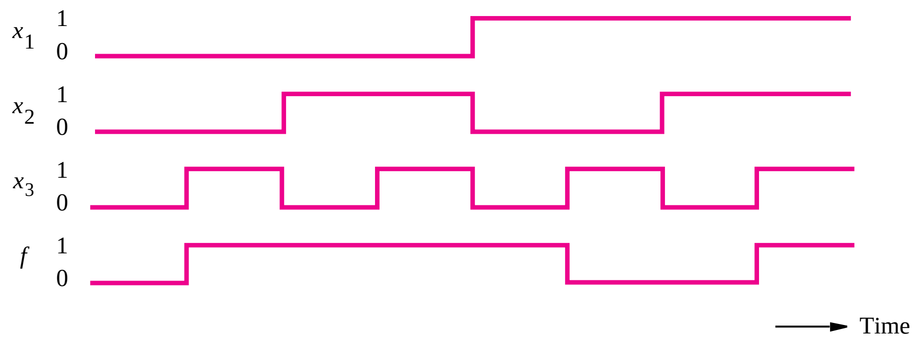


Figure P2.4 A timing diagram representing a logic function.

2.26 Design a circuit with output f and inputs x_1, x_0, y_1, and y_0. Let $X = x_1x_0$ be a number, where the four possible values of X, namely, 00, 01, 10, and 11, represent the four numbers 0, 1, 2, and 3, respectively. (We discuss representation of numbers in Chapter 5.) Similarly, let $Y = y_1y_0$ represent another number with the same four possible values. The output f should be 1 if the numbers represented by X and Y are not equal. Otherwise, f should be 0.
(a) Show the truth table for f.
(b) Synthesize the simplest possible product-of-sums expression for f.

2.27 Repeat problem 2.26 for the case where f should be 1 only if $X \geq Y$.
(a) Show the truth table for f.
(b) Show the canonical sum-of-products expression for f.
(c) Show the simplest possible sum-of-products expression for f.

2.28 (a) Use the Graphic Editor in MAX+plusII to draw schematics for the following functions

$$f_1 = x_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_2x_4 + \overline{x}_1x_2x_3 + x_1x_2x_3$$
$$f_2 = x_2\overline{x}_4 + \overline{x}_1x_2 + x_2x_3$$

(b) Use functional simulation in MAX+plusII to prove that $f_1 = f_2$.

2.29 (a) Use the Graphic Editor in MAX+plusII to draw schematics for the following functions

$$f_1 = (x_1 + x_2 + \overline{x}_4) \cdot (\overline{x}_2 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + \overline{x}_3 + \overline{x}_4)$$
$$f_2 = (x_2 + \overline{x}_4) \cdot (x_3 + \overline{x}_4) \cdot (\overline{x}_1 + \overline{x}_4)$$

(b) Use functional simulation in MAX+plusII to prove that $f_1 = f_2$.

2.30 (a) Using the Text Editor in MAX+plusII, write VHDL code to describe the following functions

$$f_1 = x_1\overline{x}_3 + x_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2 + x_1\overline{x}_4$$
$$f_2 = (x_1 + \overline{x}_3) \cdot (x_1 + x_2 + \overline{x}_4) \cdot (x_2 + \overline{x}_3 + \overline{x}_4)$$

(b) Use functional simulation in MAX+plusII to prove that $f_1 = f_2$.

2.31 Consider the following VHDL assignment statements

```
f1 <= ((x1 AND x3) OR (NOT x1 AND NOT x3)) AND ((x2 AND x4) OR
        (NOT x2 AND NOT x4)) ;
f2 <= (x1 AND x2 AND NOT x3 AND NOT x4) OR (NOT x1 AND NOT x2 AND x3 AND x4)
        OR (x1 AND NOT x2 AND NOT x3 AND x4) OR
        (NOT x1 AND x2 AND x3 AND NOT x4) ;
```

(a) Write complete VHDL code to implement f1 and f2.
(b) Use functional simulation in MAX+plusII to prove that $f1 = \overline{f2}$.

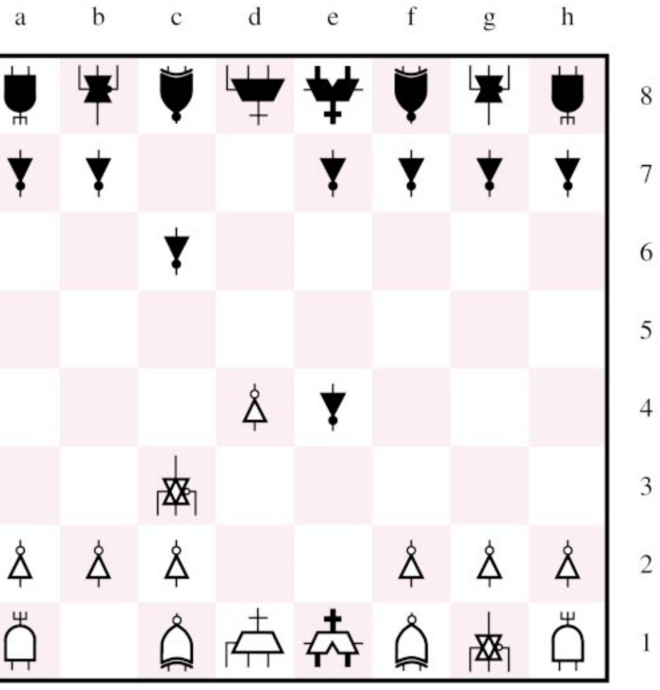
REFERENCES

1. G. Boole, *An Investigation of the Laws of Thought*, 1854, reprinted by Dover Publications, New York, 1954.
2. C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *Transactions of AIEE* 57 (1938), pp. 713–723.
3. E. V. Huntington, "Sets of Independent Postulates for the Algebra of Logic," *Transactions of the American Mathematical Society* 5 (1904), pp. 288–309.
4. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2nd ed. (McGraw-Hill: New York, 1998).
5. D. L. Perry, *VHDL*, 3rd ed. (McGraw-Hill: New York, 1998).
6. J. Bhasker, *A VHDL Primer* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
7. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
8. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, 1997).

chapter

3

Implementation Technology



3. Nb1–c3, d5xe4

In section 1.2 we said that logic circuits are implemented using transistors and that a number of different technologies exist. We now explore technology issues in more detail.

Let us first consider how logic variables can be physically represented as signals in electronic circuits. Our discussion will be restricted to binary variables, which can take on only the values 0 and 1. In a circuit these values can be represented either as levels of voltage or current. Both alternatives are used in different technologies. We will focus on the simplest and most popular representation, using voltage levels.

The most obvious way of representing two logic values as voltage levels is to define a *threshold* voltage; any voltage below the threshold represents one logic value, and voltages above the threshold correspond to the other logic value. It is an arbitrary choice as to which logic value is associated with the low and high voltage levels. Usually, logic 0 is represented by the low voltage levels and logic 1 by the high voltages. This is known as a *positive logic* system. The opposite choice, in which the low voltage levels are used to represent logic 1 and the higher voltages are used for logic 0 is known as a *negative logic* system. In this book we use only the positive logic system, but negative logic is discussed briefly in section 3.4.

Using the positive logic system, the logic values 0 and 1 are referred to simply as "low" and "high." To implement the threshold-voltage concept, a range of low and high voltage levels is defined, as shown in Figure 3.1. The figure gives the minimum voltage, called V_{SS}, and the maximum voltage, called V_{DD}, that can exist in the circuit. We will assume that V_{SS} is 0 volts, corresponding to electrical ground, denoted *Gnd*. The voltage V_{DD} represents the power supply voltage. The most common level for V_{DD} is 5 volts, but 3.3 volts is also popular. In this chapter we will usually assume that $V_{DD} = 5$ V. Figure 3.1 indicates that voltages in the range *Gnd* to $V_{0,max}$ represent logic value 0. The name $V_{0,max}$ means the maximum voltage level that a logic circuit must recognize as low. Similarly, the range from $V_{1,min}$ to V_{DD} corresponds to logic value 1, and $V_{1,min}$ is the minimum voltage level that a logic circuit must interpret as high. The exact levels of $V_{0,max}$ and $V_{1,min}$

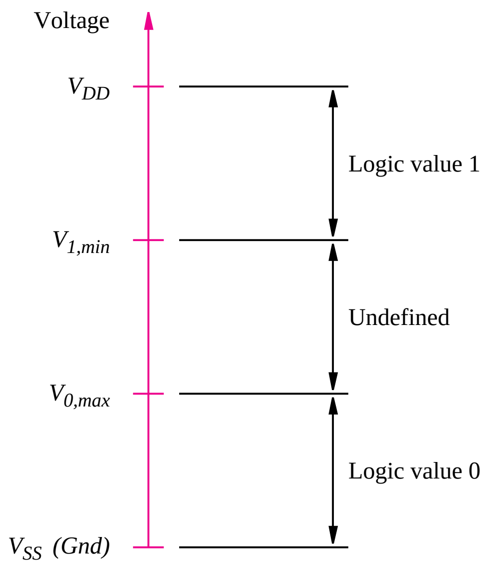


Figure 3.1 Representation of logic values by voltage levels.

depend on the particular technology used; a typical example might set $V_{0,max}$ to 40 percent of V_{DD} and $V_{1,min}$ to 60 percent of V_{DD}. The range of voltages between $V_{0,max}$ and $V_{1,min}$ is undefined. Logic signals do not normally assume voltages in this range except in transition from one logic value to the other. We will discuss the voltage levels used in logic circuits in more depth in section 3.8.3.

3.1 TRANSISTOR SWITCHES

Logic circuits are built with transistors. A full treatment of transistor behavior is beyond the scope of this text; it can be found in electronics textbooks, such as [1] and [2]. For the purpose of understanding how logic circuits are built, we can assume that a transistor operates as a simple switch. Figure 3.2*a* shows a switch controlled by a logic signal, *x*. When *x* is low, the switch is open, and when *x* is high, the switch is closed. The most popular type of transistor for implementing a simple switch is the *metal oxide semiconductor field-effect transistor (MOSFET)*. There are two different types of MOSFETs, known as *n-channel*, abbreviated *NMOS*, and *p-channel*, denoted *PMOS*.

Figure 3.2*b* gives a graphical symbol for an NMOS transistor. It has four electrical terminals, called the *source*, *drain*, *gate*, and *substrate*. In logic circuits the substrate (also

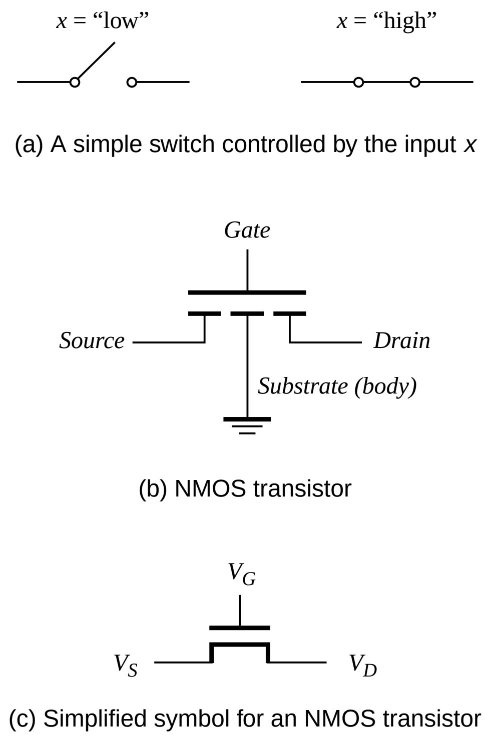

x = "low"
x = "high"
(a) A simple switch controlled by the input x
Gate
Source
Drain
Substrate (body)
(b) NMOS transistor
V_G
V_S
V_D
(c) Simplified symbol for an NMOS transistor

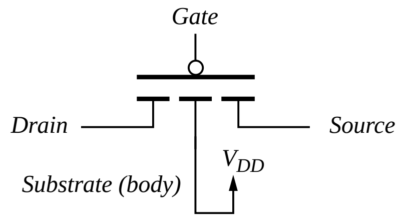
Figure 3.2 NMOS transistor as a switch.

called *body*) terminal is connected to *Gnd*. We will use the simplified graphical symbol in Figure 3.2*c*, which omits the substrate node. There is no physical difference between the source and drain terminals. They are distinguished in practice by the voltage levels applied to the transistor; by convention, the terminal with the lower voltage level is deemed to be the source.

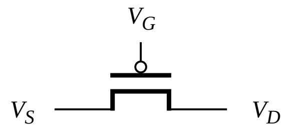
A detailed explanation of how the transistor operates will be presented in section 3.8.1. For now it is sufficient to know that it is controlled by the voltage V_G at the gate terminal. If V_G is low, then there is no connection between the source and drain, and we say that the transistor is *turned off*. If V_G is high, then the transistor is *turned on* and acts as a closed switch that connects the source and drain terminals. In section 3.8.2 we show how to calculate the resistance between the source and drain terminals when the transistor is turned on, but for now assume that the resistance is 0 Ω.

PMOS transistors have the opposite behavior of NMOS transistors. The former are used to realize the type of switch illustrated in Figure 3.3*a*, where the switch is open when the control input x is high and closed when x is low. A symbol is shown in Figure 3.3*b*. In logic circuits the substrate of the PMOS transistor is always connected to V_{DD}, leading to the simplified symbol in Figure 3.3*c*. If V_G is high, then the PMOS transistor is turned off and acts like an open switch. When V_G is low, the transistor is turned on and acts as a closed switch that connects the source and drain. In the PMOS transistor the source is the node with the higher voltage.

(a) A switch with the opposite behavior of Figure 3.2(*a*)



(b) PMOS transistor



(c) Simplified symbol for an PMOS transistor

Figure 3.3 PMOS transistor as a switch.

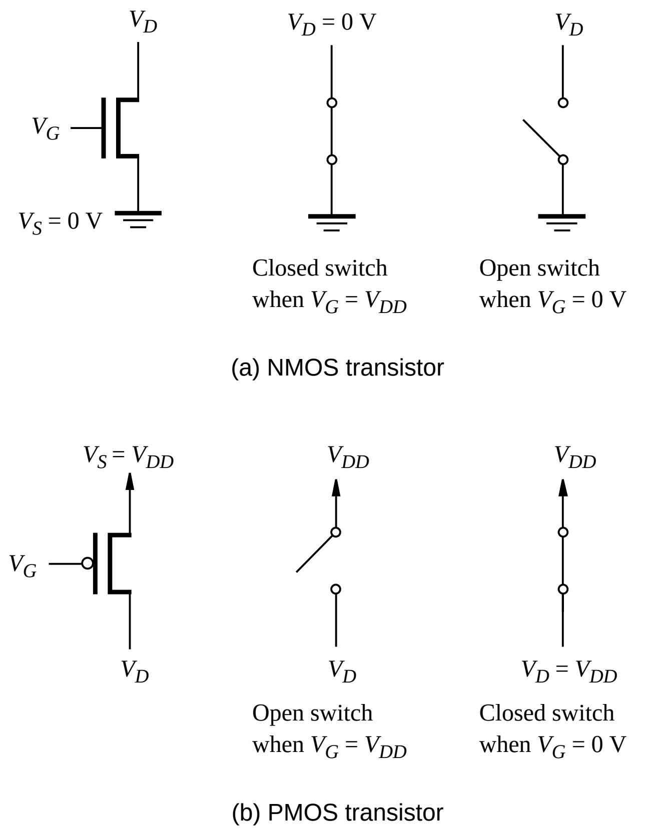


Figure 3.4 NMOS and PMOS transistors in logic circuits.

Figure 3.4 summarizes the typical use of NMOS and PMOS transistors in logic circuits. An NMOS transistor is turned on when its gate terminal is high, while a PMOS transistor is turned on when its gate is low. When the NMOS transistor is turned on, its drain is *pulled down* to *Gnd*, and when the PMOS transistor is turned on, its drain is *pulled up* to V_{DD}. Because of the way the transistors operate, an NMOS transistor cannot be used to pull its drain terminal completely up to V_{DD}. Similarly, a PMOS transistor cannot be used to pull its drain terminal completely down to *Gnd*. We discuss the operation of MOSFETs in considerable detail in section 3.8.

3.2 NMOS Logic Gates

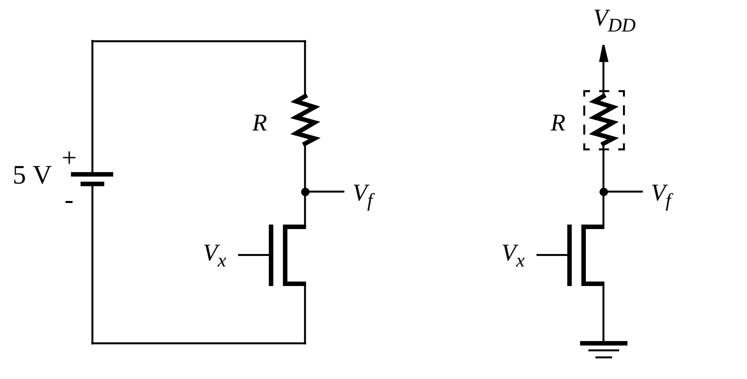
The first schemes for building logic gates with MOSFETs became popular in the 1970s and relied on either PMOS or NMOS transistors, but not both. Since the early 1980s, a combination of both NMOS and PMOS transistors has been used. We will first describe how logic circuits can be built using NMOS transistors because these circuits are easier to understand.

Such circuits are known as NMOS circuits. Then we will show how NMOS and PMOS transistors are combined in the presently popular technology known as *complementary MOS*, or *CMOS*.

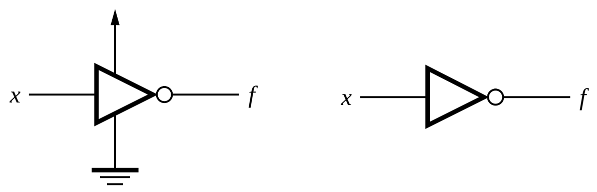
In the circuit in Figure 3.5*a*, when $V_x = 0$ V, the NMOS transistor is turned off. No current flows through the resistor *R*, and $V_f = 5$ V. On the other hand, when $V_x = 5$ V, the transistor is turned on and pulls V_f to a low voltage level. The exact voltage level of V_f in this case depends on the amount of current that flows through the resistor and transistor. Typically, V_f is about 0.2 V (see section 3.8.3). If V_f is viewed as a function of V_x, then the circuit is an NMOS implementation of a NOT gate. In logic terms this circuit implements the function $f = \overline{x}$. Figure 3.5*b* gives a simplified circuit diagram in which the connection to the positive terminal on the power supply is indicated by an arrow labeled V_{DD} and the connection to the negative power-supply terminal is indicated by the *Gnd* symbol. We will use this simplified style of circuit diagram throughout this chapter.

The purpose of the resistor in the NOT gate circuit is to limit the amount of current that flows when $V_x = 5$ V. Rather than using a resistor for this purpose, a transistor is normally used. We will discuss this issue in more detail in section 3.8.3. In subsequent diagrams a dashed box is drawn around the resistor *R* as a reminder that it is implemented using a transistor.

Figure 3.5*c* presents the graphical symbols for a NOT gate. The left symbol shows the input, output, power, and ground terminals, and the right symbol is simplified to show only



(a) Circuit diagram (b) Simplified circuit diagram



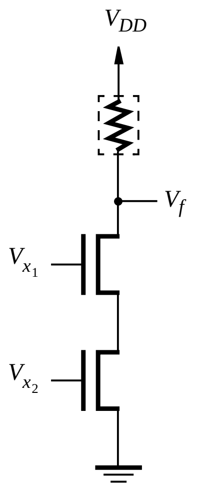
(c) Graphical symbols

Figure 3.5 A NOT gate built using NMOS technology.

the input and output terminals. In practice only the simplified symbol is used. Another name often used for the NOT gate is *inverter*. We use both names interchangeably in this book.

In section 2.1 we saw that a series connection of switches corresponds to the logic AND function, while a parallel connection represents the OR function. Using NMOS transistors, we can implement the series connection as depicted in Figure 3.6*a*. If $V_{x_1} = V_{x_2} = 5$ V, both transistors will be on and V_f will be close to 0 V. But if either V_{x_1} or V_{x_2} is 0, then no current will flow through the series-connected transistors and V_f will be pulled up to 5 V. The resulting truth table for f, provided in terms of logic values, is given in Figure 3.6*b*. The realized function is the complement of the AND function, called the *NAND* function, for NOT-AND. The circuit realizes a NAND gate. Its graphical symbols are shown in Figure 3.6*c*.

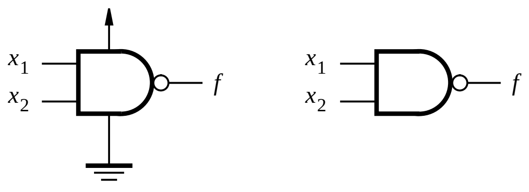
The parallel connection of NMOS transistors is given in Figure 3.7*a*. Here, if either $V_{x_1} = 5$ V or $V_{x_2} = 5$ V, then V_f will be close to 0 V. Only if both V_{x_1} and V_{x_2} are 0 will V_f be pulled up to 5 V. A corresponding truth table is given in Figure 3.7*b*. It shows that the circuit realizes the complement of the OR function, called the *NOR* function, for NOT-OR. The graphical symbols for the NOR gate appear in Figure 3.7*c*.



(a) Circuit

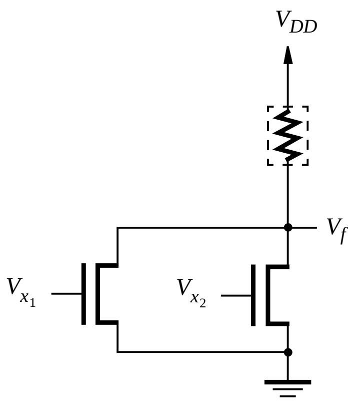
x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

(b) Truth table



(c) Graphical symbols

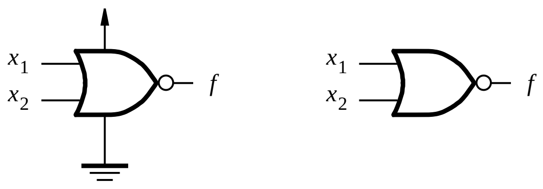
Figure 3.6 NMOS realization of a NAND gate.



(a) Circuit

x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	0

(b) Truth table



(c) Graphical symbols

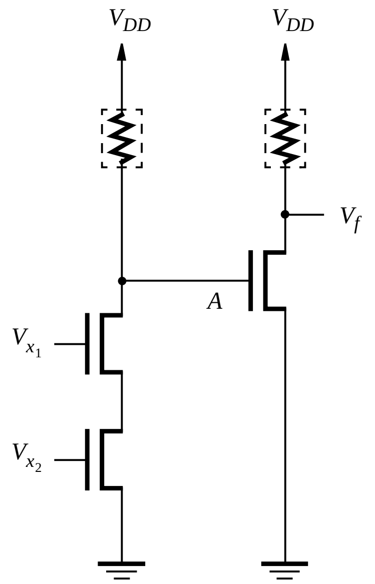
Figure 3.7 NMOS realization of a NOR gate.

Instead of the NAND and NOR gates just described, the reader would naturally be interested in the AND and OR gates that were used extensively in the previous chapter. Figure 3.8 indicates how an AND gate is built in NMOS technology by following a NAND gate with an inverter. Node A realizes the NAND of inputs x_1 and x_2, and f represents the AND function. In a similar fashion an OR gate is realized as a NOR gate followed by an inverter, as depicted in Figure 3.9.

3.3 CMOS Logic Gates

So far we have considered how to implement logic gates using NMOS transistors. For each of the circuits that has been presented, it is possible to derive an equivalent circuit that uses PMOS transistors. However, it is more interesting to consider how both NMOS and PMOS transistors can be used together. The most popular such approach is known as CMOS technology. We will see in section 3.8 that CMOS technology offers some attractive practical advantages in comparison to NMOS technology.

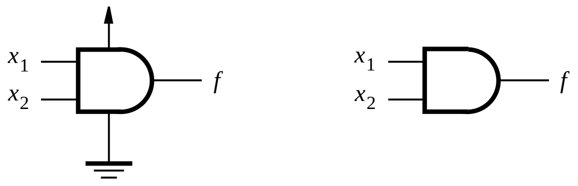
In NMOS circuits the logic functions are realized by arrangements of NMOS transistors, combined with a pull-up device that acts as a resistor. We will refer to the part of the circuit



(a) Circuit

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

(b) Truth table

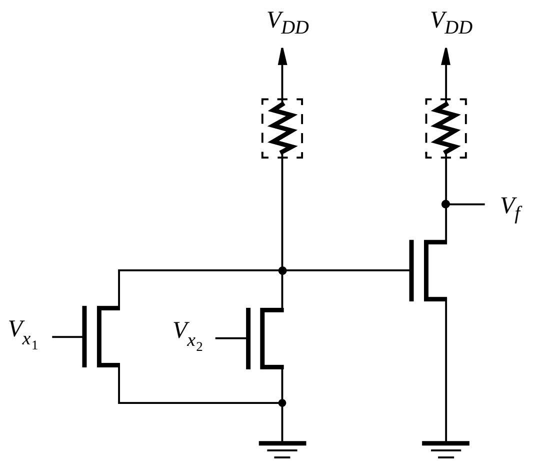


(c) Graphical symbols

Figure 3.8 NMOS realization of an AND gate.

that involves NMOS transistors as the *pull-down network (PDN)*. Then the structure of the circuits in Figures 3.5 through 3.9 can be characterized by the block diagram in Figure 3.10. The concept of CMOS circuits is based on replacing the pull-up device with a *pull-up network (PUN)* that is built using PMOS transistors, such that the functions realized by the PDN and PUN networks are complements of each other. Then a logic circuit, such as a typical logic gate, is implemented as indicated in Figure 3.11. For any given valuation of the input signals, either the PDN pulls V_f down to *Gnd* or the PUN pulls V_f up to V_{DD}. The PDN and the PUN have equal numbers of transistors, which are arranged so that the two networks are *duals* of one another. Wherever the PDN has NMOS transistors in series, the PUN has PMOS transistors in parallel, and vice versa.

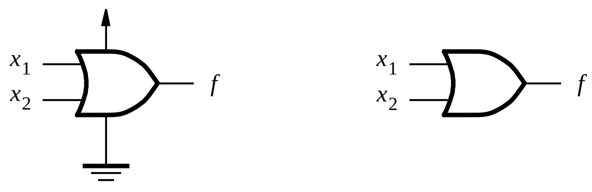
The simplest example of a CMOS circuit, a NOT gate, is shown in Figure 3.12. When $V_x = 0$ V, transistor T_2 is off and transistor T_1 is on. This makes $V_f = 5$ V, and since T_2 is



(a) Circuit

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

(b) Truth table



(c) Graphical symbols

Figure 3.9 NMOS realization of an OR gate.

off, no current flows through the transistors. When $V_x = 5$ V, T_2 is on and T_1 is off. Thus $V_f = 0$ V, and no current flows because T_1 is off.

A key point is that no current flows in a CMOS inverter when the input is either low or high. This is true for all CMOS circuits; no current flows, and hence no power is dissipated under steady state conditions. This property has led to CMOS becoming the most popular technology in use today for building logic circuits. We will discuss current flow and power dissipation in detail in section 3.8.

Figure 3.13 provides a circuit diagram of a CMOS NAND gate. It is similar to the NMOS circuit presented in Figure 3.6 except that the pull-up device has been replaced by the PUN with two PMOS transistors connected in parallel. The truth table in the figure specifies the state of each of the four transistors for each logic valuation of inputs x_1 and x_2. The reader can verify that the circuit properly implements the NAND function. Under static conditions no path exists for current flow from V_{DD} to *Gnd*.

The circuit in Figure 3.13 can be derived from the logic expression that defines the NAND operation, $f = \overline{x_1 x_2}$. This expression specifies the conditions for which $f = 1$; hence it defines the PUN. Since the PUN consists of PMOS transistors, which are turned on when their control (gate) inputs are set to 0, an input variable x_i turns on a transistor if

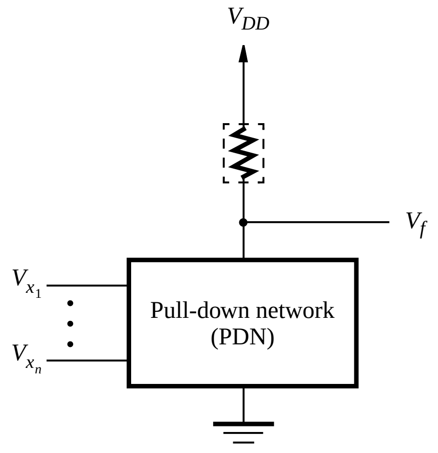


Figure 3.10 Structure of an NMOS circuit.

$x_i = 0$. From DeMorgan's law, we have

$$f = \overline{x_1 x_2} = \overline{x}_1 + \overline{x}_2$$

Thus $f = 1$ when *either* input x_1 or x_2 has the value 0, which means that the PUN must have two PMOS transistors connected in parallel. The PDN must implement the complement of f, which is

$$\overline{f} = x_1 x_2$$

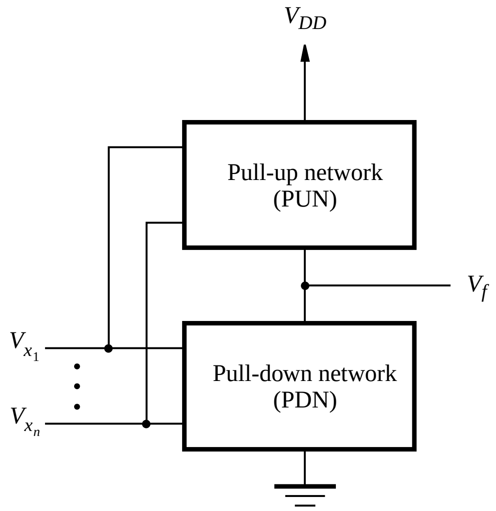
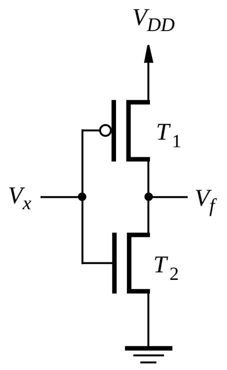


Figure 3.11 Structure of a CMOS circuit.



x	T_1	T_2	f
0	on	off	1
1	off	on	0

(a) Circuit (b) Truth table and transistor states

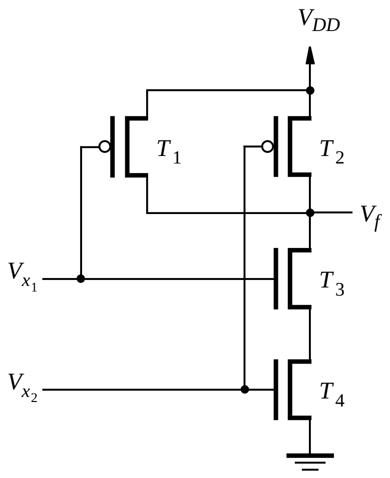
Figure 3.12 CMOS realization of a NOT gate.

Since $\overline{f} = 1$ when *both* x_1 and x_2 are 1, it follows that the PDN must have two NMOS transistors connected in series.

The circuit for a CMOS NOR gate is derived from the logic expression that defines the NOR operation

$$f = \overline{x_1 + x_2} = \overline{x}_1\overline{x}_2$$

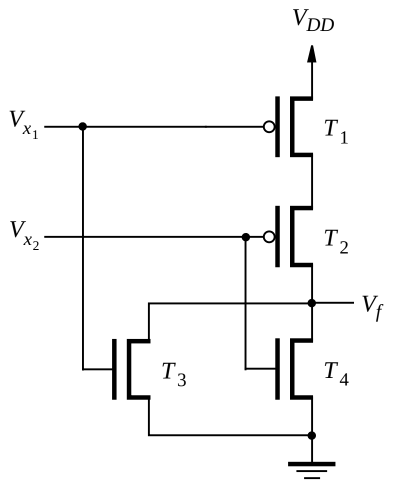
Since $f = 1$ only if both x_1 and x_2 have the value 0, then the PUN consists of two PMOS transistors connected in series. The PDN, which realizes $\overline{f} = x_1 + x_2$, has two NMOS transistors in parallel, leading to the circuit shown in Figure 3.14.



x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

(a) Circuit (b) Truth table and transistor states

Figure 3.13 CMOS realization of a NAND gate.



(a) Circuit

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

(b) Truth table and transistor states

Figure 3.14 CMOS realization of a NOR gate.

A CMOS AND gate is built by connecting a NAND gate to an inverter, as illustrated in Figure 3.15. Similarly, an OR gate is constructed with a NOR gate followed by a NOT gate.

The above procedure for deriving a CMOS circuit can be applied to more general logic functions to create *complex gates*. This process is illustrated in the following two examples.

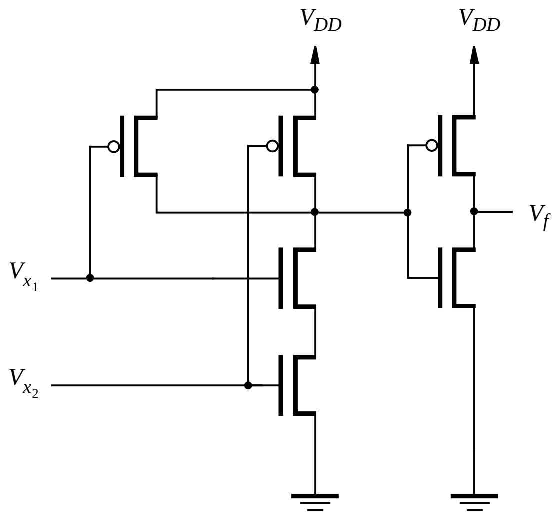


Figure 3.15 CMOS realization of an AND gate.

Example 3.1 Consider the function

$$f = \overline{x}_1 + \overline{x}_2\overline{x}_3$$

Since all variables appear in their complemented form, we can directly derive the PUN. It consists of a PMOS transistor controlled by x_1 in parallel with a series combination of PMOS transistors controlled by x_2 and x_3. For the PDN we have

$$\overline{f} = \overline{\overline{x}_1 + \overline{x}_2\overline{x}_3} = x_1(x_2 + x_3)$$

This expression gives the PDN that has an NMOS transistor controlled by x_1 in series with a parallel combination of NMOS transistors controlled by x_2 and x_3. The circuit is shown in Figure 3.16.

Example 3.2 Consider the function

$$f = \overline{x}_1 + (\overline{x}_2 + \overline{x}_3)\overline{x}_4$$

Then

$$\overline{f} = x_1(x_2x_3 + x_4)$$

These expressions lead directly to the circuit in Figure 3.17.

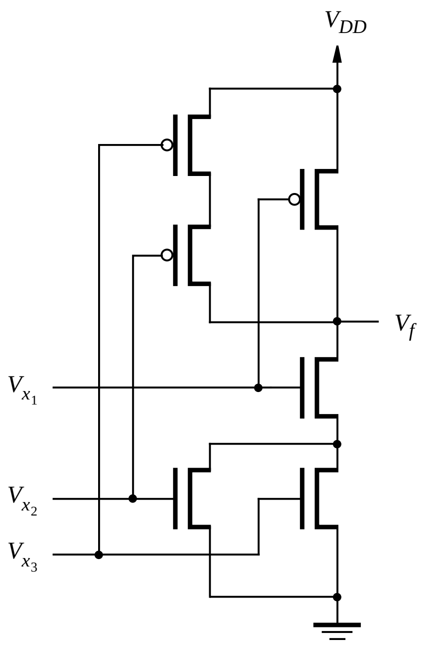


Figure 3.16 The circuit for Example 3.1.

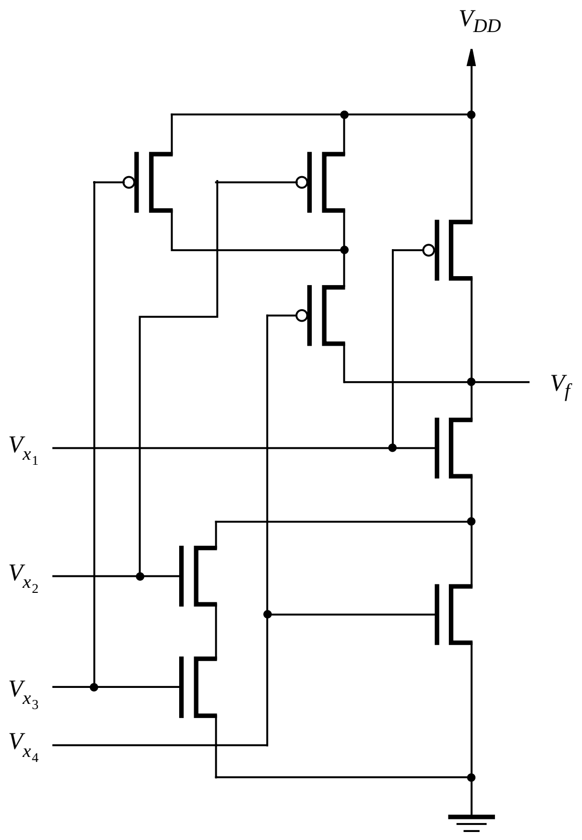


Figure 3.17 The circuit for Example 3.2.

The circuits in Figures 3.16 and 3.17 show that it is possible to implement fairly complex logic functions using combinations of series and parallel connections of transistors (acting as switches), without implementing each series or parallel connection as a complete AND (using the structure introduced in Figure 3.15) or OR gate.

3.3.1 Speed of Logic Gate Circuits

In the preceding sections we have assumed that transistors operate as ideal switches that present no resistance to current flow. Hence, while we have derived circuits that realize the functionality needed in logic gates, we have ignored the important issue of the speed of operation of the circuits. In reality transistor switches have a significant resistance when turned on. Also, transistor circuits include capacitors, which are created as a side effect of the manufacturing process. These factors affect the amount of time required for signal values to propagate through logic gates. We provide a detailed discussion of the speed of logic circuits, as well as a number of other practical issues, in section 3.8.

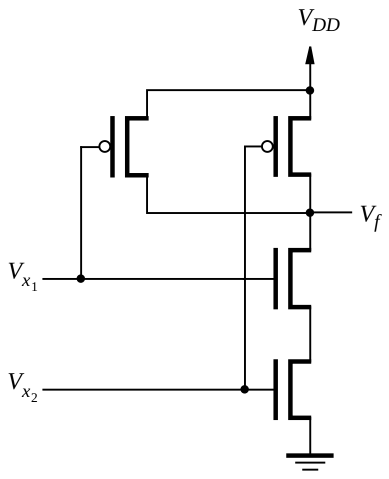
3.4 NEGATIVE LOGIC SYSTEM

At the beginning of this chapter, we said that logic values are represented as two distinct ranges of voltage levels. We are using the convention that the higher voltage levels represent logic value 1 and the lower voltages represent logic value 0. This convention is known as the positive logic system, and it is the one used in most practical applications. In this section we briefly consider the negative logic system in which the association between voltage levels and logic values is reversed.

Let us reconsider the CMOS circuit in Figure 3.13, which is reproduced in Figure 3.18*a*. Part (*b*) of the figure gives a truth table for the circuit, but the table shows voltage levels instead of logic values. In this table, L refers to the low voltage level in the circuit, which is 0 V, and H represents the high voltage level, which is V_{DD}. This is the style of truth table that manufacturers of integrated circuits often use in data sheets to describe the functionality of the chips. It is entirely up to the user of the chip as to whether L and H are interpreted in terms of logic values such that $L = 0$ and $H = 1$, or $L = 1$ and $H = 0$.

Figure 3.19*a* illustrates the positive logic interpretation in which $L = 0$ and $H = 1$. As we already know from the discussions of Figure 3.13, the circuit represents a NAND gate under this interpretation. The opposite interpretation is shown in Figure 3.19*b*. Here negative logic is used so that $L = 1$ and $H = 0$. The truth table specifies that the circuit represents a NOR gate in this case. Note that the truth table rows are listed in the opposite order from what we normally use, to be consistent with the L and H values in Figure 3.18*b*. Figure 3.19*b* also gives the logic gate symbol for the NOR gate, which includes small triangles on the gate's terminals to indicate that the negative logic system is used.

As another example, consider again the circuit in Figure 3.15. Its truth table, in terms of voltage levels, is given in Figure 3.20*a*. Using the positive logic system, this circuit



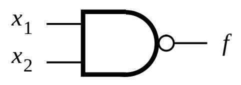
(a) Circuit

V_{x_1}	V_{x_2}	V_f
L	L	H
L	H	H
H	L	H
H	H	L

(b) Voltage levels

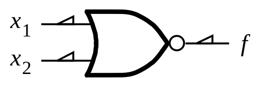
Figure 3.18 Voltage levels in the circuit in Figure 3.13.

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0



(a) Positive logic truth table and gate symbol

x_1	x_2	f
1	1	0
1	0	0
0	1	0
0	0	1



(b) Negative logic truth table and gate symbol

Figure 3.19 Interpretation of the circuit in Figure 3.18.

represents an AND gate, as indicated in Figure 3.20*b*. But using the negative logic system, the circuit represents an OR gate, as depicted in Figure 3.20*c*.

It is possible to use a mixture of positive and negative logic in a single circuit, which is known as a *mixed logic system*. In practice, the positive logic system is used in most applications. We will not consider the negative logic system further in this book.

3.5 STANDARD CHIPS

In Chapter 1 we mentioned that several different types of integrated circuit chips are available for implementation of logic circuits. We now discuss the available choices in some detail.

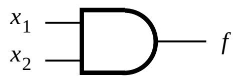
3.5.1 7400-SERIES STANDARD CHIPS

An approach used widely until the mid-1980s was to connect together multiple chips, each containing only a few logic gates. A wide assortment of chips, with different types of logic gates, is available for this purpose. They are known as 7400-series parts because the chip part numbers always begin with the digits 74. An example of a 7400-series part is given in Figure 3.21. Part (*a*) of the figure shows a type of package that the chip is provided in, called a *dual-inline package (DIP)*. Part (*b*) illustrates the 7404 chip, which comprises six NOT gates. The chip's external connections are called *pins* or *leads*. Two pins are used to connect to V_{DD} and *Gnd*, and other pins provide connections to the NOT gates. Many

V_{x_1}	V_{x_2}	V_f
L	L	L
L	H	L
H	L	L
H	H	H

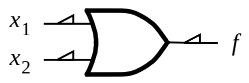
(a) Voltage levels

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1



(b) Positive logic

x_1	x_2	f
1	1	1
1	0	1
0	1	1
0	0	0



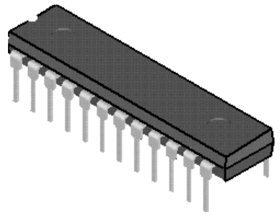
(c) Negative logic

Figure 3.20 Interpretation of the circuit in Figure 3.15.

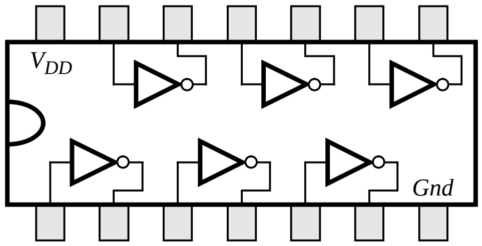
7400-series chips exist, and they are described in the data books produced by manufacturers of these chips [3–7]. Diagrams of some of the chips are also included in several textbooks, such as [8–12].

The 7400-series chips are produced in standard forms by a number of integrated circuit manufacturers, using agreed-upon specifications. Competition among various manufacturers works to the designer's advantage because it tends to lower the price of chips and ensures that parts are always readily available. For each specific 7400-series chip, several variants are built with different technologies. For instance, the part called 74LS00 is built with a technology called transistor-transistor logic (TTL), which is described in Appendix E, whereas the 74HC00 is fabricated using CMOS technology. In general, the most popular chips used today are the CMOS variants.

As an example of how a logic circuit can be implemented using 7400-series chips, consider the function $f = x_1x_2 + \overline{x}_2x_3$, which is shown in the form of a logic diagram in



(a) Dual-inline package



(b) Structure of 7404 chip

Figure 3.21 A 7400-series chip.

Figure 2.26. A NOT gate is required to produce $\overline{x}_2$, as well as 2 two-input AND gates and a two-input OR gate. Figure 3.22 shows three 7400-series chips that can be used to implement the function. We assume that the three input signals x_1, x_2, and x_3 are produced as outputs by some other circuitry that can be connected by wires to the three chips. Notice that power and ground connections are included for all three chips. This example makes use of only a portion of the gates available on the three chips, hence the remaining gates can be used to realize other functions.

Because of their low logic capacity, the standard chips are seldom used in practice today, with one exception. Many modern products include standard chips that contain buffers. Buffers are logic gates that are usually used to improve the speed of circuits. An example of a buffer chip is depicted in Figure 3.23. It is the 74244 chip, which comprises eight *tri-state buffers*. We describe how tri-state buffers work in section 3.8.8. Rather than showing how the buffers are arranged inside the chip package, as we did for the NOT gates in Figure 3.21, we show only the pin numbers of the package pins that are connected to the buffers. The package has 20 pins, and they are numbered in the same manner as shown for Figure 3.21; *Gnd* and V_{DD} connections are provided on pins 10 and 20, respectively. Many other buffer chips also exist. For example, the 162244 chip has 16 tri-state buffers. It is part of a family of devices that are similar to the 7400-series chips but with twice as many gates in each chip. These chips are available in multiple types of packages, with the most popular being a *small-outline integrated circuit (SOIC)* package. An SOIC package has a similar shape to a DIP, but the SOIC is considerably smaller in physical size.

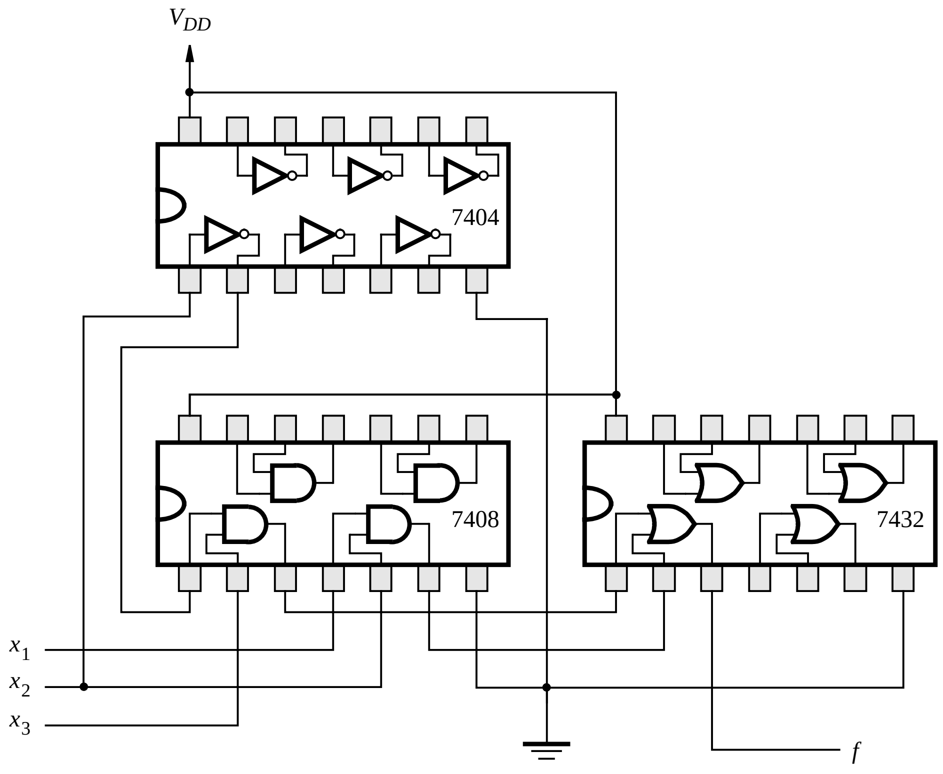


Figure 3.22 An implementation of $f = x_1x_2 + \overline{x}_2x_3$.

As integrated circuit technology has improved over time, a system of classifying chips according to their size has evolved. The earliest chips produced, such as the 7400-series chips, comprise only a few logic gates. The technology used to produce these chips is referred to as *small-scale integration (SSI)*. Chips that include slightly more logic circuitry, typically about 10 to 100 gates, represent *medium-scale integration (MSI)*. Until the mid-1980s chips that were too large to qualify as MSI were classified as *large-scale integration*

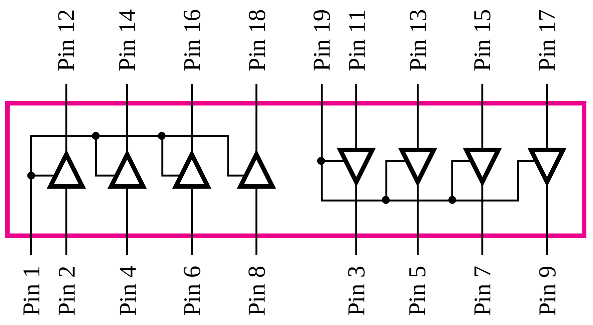


Figure 3.23 The 74244 buffer chip.

(LSI). In recent years the concept of classifying circuits according to their size has become of little practical use. Most integrated circuits today contain many thousands or millions of transistors. Regardless of their exact size, these large chips are said to be made with *very large scale integration (VLSI)* technology. The trend in digital hardware products is to integrate as much circuitry as possible onto a single chip. Thus most of the chips used today are built with VLSI technology, and the older types of chips are used rarely.

3.6 Programmable Logic Devices

The function provided by each of the 7400-series parts is fixed and cannot be tailored to suit a particular design situation. This fact, coupled with the limitation that each chip contains only a few logic gates, makes these chips inefficient for building large logic circuits. It is possible to manufacture chips that contain relatively large amounts of logic circuitry with a structure that is not fixed. Such chips were first introduced in the 1970s and are called *programmable logic devices (PLDs)*.

A PLD is a general-purpose chip for implementing logic circuitry. It contains a collection of logic circuit elements that can be customized in different ways. A PLD can be viewed as a "black box" that contains logic gates and programmable switches, as illustrated in Figure 3.24. The programmable switches allow the logic gates inside the PLD to be connected together to implement whatever logic circuit is needed.

3.6.1 Programmable Logic Array (PLA)

Several types of PLDs are commercially available. The first developed was the *programmable logic array (PLA)*. The general structure of a PLA is depicted in Figure 3.25. Based on the idea that logic functions can be realized in sum-of-products form, a PLA

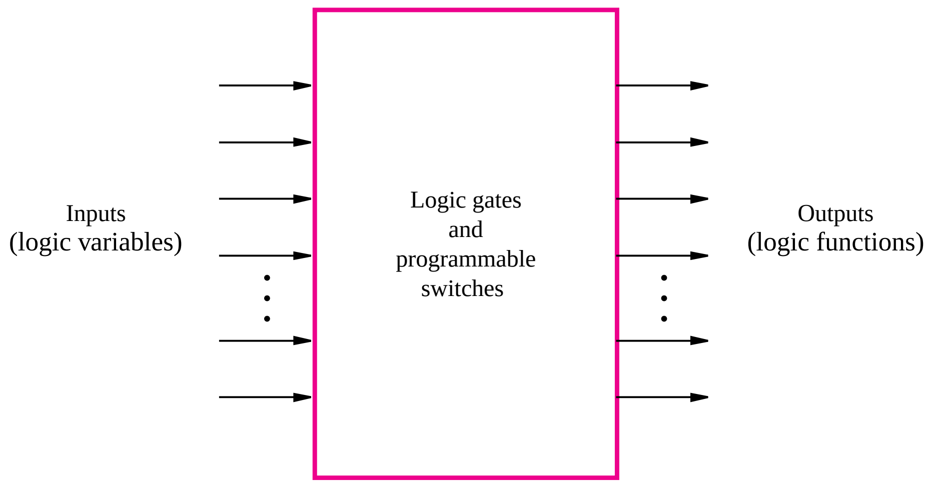


Figure 3.24 Programmable logic device as a black box.

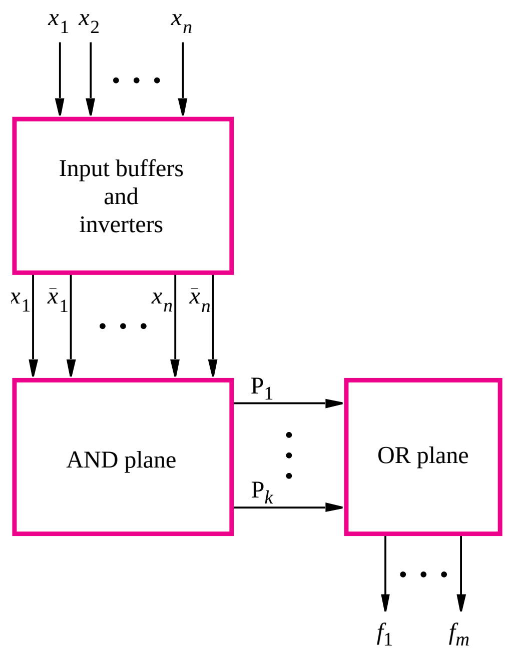

x_1 x_2 x_n
Input buffers and inverters
x_1 $\bar{x}_1$ x_n $\bar{x}_n$
AND plane
P_1
P_k
OR plane
f_1 f_m

Figure 3.25 General structure of a PLA.

comprises a collection of AND gates that feeds a set of OR gates. As shown in the figure, the PLA's inputs $x_1, \ldots, x_n$ pass through a set of buffers (which provide both the true value and complement of each input) into a circuit block called an *AND plane*, or *AND array*. The AND plane produces a set of product terms $P_1, \ldots, P_k$. Each of these terms can be configured to implement any AND function of $x_1, \ldots, x_n$. The product terms serve as the inputs to an *OR plane*, which produces the outputs $f_1, \ldots, f_m$. Each output can be configured to realize any sum of $P_1, \ldots, P_k$ and hence any sum-of-products function of the PLA inputs.

A more detailed diagram of a small PLA is given in Figure 3.26, which shows a PLA with three inputs, four product terms, and two outputs. Each AND gate in the AND plane has six inputs, corresponding to the true and complemented versions of the three input signals. Each connection to an AND gate is programmable; a signal that is connected to an AND gate is indicated with a wavy line, and a signal that is not connected to the gate is shown with a broken line. The circuitry is designed such that any unconnected AND-gate inputs do not affect the output of the AND gate. In commercially available PLAs, several methods of realizing the programmable connections exist. Detailed explanation of how a PLA can be built using transistors is given in section 3.10.

In Figure 3.26 the AND gate that produces P_1 is shown connected to the inputs x_1 and x_2. Hence $P_1 = x_1x_2$. Similarly, $P_2 = x_1\bar{x}_3$, $P_3 = \bar{x}_1\bar{x}_2x_3$, and $P_4 = x_1x_3$. Programmable connections also exist for the OR plane. Output f_1 is connected to product terms P_1, P_2, and P_3. It therefore realizes the function $f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$. Similarly, output

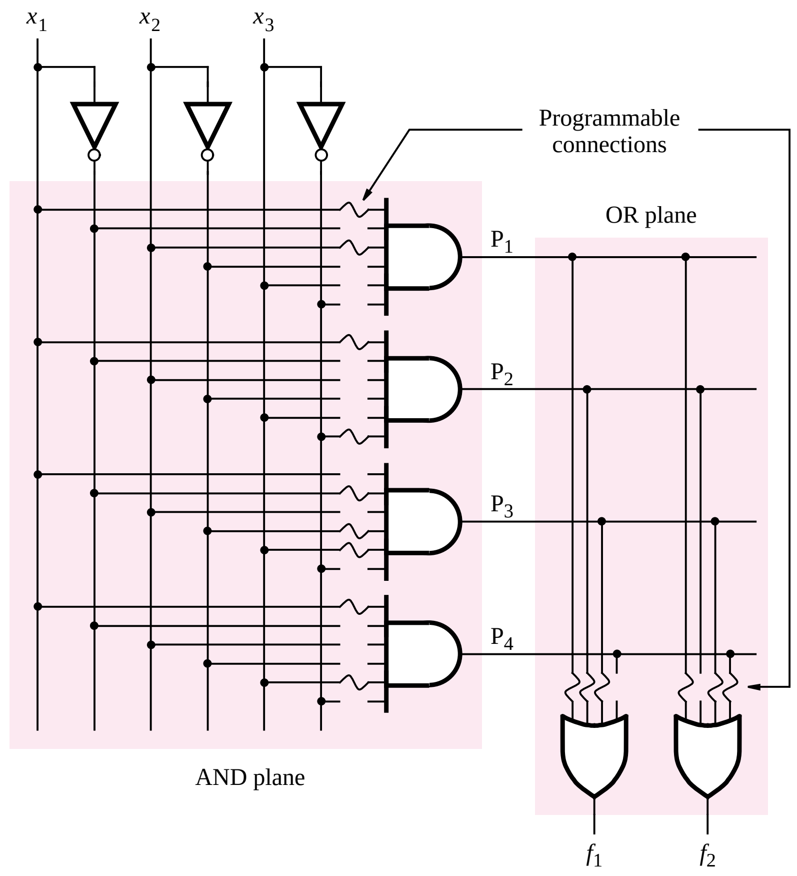


Figure 3.26 Gate-level diagram of a PLA.

$f_2 = x_1x_2 + \overline{x}_1\overline{x}_2x_3 + x_1x_3$. Although Figure 3.26 depicts the PLA programmed to implement the functions described above, by programming the AND and OR planes differently, each of the outputs f_1 and f_2 could implement various functions of x_1, x_2, and x_3. The only constraint on the functions that can be implemented is the size of the AND plane because it produces only four product terms. Commercially available PLAs come in larger sizes than we have shown here. Typical parameters are 16 inputs, 32 product terms, and eight outputs.

Although Figure 3.26 illustrates clearly the functional structure of a PLA, this style of drawing is awkward for larger chips. Instead, it has become customary in technical literature to use the style shown in Figure 3.27. Each AND gate is depicted as a single horizontal line attached to an AND-gate symbol. The possible inputs to the AND gate are drawn as vertical lines that cross the horizontal line. At any crossing of a vertical and horizontal line, a programmable connection, indicated by an X, can be made. Figure 3.27 shows the programmable connections needed to implement the product terms in Figure 3.26. Each OR gate is drawn in a similar manner, with a vertical line attached to an OR-gate symbol.

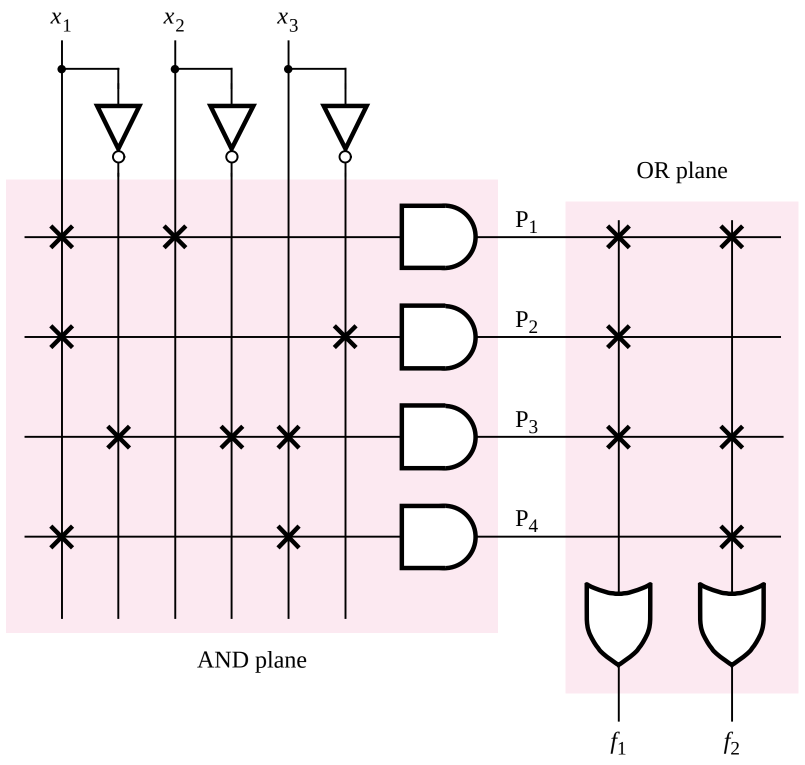


Figure 3.27 Customary schematic for the PLA in Figure 3.26.

The AND-gate outputs cross these lines, and corresponding programmable connections can be formed. The figure illustrates the programmable connections that produce the functions f_1 and f_2 from Figure 3.26.

The PLA is efficient in terms of the area needed for its implementation on an integrated circuit chip. For this reason, PLAs are often included as part of larger chips, such as microprocessors. In this case a PLA is created so that the connections to the AND and OR gates are fixed, rather than programmable. In section 3.10 we will show that both fixed and programmable PLAs can be created with similar structures.

3.6.2 Programmable Array Logic (PAL)

In a PLA both the AND and OR planes are programmable. Historically, the programmable switches presented two difficulties for manufacturers of these devices: they were hard to fabricate correctly, and they reduced the speed-performance of circuits implemented in the PLAs. These drawbacks led to the development of a similar device in which the AND plane is programmable, but the OR plane is fixed. Such a chip is known as a *programmable array logic (PAL)* device. Because they are simpler to manufacture, and thus less expensive than PLAs, and offer better performance, PALs have become popular in practical applications.

An example of a PAL with three inputs, four product terms, and two outputs is given in Figure 3.28. The product terms P_1 and P_2 are hardwired to one OR gate, and P_3 and P_4 are hardwired to the other OR gate. The PAL is shown programmed to realize the two logic functions $f_1 = x_1x_2\overline{x}_3 + \overline{x}_1x_2x_3$ and $f_2 = \overline{x}_1\overline{x}_2 + x_1x_2x_3$. In comparison to the PLA in Figure 3.27, the PAL offers less flexibility; the PLA allows up to four product terms per OR gate, whereas the OR gates in the PAL have only two inputs. To compensate for the reduced flexibility, PALs are manufactured in a range of sizes, with various numbers of inputs and outputs, and different numbers of inputs to the OR gates. An example of a commercial PAL is given in Appendix E.

So far we have assumed that the OR gates in a PAL, as in a PLA, connect directly to the output pins of the chip. In many PALs extra circuitry is added at the output of each OR gate to provide additional flexibility. It is customary to use the term *macrocell* to refer to the OR gate combined with the extra circuitry. An example of the flexibility that may be provided in a macrocell is given in Figure 3.29. The symbol labeled *flip-flop* represents a memory element. It stores the value produced by the OR gate output at a particular point in time and can hold that value indefinitely. The flip-flop is controlled by the signal called *clock*. When *clock* makes a transition from logic value 0 to 1, the flip-flop stores the value at its *D* input at that time and this value appears at the flip-flop's Q output. Flip-flops are used for implementing many types of logic circuits, as we will show in Chapter 7.

In section 2.7.2 we discussed a 2-to-1 multiplexer circuit. It has two data inputs, a select input, and one output. The select input is used to choose one of the data inputs as

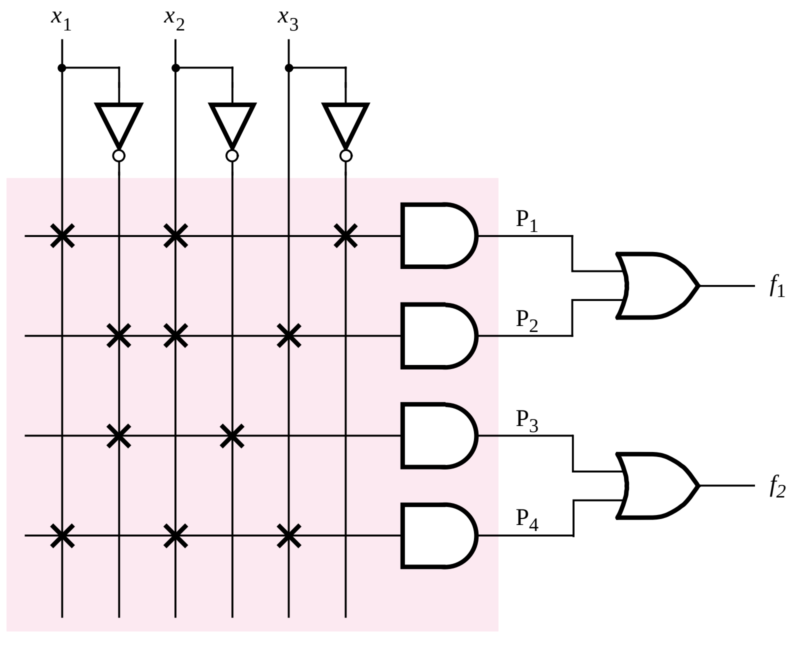


Figure 3.28 An example of a PAL.

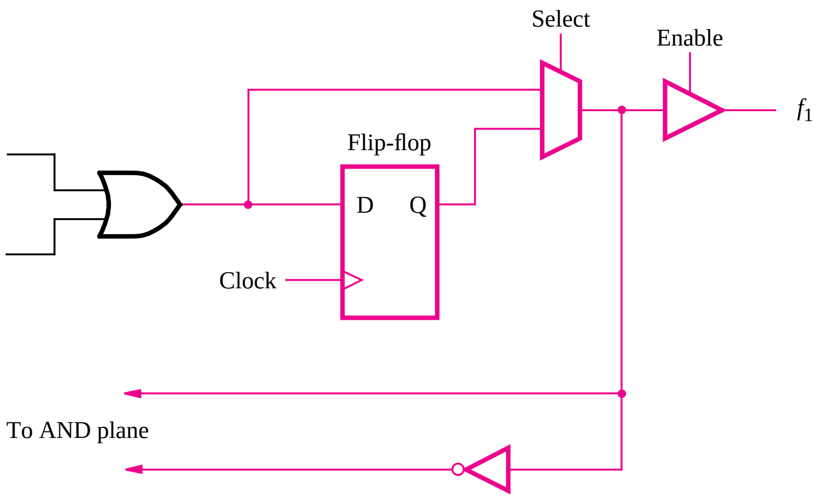


Figure 3.29 Extra circuitry added to OR-gate outputs from Figure 3.28.

the multiplexer's output. In Figure 3.29 a 2-to-1 multiplexer selects as an output from the PAL either the OR-gate output or the flip-flop output. The multiplexer's select line can be programmed to be either 0 or 1. Figure 3.29 shows another logic gate, called a tri-state buffer, connected between the multiplexer and the PAL output. We discuss tri-state buffers in section 3.8.8. Finally, the multiplexer's output is "fed back" to the AND plane in the PAL. This feedback connection allows the logic function produced by the multiplexer to be used internally in the PAL, which allows the implementation of circuits that have multiple stages, or levels, of logic gates.

A number of companies manufacture PLAs or PALs, or other, similar types of *simple PLDs (SPLDs)*. A partial list of companies, and the types of SPLDs that they manufacture, is given in Appendix E. An interested reader can examine the information that these companies provide on their products, which is available on the World Wide Web (WWW). The WWW locator for each company is given in Table E.1 in Appendix E.

3.6.3 Programming of PLAs and PALs

In Figures 3.27 and 3.28, each connection between a logic signal in a PLA or PAL and the AND/OR gates is shown as an X. We describe how these switches are implemented using transistors in section 3.10. Users' circuits are implemented in the devices by *configuring*, or *programming*, these switches. Commercial chips contain a few thousand programmable switches; hence it is not feasible for a user of these chips to specify manually the desired programming state of each switch. Instead, CAD systems are employed for this purpose. We introduced CAD tools in Chapter 2 and described methods for design entry and simulation of circuits. For CAD systems that support targeting of circuits to PLDs, the tools have the capability to automatically produce the necessary information for programming each of the switches in the device. A computer system that runs the CAD tools is connected by a cable to a dedicated *programming unit*. Once the user has completed the design of a circuit, the CAD tools generate a file, often called a *programming file* or *fuse map*, that specifies the state that each switch in the PLD should have, to realize correctly the designed circuit. The PLD is placed into the programming unit, and the programming file is transferred from the computer system. The programming unit then places the chip into a special *programming mode* and configures each switch individually. A photograph of a programming unit is shown in Figure 3.30. Several adaptors are shown beside the main unit; each adaptor is used for a specific type of chip package.

The programming procedure may take a few minutes to complete. Usually, the programming unit can automatically "read back" the state of each switch after programming, to verify that the chip has been programmed correctly. A detailed discussion of the process involved in using CAD tools to target designed circuits to programmable chips is given in Appendices B, C, and D.

PLAs or PALs used as part of a logic circuit usually reside with other chips on a printed circuit board (PCB). The procedure described above assumes that the chip can be removed from the circuit board for programming in the programming unit. Removal is made possible by using a socket on the PCB, as illustrated in Figure 3.31. Although PLAs and PALs are available in the DIP packages shown in Figure 3.21*a*, they are also available in another popular type of package, called a *plastic-leaded chip carrier (PLCC)*, which is depicted in Figure 3.31. On all four of its sides, the PLCC package has pins that "wrap around" the edges of the chip, rather than extending straight down as in the case of a DIP. The socket that houses the PLCC is attached by solder to the circuit board, and the PLCC is held in the socket by friction.

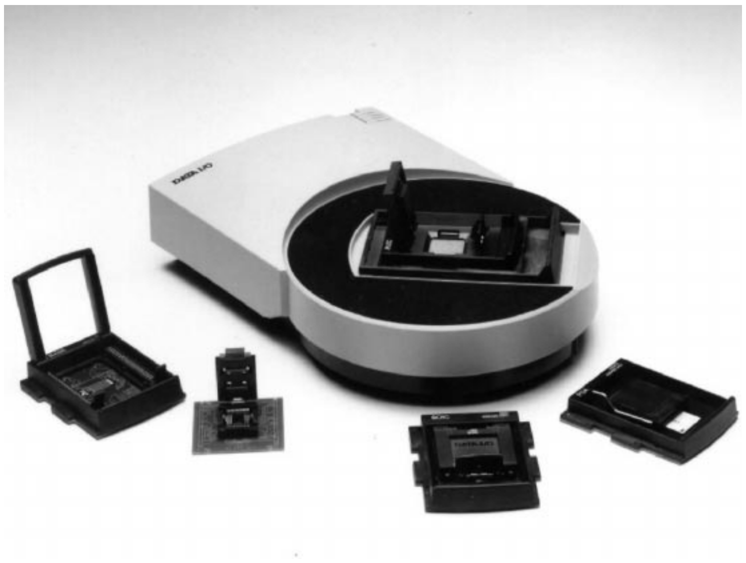


Figure 3.30 A PLD programming unit (courtesy of Data IO Corp.).

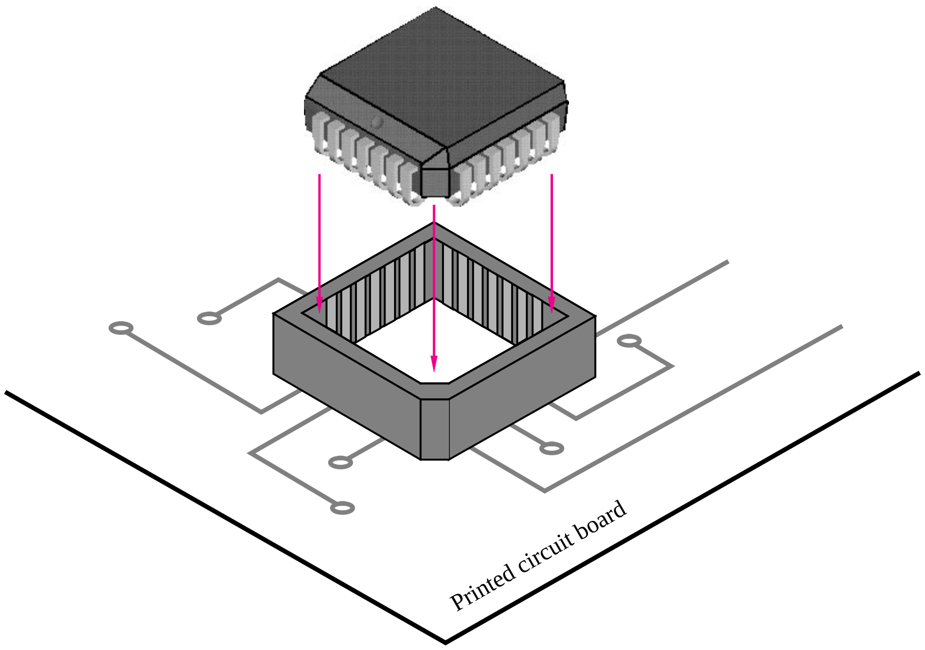


Figure 3.31 A PLCC package with socket.

Instead of relying on a programming unit to configure a chip, it would be advantageous to be able to perform the programming while the chip is still attached to its circuit board. This method of programming is called *in-system programming (ISP)*. It is not usually provided for PLAs or PALs, but is available for the more sophisticated chips that are described below.

3.6.4 Complex Programmable Logic Devices (CPLDs)

PLAs and PALs are useful for implementing a wide variety of small digital circuits. Each device can be used to implement circuits that do not require more than the number of inputs, product terms, and outputs that are provided in the particular chip. These chips are limited to fairly modest sizes, typically supporting a combined number of inputs plus outputs of not more than 32. For implementation of circuits that require more inputs and outputs, either multiple PLAs or PALs can be employed or else a more sophisticated type of chip, called a *complex programmable logic device (CPLD)*, can be used.

A CPLD comprises multiple circuit blocks on a single chip, with internal wiring resources to connect the circuit blocks. Each circuit block is similar to a PLA or a PAL; we will refer to the circuit blocks as *PAL-like blocks*. An example of a CPLD is given in Figure 3.32. It includes four PAL-like blocks that are connected to a set of *interconnection wires*. Each PAL-like block is also connected to a subcircuit labeled *I/O block*, which is attached to a number of the chip's input and output pins.

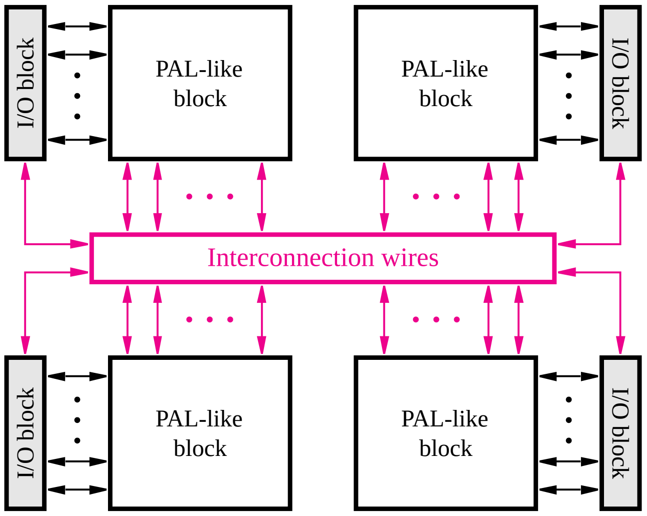


Figure 3.32 Structure of a complex programmable logic device (CPLD).

Figure 3.33 shows an example of the wiring structure and the connections to a PAL-like block in a CPLD. The PAL-like block includes 3 macrocells (real CPLDs typically have about 16 macrocells in a PAL-like block), each consisting of a four-input OR gate (real CPLDs usually provide between 5 and 20 inputs to each OR gate). The OR-gate output is connected to another type of logic gate that we have not yet introduced. It is called an Exclusive-OR (XOR) gate. We discuss XOR gates in section 3.9.1. The behavior of an XOR gate is the same as for an OR gate except that if both of the inputs are 1, the XOR gate produces a 0. One input to the XOR gate in Figure 3.33 can be programmably connected to 1 or 0; if 1, then the XOR gate complements the OR-gate output, and if 0, then the XOR gate has no effect. In many CPLDs the XOR gates can be used in other ways also, which we will see in Example 4.19, in Chapter 4. The macrocell also includes a flip-flop, a multiplexer, and a tri-state buffer. As we mentioned in the discussion for Figure 3.29, the flip-flop is used to store the output value produced by the OR gate. Each tri-state buffer (see section 3.8.8) is connected to a pin on the CPLD package. The tri-state buffer acts as a switch that allows each pin to be used either as an output from the CPLD or as an input. To use a pin as an output, the corresponding tri-state buffer is enabled, acting as a switch that is turned on. If the pin is to be used as an input, then the tri-state buffer is disabled, acting as a switch that is turned off. In this case an external source can drive a signal onto the pin, which can be connected to other macrocells using the interconnection wiring.

The interconnection wiring contains programmable switches that are used to connect the PAL-like blocks. Each of the horizontal wires can be connected to some of the vertical wires that it crosses, but not to all of them. Extensive research has been done to decide how many switches should be provided for connections between the wires. The number of switches is chosen to provide sufficient flexibility for typical circuits without wasting

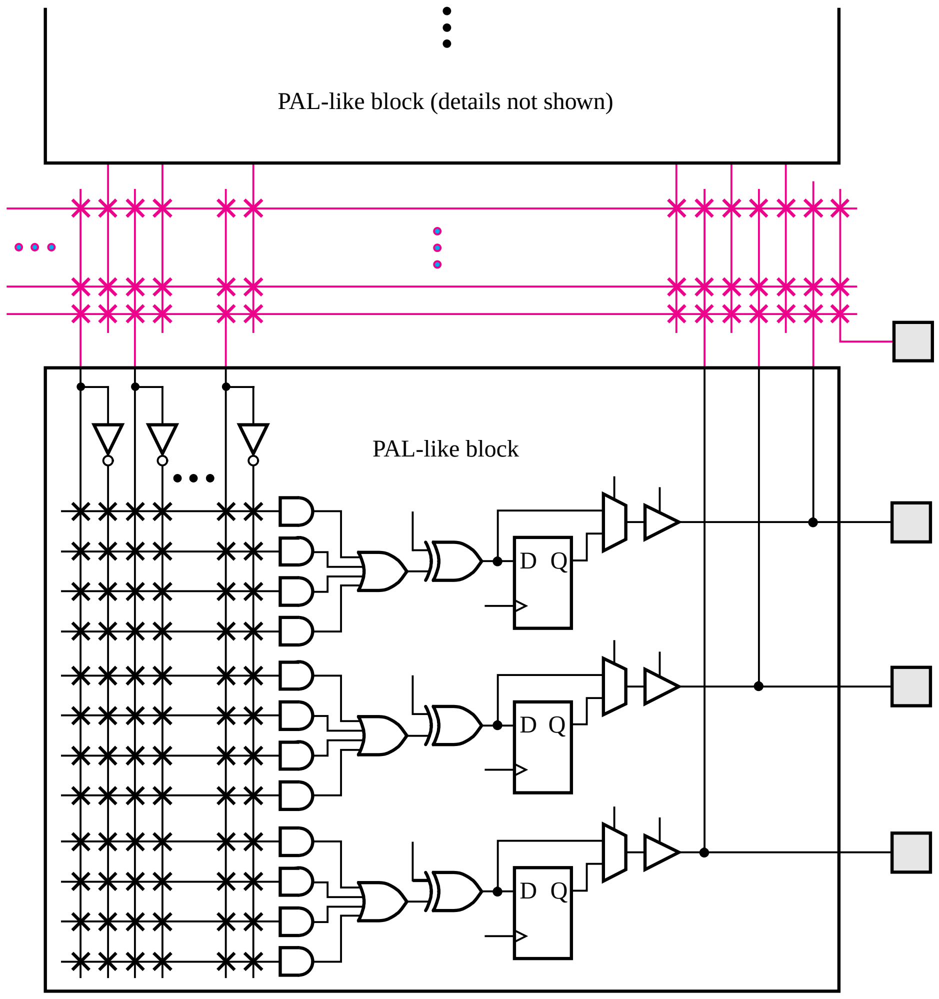
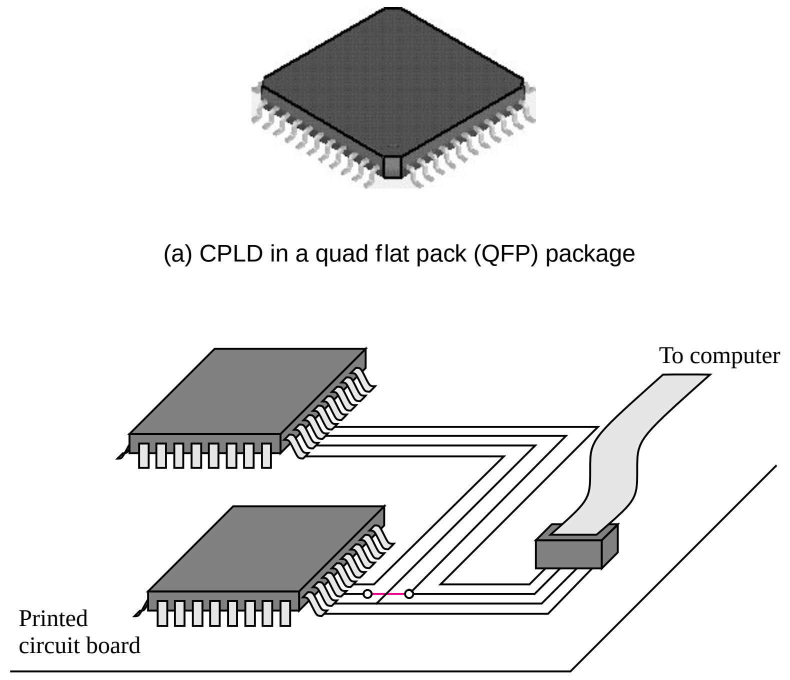


Figure 3.33 A section of the CPLD in Figure 3.32.

many switches in practice. One detail to note is that when a pin is used as an input, the macrocell associated with that pin cannot be used and is therefore wasted. Some CPLDs include additional connections between the macrocells and the interconnection wiring that avoids wasting macrocells in such situations.

Commercial CPLDs range in size from only 2 PAL-like blocks to more than 100 PAL-like blocks. They are available in a variety of packages, including the PLCC package that is shown in Figure 3.31. Figure 3.34*a* shows another type of package used to house CPLD chips, called a *quad flat pack (QFP)*. Like a PLCC package, the QFP package has pins on all four sides, but whereas the PLCC's pins wrap around the edges of the package, the QFP's pins extend outward from the package, with a downward-curving shape. The QFP's pins



(a) CPLD in a quad flat pack (QFP) package

(b) JTAG programming

Figure 3.34 CPLD packaging and programming.

are much thinner than those on a PLCC, which means that the package can support a larger number of pins; QFPs are available with more than 200 pins, whereas PLCCs are limited to fewer than 100 pins.

Most CPLDs contain the same type of programmable switches that are used in SPLDs, which are described in section 3.10. Programming of the switches may be accomplished using the same technique described in section 3.6.3, in which the chip is placed into a special-purpose programming unit. However, this programming method is rather inconvenient for large CPLDs for two reasons. First, large CPLDs may have more than 200 pins on the chip package, and these pins are often fragile and easily bent. Second, to be programmed in a programming unit, a socket is required to hold the chip. Sockets for large QFP packages are very expensive; they sometimes cost more than the CPLD device itself. For these reasons, CPLD devices usually support the ISP technique. A small connector is included on the PCB that houses the CPLD, and a cable is connected between that connector and a computer system. The CPLD is programmed by transferring the programming information generated by a CAD system through the cable, from the computer into the CPLD. The circuitry on the CPLD that allows this type of programming has been standardized by the IEEE and is usually called a *JTAG port*. It uses four wires to transfer information between the computer and the device being programmed. The term *JTAG* stands for Joint Test Action

Group. Figure 3.34*b* illustrates the use of a JTAG port for programming two CPLDs on a circuit board. The CPLDs are connected together so that both can be programmed using the same connection to the computer system. Once a CPLD is programmed, it retains the programmed state permanently, even when the power supply for the chip is turned off. This property is called *nonvolatile* programming.

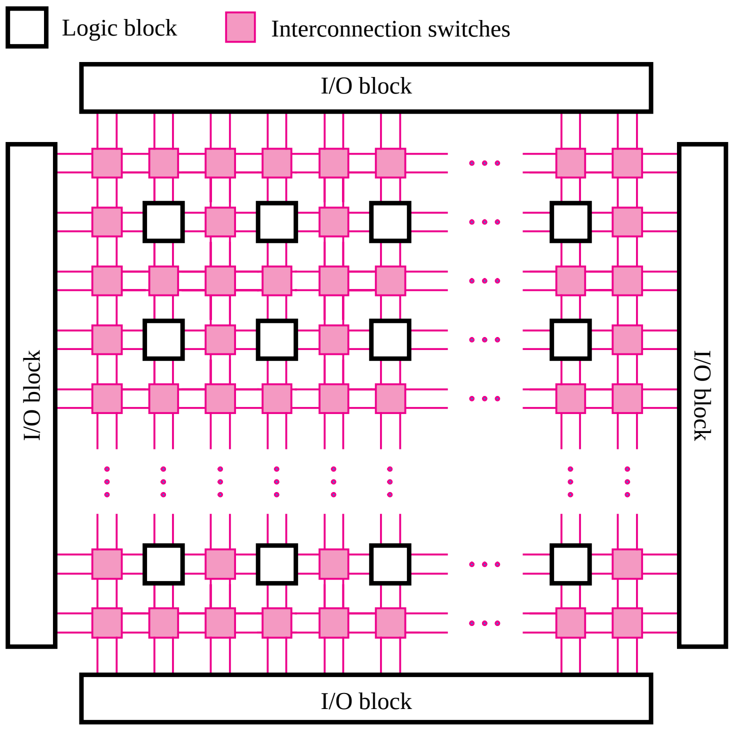
CPLDs are used for the implementation of many types of digital circuits. In industrial designs that employ some type of PLD device, CPLDs are used in about half the cases (SPLDs are used in only a small fraction of recently produced designs). A number of companies offer competing CPLDs. Appendix E lists, in Table E.2, the names of the major companies involved and shows the company's WWW locator. The reader is encouraged to examine the product information that each company provides on its Web pages. One example of a commercially available CPLD is described in detail in Appendix E. This CPLD family, manufactured by Altera and called the MAX 7000, is used in several examples presented later in the book.

3.6.5 Field-Programmable Gate Arrays

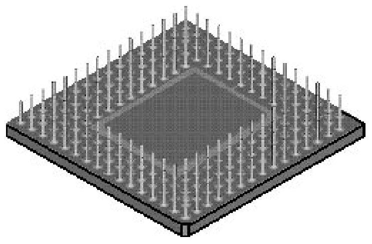
The types of chips described above, 7400 series, SPLDs, and CPLDs, are useful for implementation of a wide range of logic circuits. Except for CPLDs, these devices are rather small and are suitable only for relatively simple applications. Even for CPLDs, only moderately large logic circuits can be accommodated in a single chip. For cost and performance reasons, it is prudent to implement a desired logic circuit using as few chips as possible, so the amount of circuitry on a given chip and its functional capability are important. One way to quantify a circuit's *size* is to assume that the circuit is to be built using only simple logic gates and then estimate how many of these gates are needed. A commonly used measure is the total number of two-input NAND gates that would be needed to build the circuit; this measure is often called the number of *equivalent gates*.

Using the equivalent-gates metric, the size of a 7400-series chip is simple to measure because each chip contains only simple gates. For SPLDs and CPLDs the typical measure used is that each macrocell represents about 20 equivalent gates. Thus a typical PAL that has eight macrocells can accommodate a circuit that needs up to about 160 gates, and a large CPLD that has 1000 macrocells can implement circuits of up to about 20,000 equivalent gates.

By modern standards, a logic circuit with 20,000 gates is not large. To implement larger circuits, it is convenient to use a different type of chip that has a larger logic capacity. A *field-programmable gate array (FPGA)* is a programmable logic device that supports implementation of relatively large logic circuits. FPGAs are quite different from SPLDs and CPLDs because FPGAs do not contain AND or OR planes. Instead, FPGAs provide *logic blocks* for implementation of the required functions. The general structure of an FPGA is illustrated in Figure 3.35*a*. It contains three main types of resources: logic blocks, I/O blocks for connecting to the pins of the package, and interconnection wires and switches. The logic blocks are arranged in a two-dimensional array, and the interconnection wires are organized as horizontal and vertical *routing channels* between rows and columns of logic blocks. The routing channels contain wires and programmable switches that allow the logic blocks to be interconnected in many ways. Figure 3.35*a* shows two locations for



(a) General structure of an FPGA



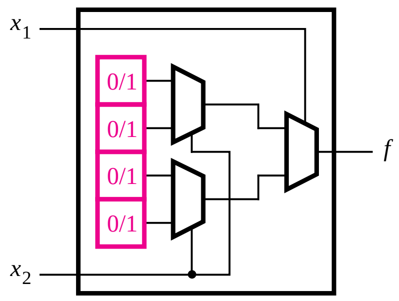
(b) Pin grid array (PGA) package (bottom view)

Figure 3.35 A field-programmable gate array (FPGA).

programmable switches; the blue boxes adjacent to logic blocks hold switches that connect the logic block input and output terminals to the interconnection wires, and the blue boxes that are diagonally between logic blocks connect one interconnection wire to another (such as a vertical wire to a horizontal wire). Programmable connections also exist between the I/O blocks and the interconnection wires. The actual number of programmable switches and wires in an FPGA varies in commercially available chips.

FPGAs can be used to implement logic circuits of more than a few hundred thousand equivalent gates in size. Two examples of FPGAs, called the Altera FLEX 10K and the Xilinx XC4000, are described in Appendix E. FPGAs are available in a variety of packages, including the PLCC and QFP packages described earlier. Figure 3.35*b* depicts another type of package, called a *pin grid array (PGA)*. A PGA package may have up to a few hundred pins in total, which extend straight outward from the bottom of the package, in a grid pattern. Yet another packaging technology that has emerged is known as the *ball grid array (BGA)*. The BGA is similar to the PGA except that the pins are small round balls, instead of posts. The advantage of BGA packages is that the pins are very small; hence more pins can be provided on the package.

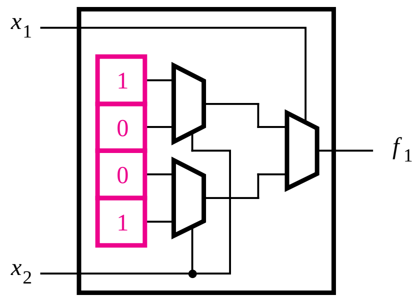
Each logic block in an FPGA typically has a small number of inputs and one output. A number of FPGA products are on the market, featuring different types of logic blocks. The most commonly used logic block is a *lookup table (LUT)*, which contains *storage cells* that are used to implement a small logic function. Each cell is capable of holding a single logic value, either 0 or 1. The stored value is produced as the output of the storage cell. LUTs of various *sizes* may be created, where the size is defined by the number of inputs. Figure 3.36*a* shows the structure of a small LUT. It has two inputs, x_1 and x_2, and one



(a) Circuit for a two-input LUT

x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

(b) $f_1 = \bar{x}_1\bar{x}_2 + x_1x_2$



(c) Storage cell contents in the LUT

Figure 3.36 A two-input lookup table (LUT).

output, f. It is capable of implementing any logic function of two variables. Because a two-variable truth table has four rows, this LUT has four storage cells. One cell corresponds to the output value in each row of the truth table. The input variables x_1 and x_2 are used as the select inputs of three multiplexers, which, depending on the valuation of x_1 and x_2, select the content of one of the four storage cells as the output of the LUT. We introduced multiplexers in section 2.7.2 and will discuss storage cells in Chapter 10.

To see how a logic function can be realized in the two-input LUT, consider the truth table in Figure 3.36*b*. The function f_1 from this table can be stored in the LUT as illustrated in Figure 3.36*c*. The arrangement of multiplexers in the LUT correctly realizes the function f_1. When $x_1 = x_2 = 0$, the output of the LUT is driven by the top storage cell, which represents the entry in the truth table for $x_1x_2 = 00$. Similarly, for all valuations of x_1 and x_2, the logic value stored in the storage cell corresponding to the entry in the truth table chosen by the particular valuation appears on the LUT output. Providing access to the contents of storage cells is only one way in which multiplexers can be used to implement logic functions. A detailed presentation of the applications of multiplexers is given in Chapter 6.

Figure 3.37 shows a three-input LUT. It has eight storage cells because a three-variable truth table has eight rows. In commercial FPGA chips, LUTs usually have either four or five inputs, which require 16 and 32 storage cells, respectively. In Figure 3.29 we showed that PALs usually have extra circuitry included with their AND-OR gates. The same is true for FPGAs, which usually have extra circuitry, besides a LUT, in each logic block. Figure 3.38 shows how a flip-flop may be included in an FPGA logic block. As discussed for Figure 3.29, the flip-flop is used to store the value of its *D* input under control of its *clock* input. Examples of logic blocks in commercial FPGAs are presented in Appendix E.

For a logic circuit to be realized in an FPGA, each logic function in the circuit must be small enough to fit within a single logic block. In practice, a user's circuit is automatically

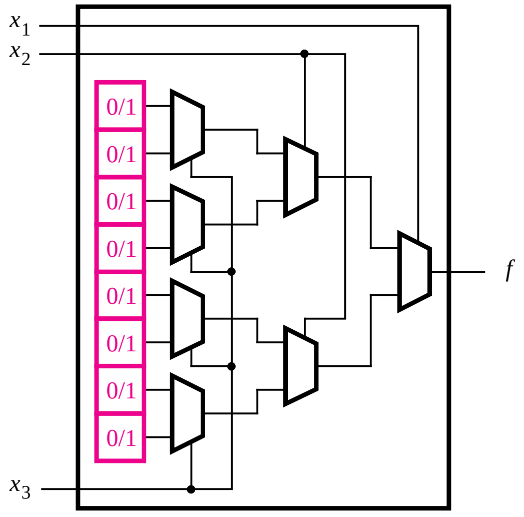


Figure 3.37 A three-input LUT.

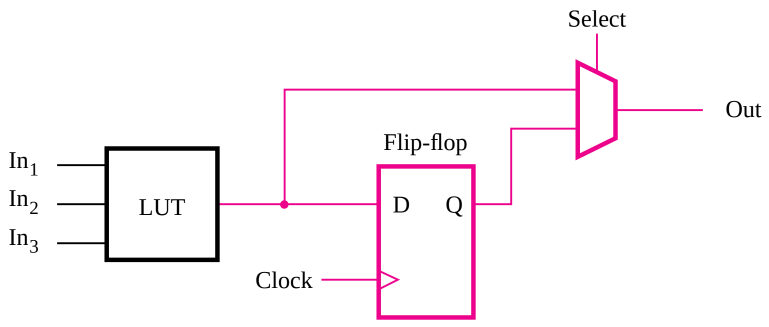


Figure 3.38 Inclusion of a flip-flop in an FPGA logic block.

translated into the required form by using CAD tools (see section 4.12). When a circuit is implemented in an FPGA, the logic blocks are programmed to realize the necessary functions and the routing channels are programmed to make the required interconnections between logic blocks. FPGAs are configured by using the ISP method, which we explained in section 3.6.4. The storage cells in the LUTs in an FPGA are *volatile*, which means that they lose their stored contents whenever the power supply for the chip is turned off. Hence the FPGA has to be programmed every time power is applied. Often a small memory chip that holds its data permanently, called a *programmable read-only memory (PROM),* is included on the circuit board that houses the FPGA. The storage cells in the FPGA are loaded automatically from the PROM when power is applied to the chips.

A small FPGA that has been programmed to implement a circuit is depicted in Figure 3.39. The FPGA has two-input LUTs, and there are four wires in each routing channel. The figure shows the programmed states of both the logic blocks and wiring switches in a section of the FPGA. Programmable wiring switches are indicated by an X. Each switch shown in blue is turned on and makes a connection between a horizontal and vertical wire. The switches shown in black are turned off. We describe how the switches are implemented by using transistors in section 3.10.1. The truth tables programmed into the logic blocks in the top row of the FPGA correspond to the functions $f_1 = x_1x_2$ and $f_2 = \overline{x}_2x_3$. The logic block in the bottom right of the figure is programmed to produce $f = f_1 + f_2 = x_1x_2 + \overline{x}_2x_3$.

3.6.6 Using CAD Tools to Implement Circuits in CPLDs and FPGAs

In section 2.8 we suggested that the reader should work through Tutorial 1, in Appendix B, to gain some experience using real CAD tools. Tutorial 1 covers the steps of design entry and functional simulation. Now that we have discussed some of the details of the implementation of circuits in chips, the reader may wish to experiment further with the CAD tools. In Tutorial 2, section C.3, we illustrate how to download a circuit from a computer into a CPLD or FPGA.

April 5, 1999 14:26 g02-ch3 Sheet number 37 Page number 97 black

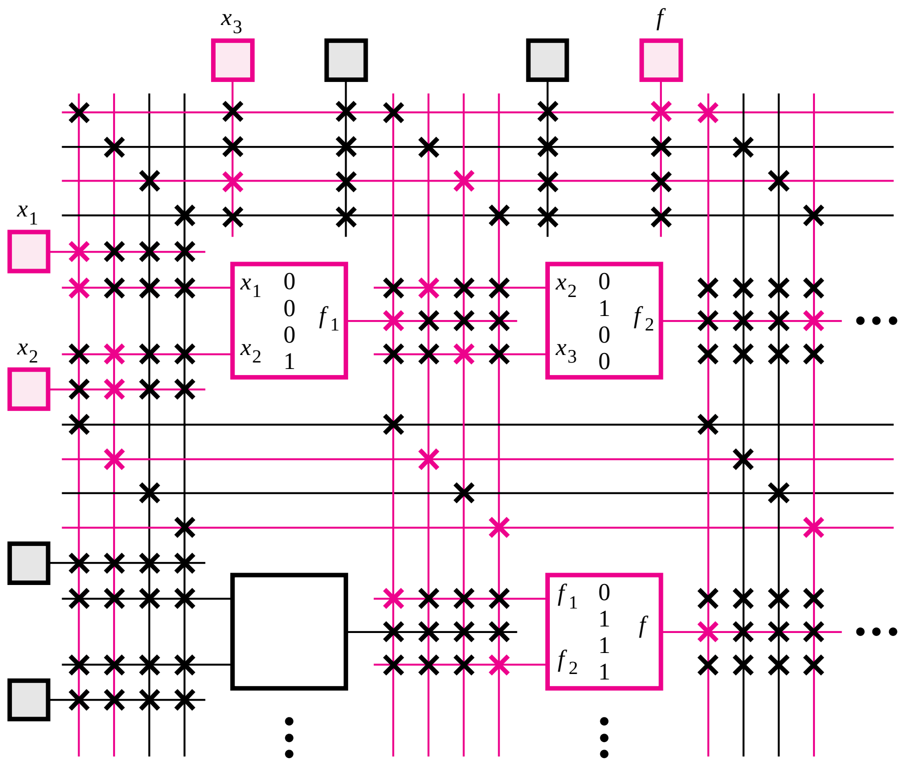


Figure 3.39 A section of a programmed FPGA.

3.7 Custom Chips, Standard Cells, and Gate Arrays

The key factor that limits the size of a circuit that can be accommodated in a PLD is the existence of programmable switches. Although these switches provide the benefit of user programmability, they consume a significant amount of space on the chip. They also result in a reduction in the speed of operation of circuits. In this section we will introduce some integrated circuit technologies that do not contain programmable switches.

Chips that provide the largest number of logic gates and the highest speed are so-called *custom chips*. Whereas a PLD is prefabricated, containing logic gates and programmable switches that are programmed to realize a user's circuit, a custom chip is created from scratch. The designer of a custom chip has complete flexibility to decide the size of the chip, the number of transistors the chip contains, the placement of each transistor on the chip, and the way the transistors are connected together. The process of defining exactly where on the chip each transistor and wire is situated is called *chip layout*. For a custom chip the designer may create any layout that is desired. Because it may contain more than a million transistors, a custom chip requires a large amount of design effort and therefore is expensive. Consequently, custom chips are used only when a very large number of transistors is needed and high-speed performance is important. Also, the product being designed must be expected to sell in sufficient quantities to recoup the expense. Two examples of products that are usually realized with custom chips are microprocessors and memory chips.

Some of the design effort incurred for a custom chip can be avoided by using a technology known as *standard cells*. Chips made using this technology are often called *application-specific integrated circuits (ASICs)*. This technology is illustrated in Figure 3.40, which depicts a small portion of a chip. The rows of logic gates may be connected by wires that are created in the *routing channels* between the rows of gates. In general, many types of logic gates may be used in such a chip. The available gates are prebuilt and are stored in a library that can be accessed by the designer. In Figure 3.40 the wires are drawn in two colors. This scheme is used because metal wires can be created on integrated circuits in multiple *layers*, which makes it possible for two wires to cross one another without creating a short circuit. The blue wires represent one layer of metal wires, and the black wires are a different layer. Each blue square represents a hard-wired connection (called a *via*) between a wire on one layer and a wire on the other layer. In current technology it is possible to have eight or more layers of metal wiring. Some of the metal layers can be placed on top of the transistors in the logic gates, resulting in a more efficient chip layout.

Like a custom chip, a standard-cell chip is created from scratch according to a user's specifications. The circuitry shown in Figure 3.40 implements the two logic functions that we realized in a PLA in Figure 3.26, namely, $f_1 = x_1x_2 + x_1\overline{x}_3 + \overline{x}_1\overline{x}_2x_3$ and $f_2 = x_1x_2 + \overline{x}_1\overline{x}_2x_3 + x_1x_3$. Because of the expense involved, a standard-cell chip would never be created for a small circuit such as this one, and thus the figure shows only a portion of a much larger chip. The layout of individual gates (standard cells) is predesigned and fixed. The chip layout can be created automatically by CAD tools because of the regular arrangement of the logic gates (cells) in rows. A typical chip has many long rows of logic gates with a large number of wires between each pair of rows. The I/O blocks around the periphery connect to the pins of the chip package, which is usually a QFP, PGA, or BGA package.

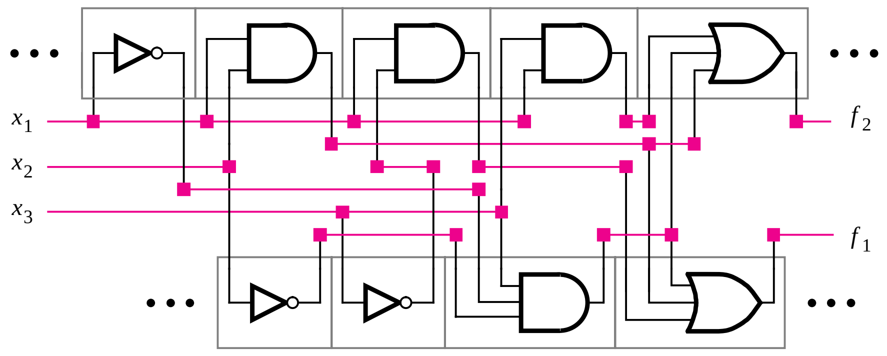


Figure 3.40 A section of two rows in a standard-cell chip.

Another technology, similar to standard cells, is the *gate-array* technology. In a gate array parts of the chip are prefabricated, and other parts are custom fabricated for a particular user's circuit. This concept exploits the fact that integrated circuits are fabricated in a sequence of steps, some steps to create transistors and other steps to create wires to connect the transistors together. In gate-array technology, the manufacturer performs most of the fabrication steps, typically those involved in the creation of the transistors, without considering the requirements of a user's circuit. This process results in a silicon wafer (see Figure 1.1) of partially finished chips, called the gate-array *template*. Later the template is modified, usually by fabricating wires that connect the transistors together, to create a user's circuit in each finished chip. The gate-array approach provides cost savings in comparison to the custom-chip approach because the gate-array manufacturer can amortize the cost of chip fabrication over a large number of template wafers, all of which are identical. Many variants of gate-array technology exist. Some have relatively large logic cells, while others are configurable at the level of a single transistor.

An example of a gate-array template is given in Figure 3.41. The gate array contains a two-dimensional array of logic cells. The chip's general structure is similar to a standard-cell chip except that in the gate array all logic cells are identical. Although the types of logic cells used in gate arrays vary, one common example is a two- or three-input NAND gate. In some gate arrays empty spaces exist between the rows of logic cells to accommodate the wires that will be added later to connect the logic cells together. However, most gate arrays do not have spaces between rows of logic cells, and the interconnection wires are fabricated on top of the logic cells. This design is possible because, as discussed for Figure 3.40, metal wires can be created on a chip in multiple layers. This technology is known

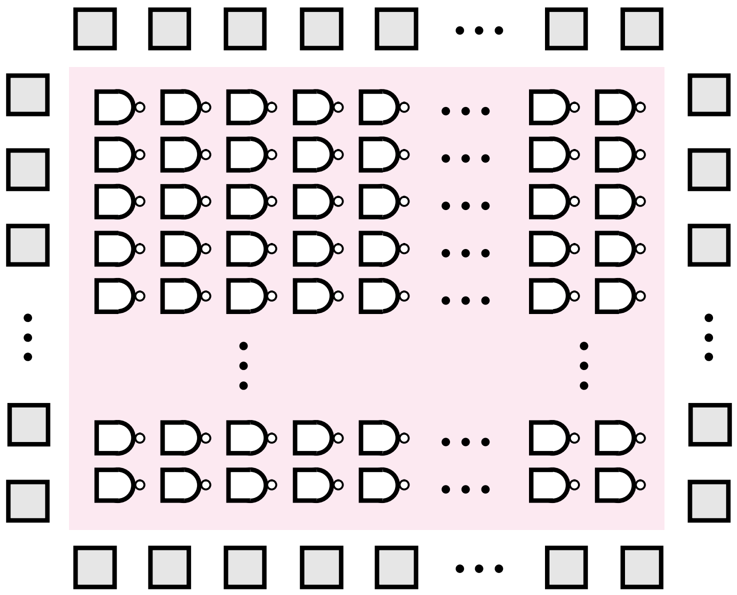


Figure 3.41 A sea-of-gates gate array.

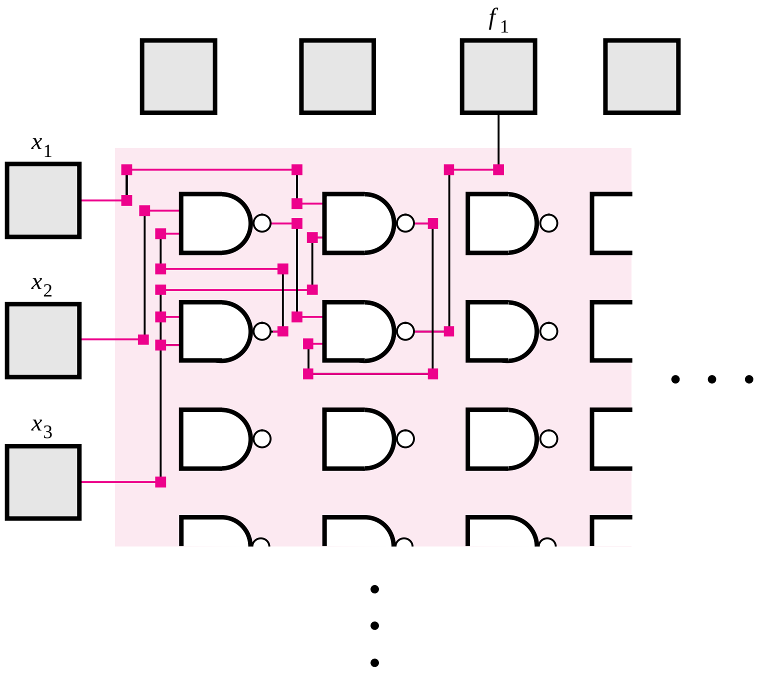


Figure 3.42 The logic function $f_1 = x_2\overline{x}_3 + x_1x_3$ in the gate array of Figure 3.41.

as the *sea-of-gates* technology. Figure 3.42 depicts a small section of a gate array that has been customized to implement the logic function $f = x_2\overline{x}_3 + x_1x_3$. It is easy to verify that this circuit with only NAND gates is equivalent to the AND-OR form of the circuit. We will describe a process for deriving this equivalence in section 4.6.

3.8 Practical Aspects

So far in this chapter, we have described the basic aspects of logic gate circuits and given examples of commercial chips. In this section we provide more detailed information on several aspects of digital circuits. We describe how transistors are fabricated in silicon and give a detailed explanation of how transistors operate. We discuss the robustness of logic circuits and discuss the important issues of signal propagation delays and power dissipation in logic gates.

3.8.1 MOSFET Fabrication and Behavior

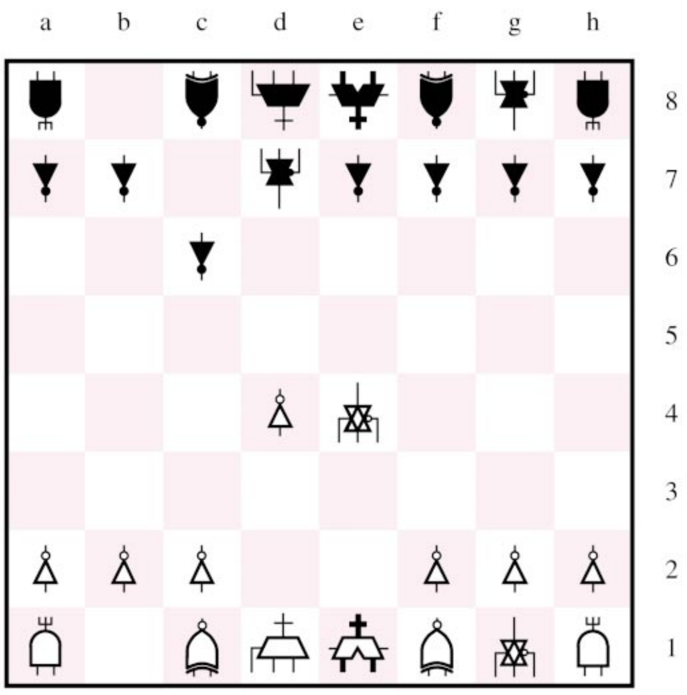
To understand the operation of NMOS and PMOS transistors, we need to consider how they are built in an integrated circuit. Integrated circuits are fabricated on silicon wafers.

April 13, 1999 12:52 g02-ch4 Sheet number 1 Page number 143 black

chapter

4

Optimized Implementation of Logic Functions



4. Nc3xe4, Nb8–d7

In Chapter 2 we showed that algebraic manipulation can be used to find the lowest-cost implementations of logic functions. The purpose of that chapter was to introduce the basic concepts in the synthesis process. The reader is probably convinced that it is easy to derive a straightforward realization of a logic function in a canonical form, but it is not at all obvious how to choose and apply the theorems and properties of section 2.5 to find a minimum-cost circuit. Indeed, the algebraic manipulation is rather tedious and quite impractical for functions of many variables.

If CAD tools are used to design logic circuits, the task of minimizing the cost of implementation does not fall to the designer; the tools perform the necessary optimizations automatically. Even so, it is essential to know something about this process. Most CAD tools have many features and options that are under control of the user. To know when and how to apply these options, the user must have an understanding of what the tools do.

In this chapter we will introduce some of the optimization techniques implemented in CAD tools and show how these techniques can be automated. As a first step we will discuss a graphical approach, known as the Karnaugh map, which provides a neat way to manually derive minimum-cost implementations of simple logic functions. Although it is not suitable for implementation in CAD tools, it illustrates a number of key concepts. We will show how both two-level and multilevel circuits can be designed. Then we will describe a cubical representation for logic functions, which is suitable for use in CAD tools. We will also continue our discussion of the VHDL language and CAD tools.

4.1 KARNAUGH MAP

In section 2.6 we saw that the key to finding a minimum-cost expression for a given logic function is to reduce the number of product (or sum) terms needed in the expression, by applying the combining property 14*a* (or 14*b*) as judiciously as possible. The Karnaugh map approach provides a systematic way of performing this optimization. To understand how it works, it is useful to review the algebraic approach from Chapter 2. Consider the function f in Figure 4.1. The canonical sum-of-products expression for f consists of minterms m_0, m_2, m_4, m_5, and m_6, so that

$$f = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3$$

The combining property 14*a* allows us to replace two minterms that differ in the value of only one variable with a single product term that does not include that variable at all. For example, both m_0 and m_2 include $\overline{x}_1$ and $\overline{x}_3$, but they differ in the value of x_2 because m_0 includes $\overline{x}_2$ while m_2 includes x_2. Thus

$$\begin{aligned}\overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 &= \overline{x}_1(\overline{x}_2 + x_2)\overline{x}_3\\ &= \overline{x}_1 \cdot 1 \cdot \overline{x}_3\\ &= \overline{x}_1\overline{x}_3\end{aligned}$$

Row number	x_1	x_2	x_3	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 4.1 The function $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

Hence m_0 and m_2 can be replaced by the single product term $\overline{x}_1\overline{x}_3$. Similarly, m_4 and m_6 differ only in the value of x_2 and can be combined using

$$\begin{aligned} x_1\overline{x}_2\overline{x}_3 + x_1x_2\overline{x}_3 &= x_1(\overline{x}_2 + x_2)\overline{x}_3 \\ &= x_1 \cdot 1 \cdot \overline{x}_3 \\ &= x_1\overline{x}_3 \end{aligned}$$

Now the two newly generated terms, $\overline{x}_1\overline{x}_3$ and $x_1\overline{x}_3$, can be combined further as

$$\begin{aligned} \overline{x}_1\overline{x}_3 + x_1\overline{x}_3 &= (\overline{x}_1 + x_1)\overline{x}_3 \\ &= 1 \cdot \overline{x}_3 \\ &= \overline{x}_3 \end{aligned}$$

These optimization steps indicate that we can replace the four minterms m_0, m_2, m_4, and m_6 with the single product term $\overline{x}_3$. In other words, the minterms m_0, m_2, m_4, and m_6 are all *included* in the term $\overline{x}_3$. The remaining minterm in f is m_5. It can be combined with m_4, which gives

$$x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 = x_1\overline{x}_2$$

Recall that theorem 7*b* in section 2.5 indicates that

$$m_4 = m_4 + m_4$$

which means that we can use the minterm m_4 twice—to combine with minterms m_0, m_2, and m_6 to yield the term $\overline{x}_3$ as explained above and also to combine with m_5 to yield the term $x_1\overline{x}_2$.

We have now accounted for all the minterms in f; hence all five input valuations for which $f = 1$ are covered by the minimum-cost expression

$$f = \overline{x}_3 + x_1\overline{x}_2$$

The expression has the product term $\overline{x}_3$ because $f = 1$ when $x_3 = 0$ regardless of the values of x_1 and x_2. The four minterms m_0, m_2, m_4, and m_6 represent all possible minterms for which $x_3 = 0$; they include all four valuations, 00, 01, 10, and 11, of variables x_1 and x_2. Thus if $x_3 = 0$, then it is guaranteed that $f = 1$. This may not be easy to see directly from the truth table in Figure 4.1, but it is obvious if we write the corresponding valuations grouped together:

	x_1	x_2	x_3
m_0	0	0	0
m_2	0	1	0
m_4	1	0	0
m_6	1	1	0

In a similar way, if we look at m_4 and m_5 as a group of two

	x_1	x_2	x_3
m_4	1	0	0
m_5	1	0	1

it is clear that when $x_1 = 1$ and $x_2 = 0$, then $f = 1$ regardless of the value of x_3.

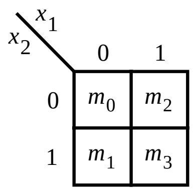
The preceding discussion suggests that it would be advantageous to devise a method that allows easy discovery of groups of minterms for which $f = 1$ that can be combined into single terms. The Karnaugh map is a useful vehicle for this purpose.

The *Karnaugh map* [1] is an alternative to the truth-table form for representing a function. The map consists of *cells* that correspond to the rows of the truth table. Consider the two-variable example in Figure 4.2. Part (*a*) depicts the truth-table form, where each of the four rows is identified by a minterm. Part (*b*) shows the Karnaugh map, which has four cells. The columns of the map are labeled by the value of x_1, and the rows are labeled by x_2. This labeling leads to the locations of minterms as shown in the figure. Compared to the truth table, the advantage of the Karnaugh map is that it allows easy recognition of minterms that can be combined using property 14*a* from section 2.5. Minterms in any two cells that are adjacent, either in the same row or the same column, can be combined. For example, the minterms m_2 and m_3 can be combined as

$$\begin{aligned} m_2 + m_3 &= x_1\overline{x}_2 + x_1x_2 \\ &= x_1(\overline{x}_2 + x_2) \\ &= x_1 \cdot 1 \\ &= x_1 \end{aligned}$$

x_1	x_2	
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3

(a) Truth table



(b) Karnaugh map

Figure 4.2 Location of two-variable minterms.

The Karnaugh map is not just useful for combining pairs of minterms. As we will see in several larger examples, the Karnaugh map can be used directly to derive a minimum-cost circuit for a logic function.

Two-Variable Map

A Karnaugh map for a two-variable function is given in Figure 4.3. It corresponds to the function f of Figure 2.15. The value of f for each valuation of the variables x_1 and x_2 is indicated in the corresponding cell of the map. Because a 1 appears in both cells of the bottom row and these cells are adjacent, there exists a single product term that can cause f to be equal to 1 when the input variables have the values that correspond to either of these cells. To indicate this fact, we have circled the cell entries in the map. Rather than using the combining property formally, we can derive the product term intuitively. Both of the cells are identified by $x_2 = 1$, but $x_1 = 0$ for the left cell and $x_1 = 1$ for the right cell. Thus if $x_2 = 1$, then $f = 1$ regardless of whether x_1 is equal to 0 or 1. The product term representing the two cells is simply x_2.

Similarly, $f = 1$ for both cells in the first column. These cells are identified by $x_1 = 0$. Therefore, they lead to the product term $\overline{x}_1$. Since this takes care of all instances where $f = 1$, it follows that the minimum-cost realization of the function is

$$f = x_2 + \overline{x}_1$$

Evidently, to find a minimum-cost implementation of a given function, it is necessary to find the smallest number of product terms that produce a value of 1 for all cases where

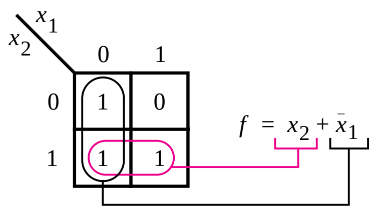


Figure 4.3 The function of Figure 2.15.

$f = 1$. Moreover, the cost of these product terms should be as low as possible. Note that a product term that covers two adjacent cells is cheaper to implement than a term that covers only a single cell. For our example once the two cells in the bottom row have been covered by the product term x_2, only one cell (top left) remains. Although it could be covered by the term $\overline{x}_1\overline{x}_2$, it is better to combine the two cells in the left column to produce the product term $\overline{x}_1$ because this term is cheaper to implement.

Three-Variable Map

A three-variable Karnaugh map is constructed by placing 2 two-variable maps side by side. Figure 4.4 shows the map and indicates the locations of minterms in it. In this case each valuation of x_1 and x_2 identifies a column in the map, while the value of x_3 distinguishes the two rows. To ensure that minterms in the adjacent cells in the map can always be combined into a single product term, the adjacent cells must differ in the value of only one variable. Thus the columns are identified by the sequence of (x_1, x_2) values of 00, 01, 11, and 10, rather than the more obvious 00, 01, 10, and 11. This makes the second and third columns different only in variable x_1. Also, the first and the fourth columns differ only in variable x_1, which means that these columns can be considered as being adjacent. The reader may find it useful to visualize the map as a rectangle folded into a cylinder where the left and the right edges in Figure 4.4*b* are made to touch. (A sequence of codes, or valuations, where consecutive codes differ in one variable only is known as the *Gray code*. This code is used for a variety of purposes, some of which will be encountered later in the book.)

Figure 4.5*a* represents the function of Figure 2.18 in Karnaugh-map form. To synthesize this function, it is necessary to cover the four 1s in the map as efficiently as possible. It is not difficult to see that two product terms suffice. The first covers the 1s in the top row, which are represented by the term $x_1\overline{x}_3$. The second term is $\overline{x}_2x_3$, which covers the 1s in the bottom row. Hence the function is implemented as

$$f = x_1\overline{x}_3 + \overline{x}_2x_3$$

which describes the circuit obtained in Figure 2.19*a*.

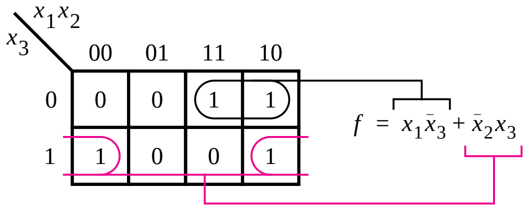
x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

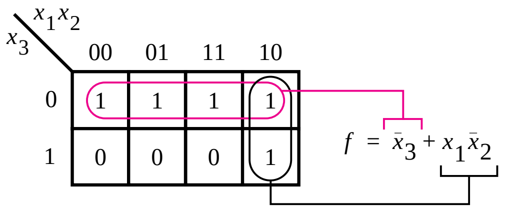
x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

(b) Karnaugh map

Figure 4.4 Location of three-variable minterms.



(a) The function of Figure 2.18



(b) The function of Figure 4.1

Figure 4.5 Examples of three-variable Karnaugh maps.

In a three-variable map it is possible to combine cells to produce product terms that correspond to a single cell, two adjacent cells, or a group of four adjacent cells. Realization of a group of four adjacent cells using a single product term is illustrated in Figure 4.5*b*, using the function from Figure 4.1. The four cells in the top row correspond to the (x_1, x_2, x_3) valuations 000, 010, 110, and 100. As we discussed before, this indicates that if $x_3 = 0$, then $f = 1$ for all four possible valuations of x_1 and x_2, which means that the only requirement is that $x_3 = 0$. Therefore, the product term $\bar{x}_3$ represents these four cells. The remaining 1, corresponding to minterm m_5, is best covered by the term $x_1\bar{x}_2$, obtained by combining the two cells in the right-most column. The complete realization of f is

$$f = \bar{x}_3 + x_1\bar{x}_2$$

It is also possible to have a group of eight 1s in a three-variable map. This is the trivial case where $f = 1$ for all valuations of input variables; in other words, f is equal to the constant 1.

The Karnaugh map provides a simple mechanism for generating the product terms that should be used to implement a given function. A product term must include only those variables that have the same value for all cells in the group represented by this term. If the variable is equal to 1 in the group, it appears uncomplemented in the product term; if it is equal to 0, it appears complemented. Each variable that is sometimes 1 and sometimes 0 in the group does not appear in the product term.

Four-Variable Map

A four-variable map is constructed by placing 2 three-variable maps together to create four rows in the same fashion as we used 2 two-variable maps to form the four columns in a three-variable map. Figure 4.6 shows the structure of the four-variable map and the location of minterms. We have included in this figure another frequently used way of designating the rows and columns. As shown in blue, it is sufficient to indicate the rows and columns for which a given variable is equal to 1. Thus $x_1 = 1$ for the two right-most columns, $x_2 = 1$ for the two middle columns, $x_3 = 1$ for the bottom two rows, and $x_4 = 1$ for the two middle rows.

Figure 4.7 gives four examples of four-variable functions. The function f_1 has a group of four 1s in adjacent cells in the bottom two rows, for which $x_2 = 0$ and $x_3 = 1$—they are represented by the product term $\overline{x}_2 x_3$. This leaves the two 1s in the second row to be covered, which can be accomplished with the term $x_1 \overline{x}_3 x_4$. Hence the minimum-cost implementation of the function is

$$f_1 = \overline{x}_2 x_3 + x_1 \overline{x}_3 x_4$$

The function f_2 includes a group of eight 1s that can be implemented by a single term, x_3. Again, the reader should note that if the remaining two 1s were implemented separately, the result would be the product term $x_1 \overline{x}_3 x_4$. Implementing these 1s as a part of a group of four 1s, as shown in the figure, gives the less expensive product term $x_1 x_4$.

Just as the left and the right edges of the map are adjacent in terms of the assignment of the variables, so are the top and the bottom edges. Indeed, the four corners of the map are adjacent to each other and thus can form a group of four 1s, which may be implemented by the product term $\overline{x}_2 \overline{x}_4$. This case is depicted by the function f_3. In addition to this group of 1s, there are four other 1s that must be covered to implement f_3. This can be done as shown in the figure.

In all examples that we have considered so far, a unique solution exists that leads to a minimum-cost circuit. The function f_4 provides an example where there is some choice. The groups of four 1s in the top-left and bottom-right corners of the map are realized by the terms $\overline{x}_1 \overline{x}_3$ and $x_1 x_3$, respectively. This leaves the two 1s that correspond to the term $x_1 x_2 \overline{x}_3$. But these two 1s can be realized more economically by treating them as a part of a group of four 1s. They can be included in two different groups of four, as shown in the figure. One

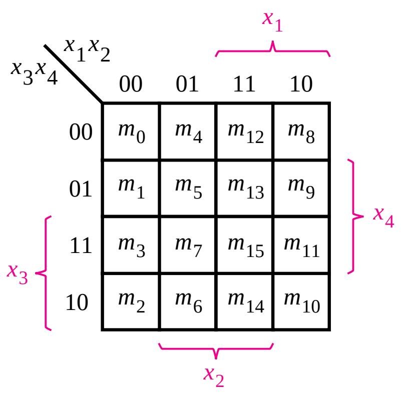


Figure 4.6 A four-variable Karnaugh map.

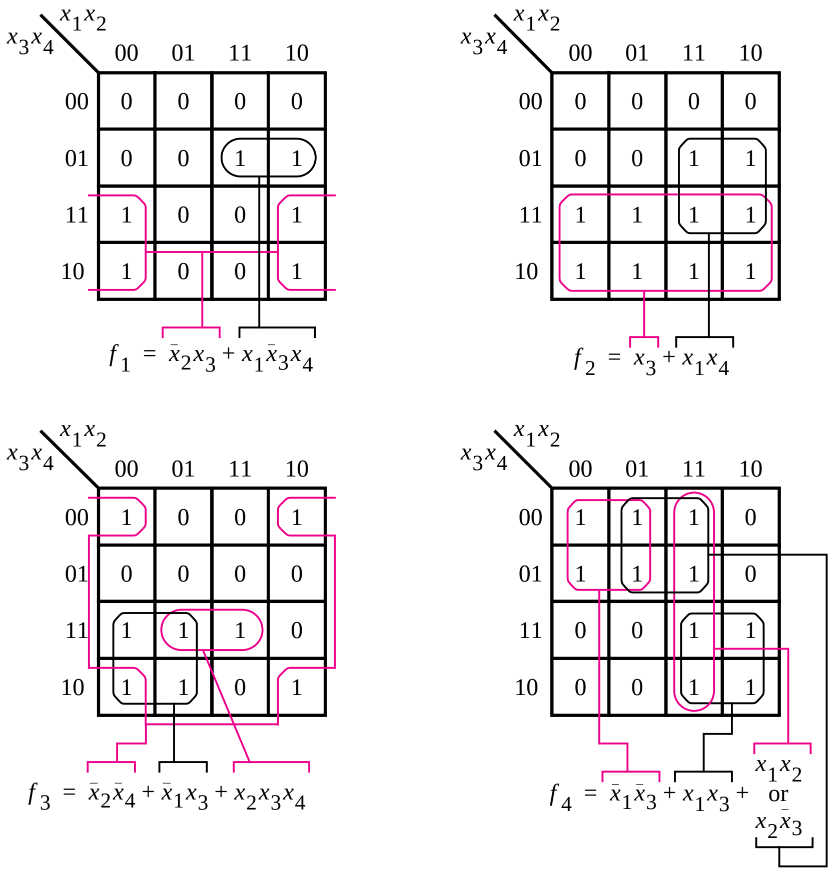
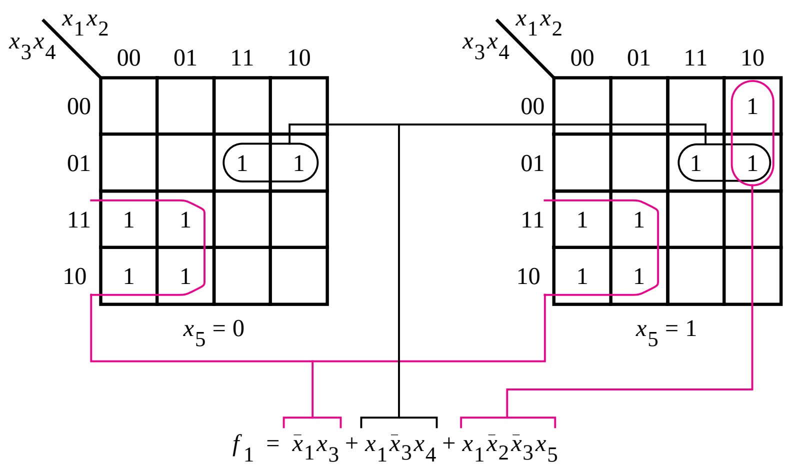

x_1x_2
x_3x_4
00 01 11 10
$f_1 = \bar{x}_2x_3 + x_1\bar{x}_3x_4$
$f_2 = x_3 + x_1x_4$
$f_3 = \bar{x}_2\bar{x}_4 + \bar{x}_1x_3 + x_2x_3x_4$
$f_4 = \bar{x}_1\bar{x}_3 + x_1x_3 +$ x_1x_2 or $x_2\bar{x}_3$

Figure 4.7 Examples of four-variable Karnaugh maps.

choice leads to the product term x_1x_2, and the other leads to $x_2\overline{x}_3$. Both of these terms have the same cost; hence it does not matter which one is chosen in the final circuit. Note that the complement of x_3 in the term $x_2\overline{x}_3$ does not imply an increased cost in comparison with x_1x_2, because this complement must be generated anyway to produce the term $\overline{x}_1\overline{x}_3$, which is included in the implementation.

Five-Variable Map

We can use 2 four-variable maps to construct a five-variable map. It is easy to imagine a structure where one map is directly behind the other, and they are distinguished by $x_5 = 0$ for one map and $x_5 = 1$ for the other map. Since such a structure is awkward to draw, we can simply place the two maps side by side as shown in Figure 4.8. For the logic function given in this example, two groups of four 1s appear in the same place in both four-variable maps; hence their realization does not depend on the value of x_5. The same is true for the two groups of two 1s in the second row. The 1 in the top-right corner appears only in the



x_1x_2
x_3x_4
00
01
11
10
$x_5 = 0$
$x_5 = 1$
$f_1 = \bar{x}_1x_3 + x_1\bar{x}_3x_4 + x_1\bar{x}_2\bar{x}_3x_5$

Figure 4.8 A five-variable Karnaugh map.

right map, where $x_5 = 1$; it is a part of the group of two 1s realized by the term $x_1\bar{x}_2\bar{x}_3x_5$. Note that in this map we left blank those cells for which $f = 0$, to make the figure more readable. We will do likewise in a number of maps that follow.

Using a five-variable map is obviously more awkward than using maps with fewer variables. Extending the Karnaugh map concept to more variables is not useful from the practical point of view. This is not troublesome, because practical synthesis of logic functions is done with CAD tools that perform the necessary minimization automatically. Although Karnaugh maps are occasionally useful for designing small logic circuits, our main reason for introducing the Karnaugh maps is to provide a simple vehicle for illustrating the ideas involved in the minimization process.

4.2 STRATEGY FOR MINIMIZATION

For the examples in the preceding section, we used an intuitive approach to decide how the 1s in a Karnaugh map should be grouped together to obtain the minimum-cost implementation of a given function. Our intuitive strategy was to find as few as possible and as large as possible groups of 1s that cover all cases where the function has a value of 1. Each group of 1s has to comprise cells that can be represented by a single product term. The larger the group of 1s, the fewer the number of variables in the corresponding product term. This approach worked well because the Karnaugh maps in our examples were small. For larger logic functions, which have many variables, such intuitive approach is unsuitable. Instead, we must have an organized method for deriving a minimum-cost implementation. In this section we will introduce a possible method, which is similar to the techniques that are automated in CAD tools. To illustrate the main ideas, we will use Karnaugh maps. Later, in section 4.9, we will describe a different way of representing logic functions, which is used in CAD tools.

4.2.1 Terminology

A huge amount of research work has gone into the development of techniques for synthesis of logic functions. The results of this research have been published in numerous papers. To facilitate the presentation of the results, certain terminology has evolved that avoids the need for using highly descriptive phrases. We define some of this terminology in the following paragraphs because it is useful for describing the minimization process.

Literal

A given product term consists of some number of variables, each of which may appear either in uncomplemented or complemented form. Each appearance of a variable, either uncomplemented or complemented, is called a *literal*. For example, the product term $x_1\overline{x}_2x_3$ has three literals, and the term $\overline{x}_1x_3\overline{x}_4x_6$ has four literals.

Implicant

A product term that indicates the input valuation(s) for which a given function is equal to 1 is called an *implicant* of the function. The most basic implicants are the minterms, which we introduced in section 2.6.1. For an n-variable function, a minterm is an implicant that consists of n literals.

Consider the three-variable function in Figure 4.9. There are 11 possible implicants for this function. This includes the five minterms: $\overline{x}_1\overline{x}_2\overline{x}_3$, $\overline{x}_1\overline{x}_2x_3$, $\overline{x}_1x_2\overline{x}_3$, $\overline{x}_1x_2x_3$, and $x_1x_2x_3$. Then there are the implicants that correspond to all possible pairs of minterms that can be combined, namely, $\overline{x}_1\overline{x}_2$ (m_0 and m_1), $\overline{x}_1\overline{x}_3$ (m_0 and m_2), $\overline{x}_1x_3$ (m_1 and m_3), $\overline{x}_1x_2$ (m_2 and m_3), and x_2x_3 (m_3 and m_7). Finally, there is one implicant that covers a group of four minterms, which consists of a single literal $\overline{x}_1$.

Prime Implicant

An implicant is called a *prime implicant* if it cannot be combined into another implicant that has fewer literals. Another way of stating this definition is to say that it is impossible to delete any literal in a prime implicant and still have a valid implicant.

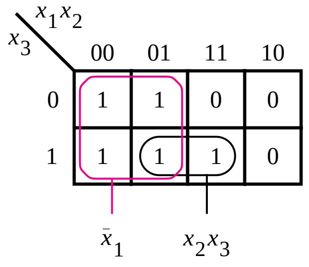


Figure 4.9 Three-variable function $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$.

In Figure 4.9 there are two prime implicants: $\overline{x}_1$ and x_2x_3. It is not possible to delete a literal in either of them. Doing so for $\overline{x}_1$ would make it disappear. For x_2x_3, deleting a literal would leave either x_2 or x_3. But x_2 is not an implicant because it includes the valuation $(x_1, x_2, x_3) = 110$ for which $f = 0$, and x_3 is not an implicant because it includes $(x_1, x_2, x_3) = 101$ for which $f = 0$.

Cover

A collection of implicants that account for all valuations for which a given function is equal to 1 is called a *cover* of that function. A number of different covers exist for most functions. Obviously, a set of all minterms for which $f = 1$ is a cover. It is also apparent that a set of all prime implicants is a cover.

A cover defines a particular implementation of the function. In Figure 4.9 a cover consisting of minterms leads to the expression

$$f = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1x_2x_3$$

Another valid cover is given by the expression

$$f = \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + x_2x_3$$

The cover comprising the prime implicants is

$$f = \overline{x}_1 + x_2x_3$$

While all of these expressions represent the function f correctly, the cover consisting of prime implicants leads to the lowest-cost implementation.

Cost

In Chapter 2 we suggested that a good indication of the cost of a logic circuit is the number of gates plus the total number of inputs to all gates in the circuit. We will use this definition of cost throughout the book. But we will assume that primary inputs, namely, the input variables, are available in both true and complemented forms at zero cost. Thus the expression

$$f = x_1\overline{x}_2 + x_3\overline{x}_4$$

has a cost of nine because it can be implemented using two AND gates and one OR gate, with six inputs to the AND and OR gates.

If an inversion is needed inside a circuit, then the corresponding NOT gate and its input are included in the cost. For example, the expression

$$g = \overline{x_1\overline{x}_2 + x_3}(\overline{x}_4 + x_5)$$

is implemented using two AND gates, two OR gates, and one NOT gate to complement $(x_1\overline{x}_2 + x_3)$, with nine inputs. Hence the total cost is 14.

4.2.2 MINIMIZATION PROCEDURE

We have seen that it is possible to implement a given logic function with various circuits. These circuits may have different structures and different costs. When designing a logic circuit, there are usually certain criteria that must be met. One such criterion is likely to be the cost of the circuit, which we considered in the previous discussion. In general, the larger the circuit, the more important the cost issue becomes. In this section we will assume that the main objective is to obtain a minimum-cost circuit.

Having said that cost is the primary concern, we should note that other optimization criteria may be more appropriate in some cases. For instance, in Chapter 3 we described several types of programmable-logic devices (PLDs) that have a predefined basic structure and can be programmed to realize a variety of different circuits. For such devices the main objective is to design a particular circuit so that it will fit into the target device. Whether or not this circuit has the minimum cost is not important if it can be realized successfully on the device. A CAD tool intended for design with a specific device in mind will automatically perform optimizations that are suitable for that device. We will show in section 4.7 that the way in which a circuit should be optimized may be different for different types of devices.

In the previous subsection we concluded that the lowest-cost implementation is achieved when the cover of a given function consists of prime implicants. The question then is how to determine the minimum-cost subset of prime implicants that will cover the function. Some prime implicants may have to be included in the cover, while for others there may be a choice. If a prime implicant includes a minterm for which $f = 1$ that is not included in any other prime implicant, then it must be included in the cover and is called an *essential prime implicant*. In the example in Figure 4.9, both prime implicants are essential. The term x_2x_3 is the only prime implicant that covers the minterm m_7, and $\overline{x}_1$ is the only one that covers the minterms m_0, m_1, and m_2. Notice that the minterm m_3 is covered by both of these prime implicants. The minimum-cost realization of the function is

$$f = \overline{x}_1 + x_2x_3$$

We will now present several examples in which there is a choice as to which prime implicants to include in the final cover. Consider the four-variable function in Figure 4.10. There are five prime implicants: $\overline{x}_1x_3$, $\overline{x}_2x_3$, $x_3\overline{x}_4$, $\overline{x}_1x_2x_4$, and $x_2\overline{x}_3x_4$. The essential ones

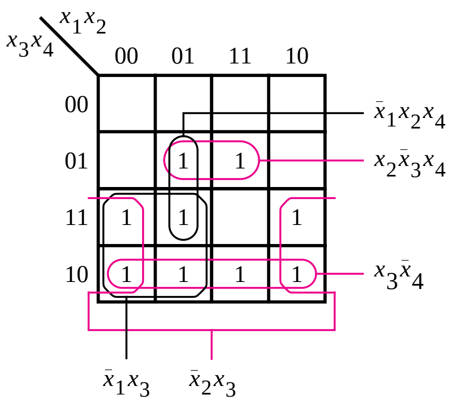


Figure 4.10 Four-variable function $f(x_1, \ldots, x_4) = \sum m(2, 3, 5, 6, 7, 10, 11, 13, 14)$.

(highlighted in blue) are $\overline{x}_2x_3$ (because of m_{11}), $x_3\overline{x}_4$ (because of m_{14}), and $x_2\overline{x}_3x_4$ (because of m_{13}). They must be included in the cover. These three prime implicants cover all minterms for which $f = 1$ except m_7. It is clear that m_7 can be covered by either $\overline{x}_1x_3$ or $\overline{x}_1x_2x_4$. Because $\overline{x}_1x_3$ has a lower cost, it is chosen for the cover. Therefore, the minimum-cost realization is

$$f = \overline{x}_2x_3 + x_3\overline{x}_4 + x_2\overline{x}_3x_4 + \overline{x}_1x_3$$

From the preceding discussion, the process of finding a minimum-cost circuit involves the following steps:

1. Generate all prime implicants for the given function f.
2. Find the set of essential prime implicants.
3. If the set of essential prime implicants covers all valuations for which $f = 1$, then this set is the desired cover of f. Otherwise, determine the nonessential prime implicants that should be added to form a complete minimum-cost cover.

The choice of nonessential prime implicants to be included in the cover is governed by the cost considerations. This choice is often not obvious. Indeed, for large functions there may exist many possibilities, and some *heuristic* approach (i.e., an approach that considers only a subset of possibilities but gives good results most of the time) has to be used. One such approach is to arbitrarily select one nonessential prime implicant and include it in the cover and then determine the rest of the cover. Next, another cover is determined assuming that this prime implicant is not in the cover. The costs of the resulting covers are compared, and the less-expensive cover is chosen for implementation.

We can illustrate the process by using the function in Figure 4.11. Of the six prime implicants, only $\overline{x}_3\overline{x}_4$ is essential. Consider next $x_1x_2\overline{x}_3$ and assume first that it will be included in the cover. Then the remaining three minterms, m_{10}, m_{11}, and m_{15}, will require two more prime implicants to be included in the cover. A possible implementation is

$$f = \overline{x}_3\overline{x}_4 + x_1x_2\overline{x}_3 + x_1x_3x_4 + x_1\overline{x}_2x_3$$

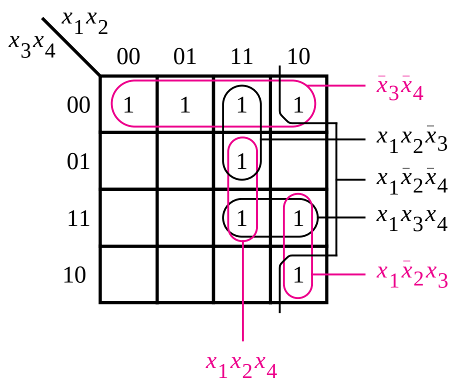


Figure 4.11 The function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$.

The second possibility is that $x_1x_2\overline{x}_3$ is not included in the cover. Then $x_1x_2x_4$ becomes essential because there is no other way of covering m_{13}. Because $x_1x_2x_4$ also covers m_{15}, only m_{10} and m_{11} remain to be covered, which can be achieved with $x_1\overline{x}_2x_3$. Therefore, the alternative implementation is

$$f = \overline{x}_3\overline{x}_4 + x_1x_2x_4 + x_1\overline{x}_2x_3$$

Clearly, this implementation is a better choice.

Sometimes there may not be any essential prime implicants at all. An example is given in Figure 4.12. Choosing any of the prime implicants and first including it, then excluding it from the cover leads to two alternatives of equal cost. One includes the prime implicants indicated in black, which yields

$$f = \overline{x}_1\overline{x}_3\overline{x}_4 + x_2\overline{x}_3x_4 + x_1x_3x_4 + \overline{x}_2x_3\overline{x}_4$$

The other includes the prime implicants indicated in blue, which yields

$$f = \overline{x}_1\overline{x}_2\overline{x}_4 + \overline{x}_1x_2\overline{x}_3 + x_1x_2x_4 + x_1\overline{x}_2x_3$$

This procedure can be used to find minimum-cost implementations of both small and large logic functions. For our small examples it was convenient to use Karnaugh maps to determine the prime implicants of a function and then choose the final cover. Other techniques based on the same principles are much more suitable for use in CAD tools; we will introduce one such technique in sections 4.9 and 4.10.

The previous examples have been based on the sum-of-products form. We will next illustrate that the same concepts apply for the product-of-sums form.

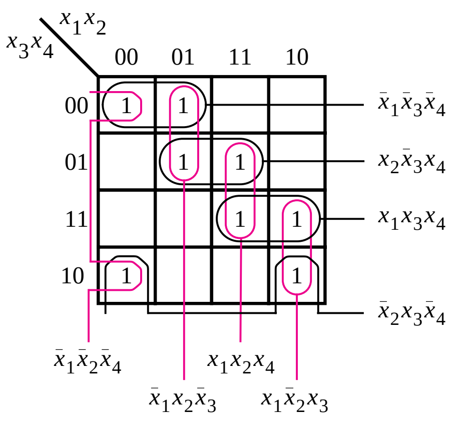


Figure 4.12 The function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 10, 11, 13, 15)$.

4.3 Minimization of Product-of-Sums Forms

Now that we know how to find the minimum-cost sum-of-products (SOP) implementations of functions, we can use the same techniques and the principle of duality to obtain minimum-cost product-of-sums (POS) implementations. In this case it is the maxterms for which $f = 0$ that have to be combined into sum terms that are as large as possible. Again, a sum term is considered larger if it covers more maxterms, and the larger the term, the less costly it is to implement.

Figure 4.13 depicts the same function as Figure 4.9 depicts. There are three maxterms that must be covered: M_4, M_5, and M_6. They can be covered by two sum terms shown in the figure, leading to the following implementation:

$$f = (\overline{x}_1 + x_2)(\overline{x}_1 + x_3)$$

A circuit corresponding to this expression has two OR gates and one AND gate, with two inputs for each gate. Its cost is greater than the cost of the equivalent SOP implementation derived in Figure 4.9, which requires only one OR gate and one AND gate.

The function from Figure 4.10 is reproduced in Figure 4.14. The maxterms for which $f = 0$ can be covered as shown, leading to the expression

$$f = (x_2 + x_3)(x_3 + x_4)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4)$$

This expression represents a circuit with three OR gates and one AND gate. Two of the OR gates have two inputs, and the third has four inputs; the AND gate has three inputs. Assuming that both the complemented and uncomplemented versions of the input variables x_1 to x_4 are available at no extra cost, the cost of this circuit is 15. This compares favorably with the SOP implementation derived from Figure 4.10, which requires five gates and 13 inputs at a total cost of 18.

In general, as we already know from section 2.6.1, the SOP and POS implementations of a given function may or may not entail the same cost. The reader is encouraged to find the POS implementations for the functions in Figures 4.11 and 4.12 and compare the costs with the SOP forms.

We have shown how to obtain minimum-cost POS implementations by finding the largest sum terms that cover all maxterms for which $f = 0$. Another way of obtaining

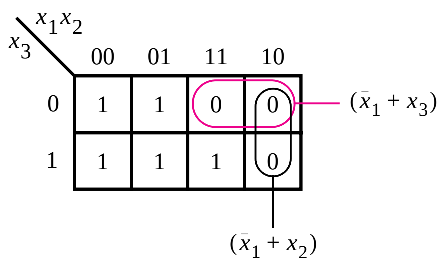


Figure 4.13 POS minimization of $f(x_1, x_2, x_3) = \Pi M(4, 5, 6)$.

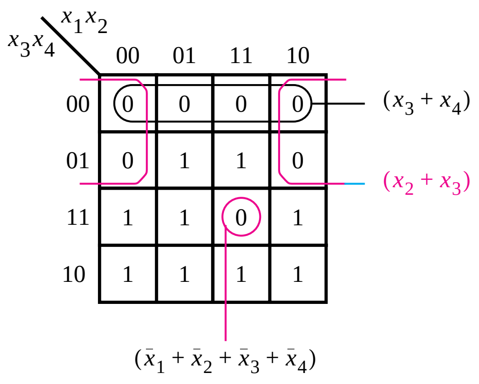


Figure 4.14 POS minimization of $f(x_1, \ldots, x_4) = \Pi M(0, 1, 4, 8, 9, 12, 15)$.

the same result is by finding a minimum-cost SOP implementation of the complement of f. Then we can apply DeMorgan's theorem to this expression to obtain the simplest POS realization because $f = \overline{\overline{f}}$. For example, the simplest SOP implementation of $\overline{f}$ in Figure 4.13 is

$$\overline{f} = x_1\overline{x}_2 + x_1\overline{x}_3$$

Complementing this expression using DeMorgan's theorem yields

$$\begin{aligned} f = \overline{\overline{f}} &= \overline{x_1\overline{x}_2 + x_1\overline{x}_3} \\ &= \overline{x_1\overline{x}_2} \cdot \overline{x_1\overline{x}_3} \\ &= (\overline{x}_1 + x_2)(\overline{x}_1 + x_3) \end{aligned}$$

which is the same result as obtained above.

Using this approach for the function in Figure 4.14 gives

$$\overline{f} = \overline{x}_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2x_3x_4$$

Complementing this expression produces

$$\begin{aligned} f = \overline{\overline{f}} &= \overline{\overline{x}_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2x_3x_4} \\ &= \overline{\overline{x}_2\overline{x}_3} \cdot \overline{\overline{x}_3\overline{x}_4} \cdot \overline{x_1x_2x_3x_4} \\ &= (x_2 + x_3)(x_3 + x_4)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4) \end{aligned}$$

which matches the previously derived implementation.

4.4 INCOMPLETELY SPECIFIED FUNCTIONS

In digital systems it often happens that certain input conditions can never occur. For example, suppose that x_1 and x_2 control two interlocked switches such that both switches cannot be closed at the same time. Thus the only three possible states of the switches are that both switches are open or that one switch is open and the other switch is closed. Namely, the input valuations $(x_1, x_2) = 00$, 01, and 10 are possible, but 11 is guaranteed not to occur. Then we say that $(x_1, x_2) = 11$ is a *don't-care condition*, meaning that a circuit with x_1 and x_2 as inputs can be designed by ignoring this condition. A function that has don't-care condition(s) is said to be *incompletely specified*.

Don't-care conditions, or *don't cares* for short, can be used to advantage in the design of logic circuits. Since these input valuations will never occur, the designer may assume that the function value for these valuations is either 1 or 0, whichever is more useful in trying to find a minimum-cost implementation. Figure 4.15 illustrates this idea. The required function has a value of 1 for minterms m_2, m_4, m_5, m_6, and m_{10}. Assuming the above-

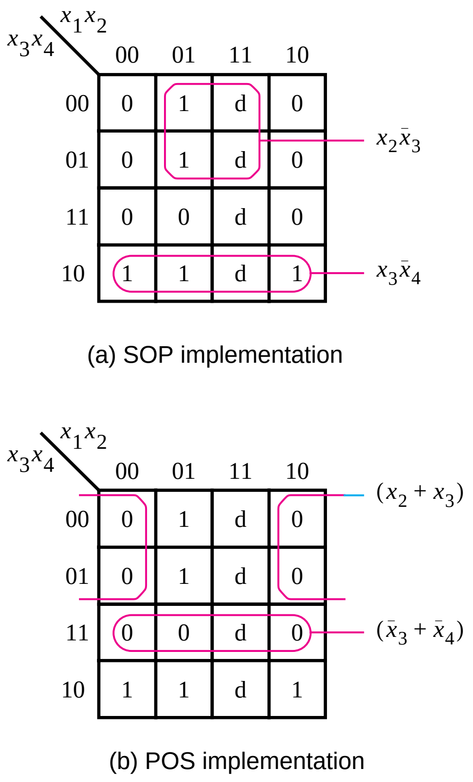


Figure 4.15 Two implementations of the function $f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$.

mentioned interlocked switches, the x_1 and x_2 inputs will never be equal to 1 at the same time; hence the minterms m_{12}, m_{13}, m_{14}, and m_{15} can all be used as don't cares. The don't cares are denoted by the letter d in the map. Using the shorthand notation, the function f is specified as

$$f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

where D is the set of don't cares.

Part (a) of the figure indicates the best sum-of-products implementation. To form the largest possible groups of 1s, thus generating the lowest-cost prime implicants, it is necessary to assume that the don't cares D_{12}, D_{13}, and D_{14} (corresponding to minterms m_{12}, m_{13}, and m_{14}) have the value of 1 while D_{15} has the value of 0. Then there are only two prime implicants, which provide a complete cover of f. The resulting implementation is

$$f = x_2\overline{x}_3 + x_3\overline{x}_4$$

Part (b) shows how the best product-of-sums implementation can be obtained. The same values are assumed for the don't cares. The result is

$$f = (x_2 + x_3)(\overline{x}_3 + \overline{x}_4)$$

The freedom in choosing the value of don't cares leads to greatly simplified realizations. If we were to naively exclude the don't cares from the synthesis of the function, by assuming that they always have a value of 0, the resulting SOP expression would be

$$f = \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_3\overline{x}_4 + \overline{x}_2x_3\overline{x}_4$$

and the POS expression would be

$$f = (x_2 + x_3)(\overline{x}_3 + \overline{x}_4)(\overline{x}_1 + \overline{x}_2)$$

Both of these expressions have higher costs than the expressions obtained with a more appropriate assignment of values to don't cares.

Although don't-care values can be assigned arbitrarily, an arbitrary assignment may not lead to a minimum-cost implementation of a given function. If there are k don't cares, then there are 2^k possible ways of assigning 0 or 1 values to them. In the Karnaugh map we can usually see how best to do this assignment to find the simplest implementation.

Using interlocked switches to illustrate how don't-care conditions can occur in a real system may seem to be somewhat contrived. However, in Chapters 6, 8, and 9 we will encounter many examples of don't cares that occur in the course of practical design of digital circuits.

4.5 MULTIPLE-OUTPUT CIRCUITS

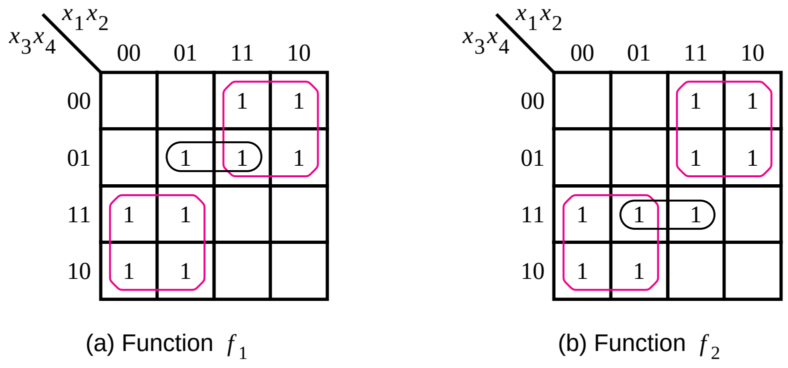
In all previous examples we have considered single functions and their circuit implementations. In practical digital systems it is necessary to implement a number of functions as part of some large logic circuit. Circuits that implement these functions can often be combined into a less-expensive single circuit with multiple outputs by sharing some of the gates needed in the implementation of individual functions.

Example 4.1 An example of gate sharing is given in Figure 4.16. Two functions, f_1 and f_2, of the same variables are to be implemented. The minimum-cost implementations for these functions are obtained as shown in parts (*a*) and (*b*) of the figure. This results in the expressions

$$f_1 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2\bar{x}_3x_4$$

$$f_2 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2x_3x_4$$

The cost of f_1 is four gates and 10 inputs, for a total of 14. The cost of f_2 is the same. Thus the total cost is 28 if both functions are implemented by separate circuits. A less-expensive



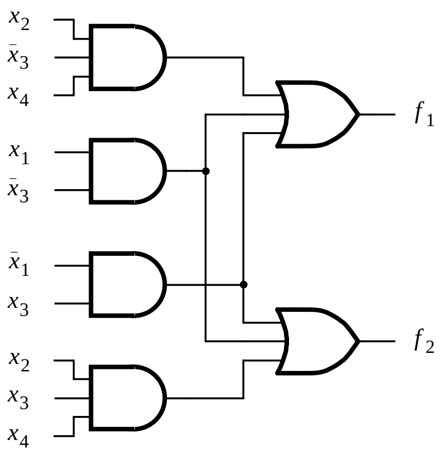


Figure 4.16 An example of multiple-output synthesis.

realization is possible if the two circuits are combined into a single circuit with two outputs. Because the first two product terms are identical in both expressions, the AND gates that implement them need not be duplicated. The combined circuit is shown in Figure 4.16*c*. Its cost is six gates and 16 inputs, for a total of 22.

In this example we reduced the overall cost by finding minimum-cost realizations of f_1 and f_2 and then sharing the gates that implement the common product terms. This strategy does not necessarily always work the best, as the next example shows.

Example 4.2

Figure 4.17 shows two functions to be implemented by a single circuit. Minimum-cost realizations of the individual functions f_3 and f_4 are obtained from parts (*a*) and (*b*) of the figure.

$$f_3 = \overline{x}_1 x_4 + x_2 x_4 + \overline{x}_1 x_2 x_3$$

$$f_4 = x_1 x_4 + \overline{x}_2 x_4 + \overline{x}_1 x_2 x_3 \overline{x}_4$$

None of the AND gates can be shared, which means that the cost of the combined circuit would be six AND gates, two OR gates, and 21 inputs, for a total of 29.

But several alternative realizations are possible. Instead of deriving the expressions for f_3 and f_4 using only prime implicants, we can look for other implicants that may be shared advantageously in the combined realization of the functions. Figure 4.17*c* shows the best choice of implicants, which yields the realization

$$f_3 = x_1 x_2 x_4 + \overline{x}_1 x_2 x_3 \overline{x}_4 + \overline{x}_1 x_4$$

$$f_4 = x_1 x_2 x_4 + \overline{x}_1 x_2 x_3 \overline{x}_4 + \overline{x}_2 x_4$$

The first two implicants are identical in both expressions. The resulting circuit is given in Figure 4.17*d*. It has the cost of six gates and 17 inputs, for a total of 23.

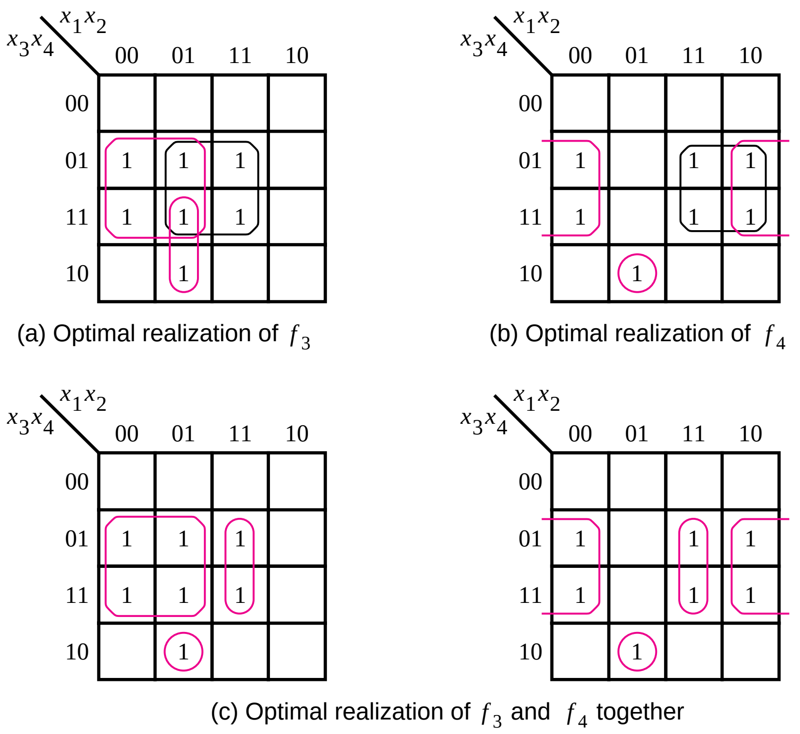
Example 4.3

In Example 4.1 we sought the best SOP implementation for the functions f_1 and f_2 in Figure 4.16. We will now consider the POS implementation of the same functions. The minimum-cost POS expressions for f_1 and f_2 are

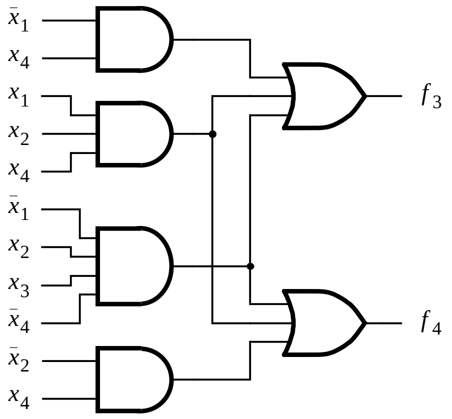
$$f_1 = (\overline{x}_1 + \overline{x}_3)(x_1 + x_2 + x_3)(x_1 + x_3 + x_4)$$

$$f_2 = (x_1 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_3 + x_4)$$

There are no common sum terms in these expressions that could be shared in the implementation. Moreover, from the Karnaugh maps in Figure 4.16, it is apparent that there is no sum term (covering the cells where $f_1 = f_2 = 0$) that can be profitably used in realizing both f_1 and f_2. Thus the best choice is to implement each function separately, according to the preceding expressions. Each function requires three OR gates, one AND gate, and 11 inputs. Therefore, the total cost of the circuit that implements both functions is 30. This realization is costlier than the SOP realization derived in Example 4.1.



x_1x_2
x_3x_4
00
01
11
10
1
(a) Optimal realization of f_3
(b) Optimal realization of f_4
(c) Optimal realization of f_3 and f_4 together



(d) Combined circuit for f_3 and f_4

Figure 4.17 Another example of multiple-output synthesis.

Example 4.4

Consider now the POS realization of the functions f_3 and f_4 in Figure 4.17. The minimum-cost POS expressions for f_3 and f_4 are

$$f_3 = (x_3 + x_4)(x_2 + x_4)(\overline{x}_1 + x_4)(\overline{x}_1 + x_2)$$

$$f_4 = (x_3 + x_4)(x_2 + x_4)(\overline{x}_1 + x_4)(x_1 + \overline{x}_2 + \overline{x}_4)$$

The first three sum terms are the same in both f_3 and f_4; they can be shared in a combined circuit. These terms require three OR gates and six inputs. In addition, one 2-input OR gate and one 4-input AND gate are needed for f_3, and one 3-input OR gate and one 4-input AND gate are needed for f_4. Thus the combined circuit comprises five OR gates, two AND gates, and 19 inputs, for a total cost of 26. This cost is slightly higher than the cost of the circuit derived in Example 4.2.

These examples show that the complexities of the best SOP or POS implementations of given functions may be quite different. For the functions in Figures 4.16 and 4.17, the SOP form gives better results. But if we are interested in implementing the complements of the four functions in these figures, then the POS form would be less costly.

Sophisticated CAD tools used to synthesize logic functions will automatically perform the types of optimizations illustrated in the preceding examples.

4.6 NAND AND NOR LOGIC NETWORKS

In Chapter 3 we saw that it is possible to design electronic circuits that realize basic logic functions other than AND, OR, and NOT, which have been the focus of our discussion to this point. From Figures 3.6 to 3.9 and Figures 3.13 to 3.15, it is obvious that NAND and NOR gates are simpler to implement than AND and OR gates. Then we should ask whether these gates can be used directly in the synthesis of logic circuits, rather than just being a part of the individual AND and OR gates. In section 2.5 we introduced DeMorgan's theorem. Its logic gate interpretation is shown in Figure 4.18. Identity 15*a* from section 2.5 is interpreted in part (*a*) of the figure. It specifies that a NAND of variables x_1 and x_2 is equivalent to first complementing each of the variables and then ORing them. Notice on the far-right side that we have indicated the NOT gates simply as small circles, which denote inversion of the logic value at that point. The other half of DeMorgan's theorem, identity 15*b*, appears in part (*b*) of the figure. It states that the NOR function is equivalent to first inverting the input variables and then ANDing them.

In previous sections we explained how any logic function can be implemented either in sum-of-products or product-of-sums form, which leads to logic networks that have either an AND-OR or an OR-AND structure, respectively. We will now show that such networks can be implemented using only NAND gates or only NOR gates.

Consider the network in Figure 4.19 as a representative of general AND-OR networks. This network can be transformed into a network of NAND gates as shown in the figure. First, each connection between the AND gate and an OR gate is replaced by a connection

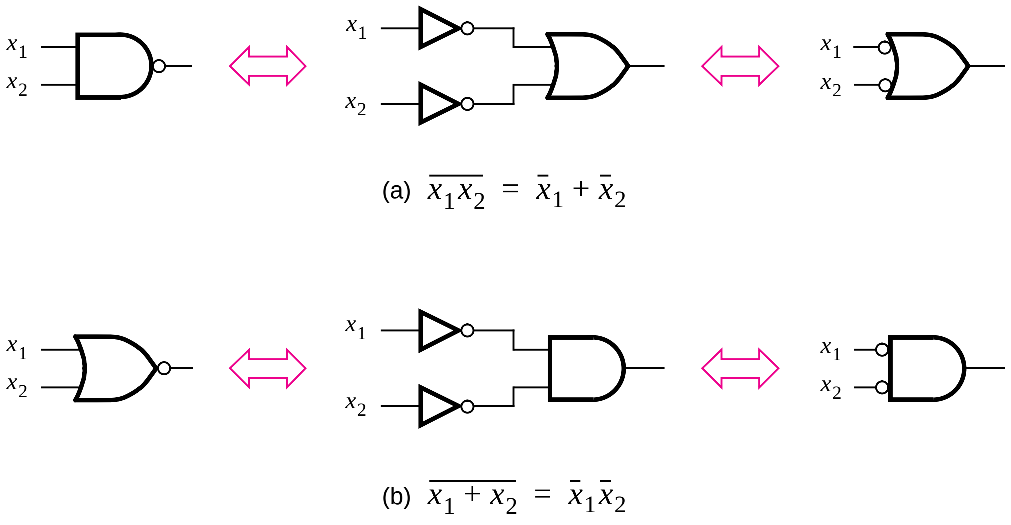

x_1
x_2
(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$
(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Figure 4.18 DeMorgan's theorem in terms of logic gates.

that includes two inversions of the signal: one inversion at the output of the AND gate and the other at the input of the OR gate. Such double inversion has no effect on the behavior of the network, as stated formally in theorem 9 in section 2.5. According to Figure 4.18*a*, the OR gate with inversions at its inputs is equivalent to a NAND gate. Thus we can redraw the network using only NAND gates, as shown in Figure 4.19. This example shows that

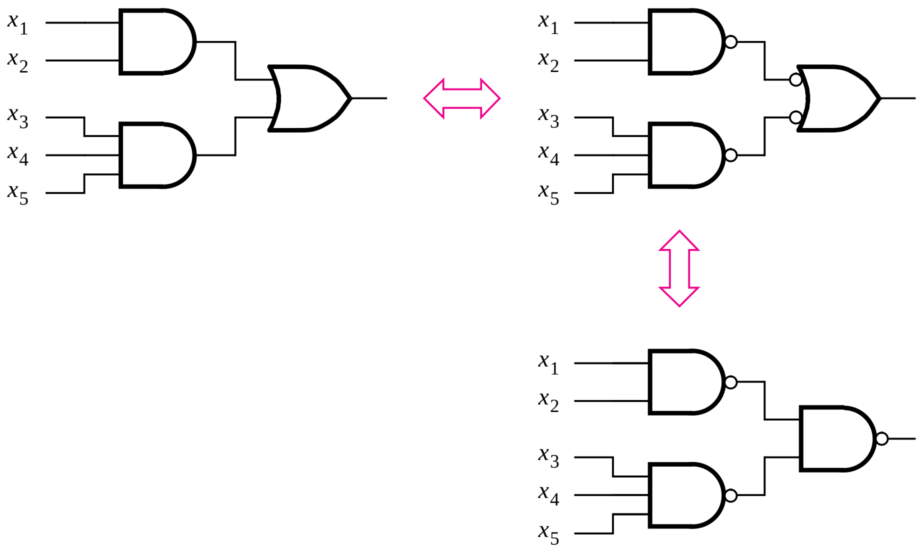


Figure 4.19 Using NAND gates to implement a sum-of-products.

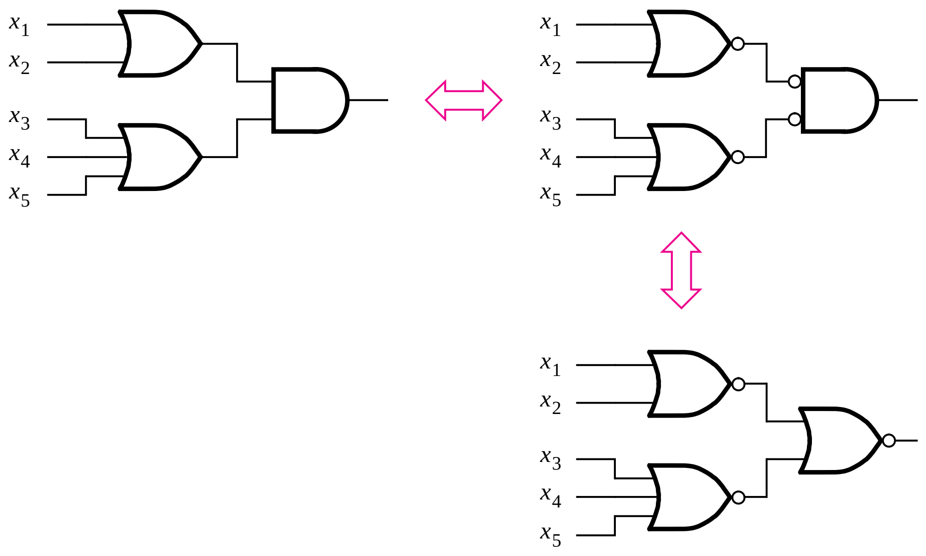


Figure 4.20 Using NOR gates to implement a product-of-sums.

any AND-OR network can be implemented as a NAND-NAND network having the same topology.

Figure 4.20 gives a similar construction for a product-of-sums network, which can be transformed into a circuit with only NOR gates. The procedure is exactly the same as the one described for Figure 4.19 except that now the identity in Figure 4.18*b* is applied. The conclusion is that any OR-AND network can be implemented as a NOR-NOR network having the same topology.

4.7 Multilevel Synthesis

In the preceding sections our objective was to find a minimum-cost sum-of-products or product-of-sums realization of a given logic function. Logic circuits of this type have *two levels* (stages) of gates. In the sum-of-products form, the first level comprises AND gates that are connected to a second-level OR gate. In the product-of-sums form, the first-level OR gates feed the second-level AND gate. We have assumed that both true and complemented versions of the input variables are available so that NOT gates are not needed to complement the variables.

A two-level realization is usually efficient for functions of a few variables. However, as the number of inputs increases, a two-level circuit may result in fan-in problems. Whether or not this is an issue depends on the type of technology that is used to implement the circuit. For example, consider the following function:

$$f(x_1, \ldots, x_7) = x_1x_3\overline{x}_6 + x_1x_4x_5\overline{x}_6 + x_2x_3x_7 + x_2x_4x_5x_7$$

This is a minimum-cost SOP expression. Now consider implementing f in two types of PLDs: a CPLD and an FPGA. Figure 4.21 shows one of the PAL-like blocks from Figure 3.33. The figure indicates in blue the circuitry used to realize the function f. Clearly, the SOP form of the function is well suited to the chip architecture of the CPLD.

Next, consider implementing f in an FPGA. For this example we will use the FPGA shown in Figure 3.39, which contains two-input LUTs. Since the SOP expression for f requires three- and four-input AND operations and a four-input OR, it cannot be directly implemented in this FPGA. The problem is that the fan-in required to implement the function is too high for our target chip architecture.

To solve the fan-in problem, f must be expressed in a form that has more than two levels of logic operations. Such a form is called a *multilevel* logic expression. There are several different approaches for synthesis of multilevel circuits. We will discuss two important techniques known as *factoring* and *functional decomposition*.

4.7.1 FACTORING

The distributive property in section 2.5 allows us to factor the preceding expression for f as follows

$$\begin{aligned} f &= x_1\overline{x}_6(x_3 + x_4x_5) + x_2x_7(x_3 + x_4x_5) \\ &= (x_1\overline{x}_6 + x_2x_7)(x_3 + x_4x_5) \end{aligned}$$

The corresponding circuit has a maximum fan-in of two; hence it can be realized using two-input LUTs. Figure 4.22 gives a possible implementation using the FPGA from Figure 3.39. Note that a two-variable function that has to be realized by each LUT is indicated in the box that represents the LUT.

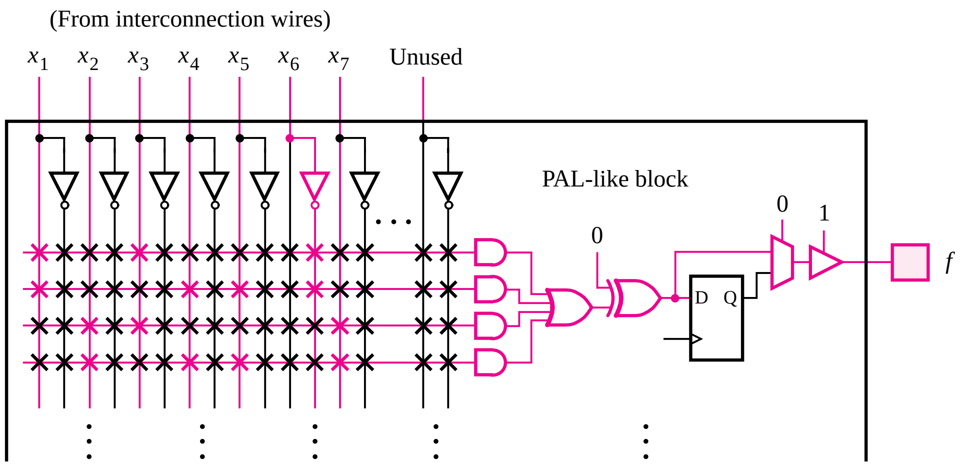


Figure 4.21 Implementation in a CPLD.

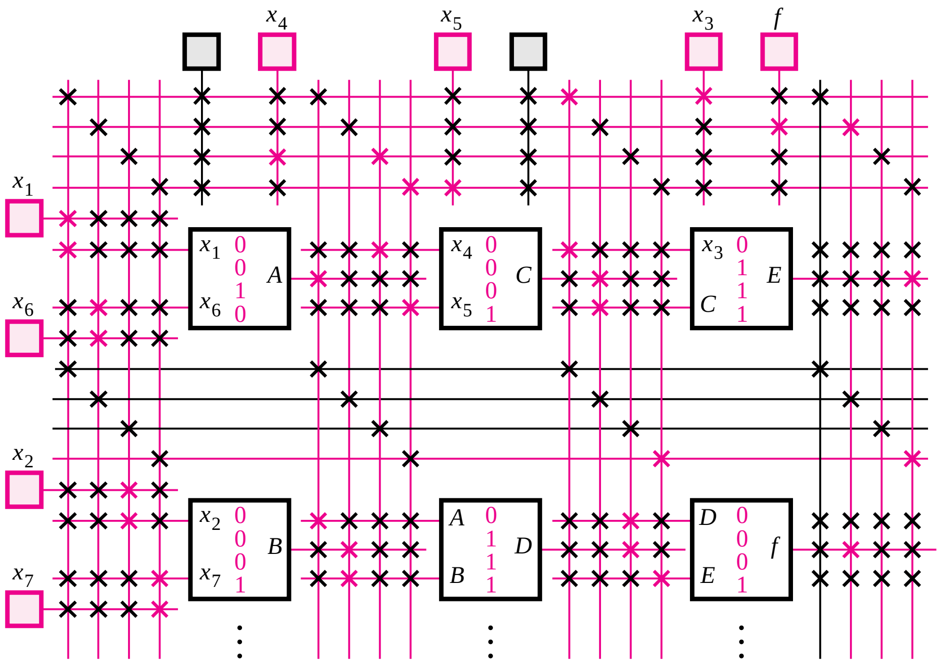


Figure 4.22 Implementation in an FPGA.

Fan-in Problem

In the preceding example, the fan-in restrictions were caused by the fixed structure of the FPGA, where each LUT has only two inputs. However, even when the target chip architecture is not fixed, the fan-in may still be an issue. To illustrate this situation, let us consider the implementation of a circuit in a custom chip. Recall that custom chips usually contain a large number of gates. If the chip is fabricated using CMOS technology, then there will be fan-in limitations as discussed in section 3.8.8. In this technology the number of inputs to a logic gate should be small. For instance, we may wish to limit the number of inputs to an AND gate to be less than five. Under this restriction, if a logic expression includes a seven-input product term, we would have to use 2 four-input AND gates, as indicated in Figure 4.23.

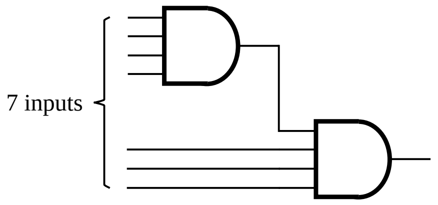


Figure 4.23 Using four-input AND gates to realize a seven-input product term.

Factoring can be used to deal with the fan-in problem. Suppose again that the available gates have a maximum fan-in of four and that we want to realize the function

$$f = x_1\overline{x}_2x_3\overline{x}_4x_5x_6 + x_1x_2\overline{x}_3\overline{x}_4\overline{x}_5x_6$$

This is a minimal sum-of-products expression. Using the approach of Figure 4.23, we will need four AND gates and one OR gate to implement this expression. A better solution is to factor the expression as follows

$$f = x_1\overline{x}_4x_6(\overline{x}_2x_3x_5 + x_2\overline{x}_3\overline{x}_5)$$

Then three AND gates and one OR gate suffice for realization of the required function, as shown in Figure 4.24.

Example 4.5 In practical situations a designer of logic circuits often encounters specifications that naturally lead to an initial design where the logic expressions are in a factored form. Suppose we need a circuit that meets the following requirements. There are four inputs: x_1, x_2, x_3, and x_4. An output, f_1, must have the value 1 if at least one of the inputs x_1 and x_2 is equal to 1 and both x_3 and x_4 are equal to 1; it must also be 1 if $x_1 = x_2 = 0$ and either x_3 or x_4 is 1. In all other cases $f_1 = 0$. A different output, f_2, is to be equal to 1 in all cases except when both x_1 and x_2 are equal to 0 or when both x_3 and x_4 are equal to 0.

From this specification, the function f_1 can be expressed as

$$f_1 = (x_1 + x_2)x_3x_4 + \overline{x}_1\overline{x}_2(x_3 + x_4)$$

This expression can be simplified to

$$f_1 = x_3x_4 + \overline{x}_1\overline{x}_2(x_3 + x_4)$$

which the reader can verify by using a Karnaugh map.

The second function, f_2, is most easily defined in terms of its complement, such that

$$\overline{f}_2 = \overline{x}_1\overline{x}_2 + \overline{x}_3\overline{x}_4$$

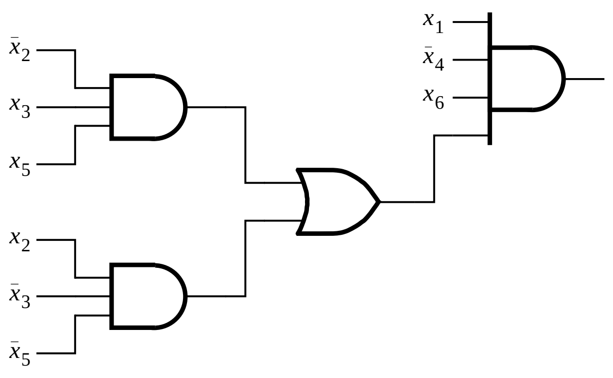


Figure 4.24 A factored circuit.

Then using DeMorgan's theorem gives

$$f_2 = (x_1 + x_2)(x_3 + x_4)$$

which is the minimum-cost expression for f_2; the cost increases significantly if the SOP form is used.

Because our objective is to design the lowest-cost combined circuit that implements f_1 and f_2, it seems that the best result can be achieved if we use the factored forms for both functions, in which case the sum term $(x_3 + x_4)$ can be shared. Moreover, observing that $\overline{x}_1\overline{x}_2 = \overline{x_1 + x_2}$, the sum term $(x_1 + x_2)$ can also be shared if we express f_1 in the form

$$f_1 = x_3x_4 + \overline{x_1 + x_2}(x_3 + x_4)$$

Then the combined circuit, shown in Figure 4.25, comprises three OR gates, three AND gates, one NOT gate, and 13 inputs, for a total of 20.

Impact on Wiring Complexity

The space on integrated circuit chips is occupied by the circuitry that implements logic gates and by the wires needed to make connections among the gates. The amount of space needed for wiring is a substantial portion of the chip area. Therefore, it is useful to keep the wiring complexity as low as possible.

In a logic expression each literal corresponds to a wire in the circuit that carries the desired logic signal. Since factoring usually reduces the number of literals, it provides a powerful mechanism for reducing the wiring complexity in a logic circuit. In the synthesis process the CAD tools consider many different issues, including the cost of the circuit, the fan-in, and the wiring complexity.

4.7.2 Functional Decomposition

In the preceding examples, which illustrated the factoring approach, multilevel circuits were used to deal with fan-in limitations. However, such circuits may be preferable to their two-level equivalents even if fan-in is not a problem. In some cases the multilevel circuits

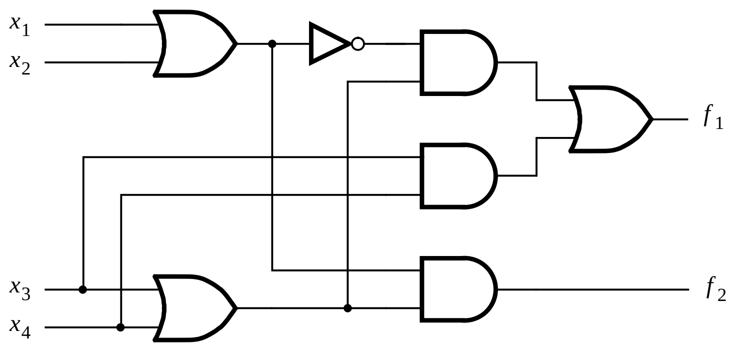


Figure 4.25 Circuit for Example 4.5.

may reduce the cost of implementation. On the other hand, they usually imply longer propagation delays, because they use multiple stages of logic gates. We will explore these issues by means of illustrative examples.

Complexity of a logic circuit, in terms of wiring and logic gates, can often be reduced by *decomposing* a two-level circuit into subcircuits, where one or more subcircuits implement functions that may be used in several places to construct the final circuit. To achieve this objective, a two-level logic expression is replaced by two or more new expressions, which are then combined to define a multilevel circuit. We can illustrate this idea by a simple example.

Example 4.6 Consider the minimum-cost sum-of-products expression

$$f = \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 x_4 + \overline{x}_1 \overline{x}_2 x_4$$

and assume that the inputs x_1 to x_4 are available only in their true form. Then the expression defines a circuit that has four AND gates, one OR gate, two NOT gates, and 18 inputs (wires) to all gates. The fan-in is three for the AND gates and four for the OR gate. The reader should observe that in this case we have included the cost of NOT gates needed to complement x_1 and x_2, rather than assume that both true and complemented versions of all input variables are available, as we had done before.

Factoring x_3 from the first two terms and x_4 from the last two terms, this expression becomes

$$f = (\overline{x}_1 x_2 + x_1 \overline{x}_2) x_3 + (x_1 x_2 + \overline{x}_1 \overline{x}_2) x_4$$

Now let $g(x_1, x_2) = \overline{x}_1 x_2 + x_1 \overline{x}_2$ and observe that

$$\begin{aligned}
\overline{g} &= \overline{\overline{x}_1 x_2 + x_1 \overline{x}_2} \\
&= \overline{\overline{x}_1 x_2} \cdot \overline{x_1 \overline{x}_2} \\
&= (x_1 + \overline{x}_2)(\overline{x}_1 + x_2) \\
&= x_1 \overline{x}_1 + x_1 x_2 + \overline{x}_2 \overline{x}_1 + \overline{x}_2 x_2 \\
&= 0 + x_1 x_2 + \overline{x}_1 \overline{x}_2 + 0 \\
&= x_1 x_2 + \overline{x}_1 \overline{x}_2
\end{aligned}$$

Then f can be written as

$$f = g x_3 + \overline{g} x_4$$

which leads to the circuit shown in Figure 4.26. This circuit requires an additional OR gate and a NOT gate to invert the value of g. But it needs only 15 inputs. Moreover, the largest fan-in has been reduced to two. The cost of this circuit is lower than the cost of its two-level equivalent. The trade-off is an increased propagation delay because the circuit has three more levels of logic.

In this example the subfunction g is a function of variables x_1 and x_2. The subfunction is used as an input to the rest of the circuit that completes the realization of the required function f. Let h denote the function of this part of the circuit, which depends on only three

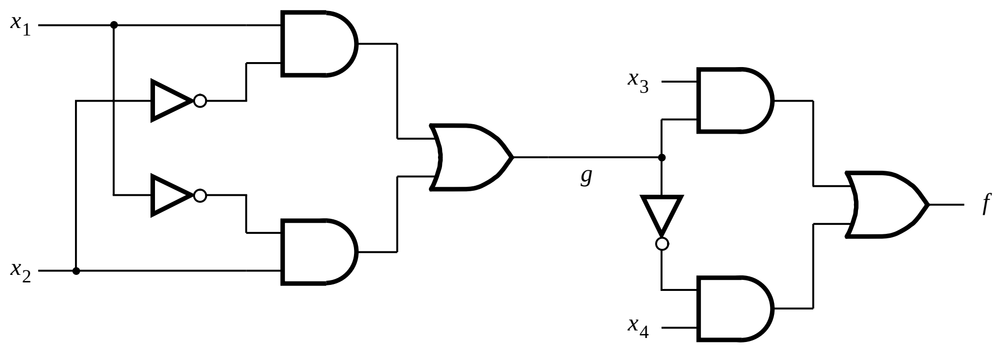


Figure 4.26 Logic circuit for Example 4.6.

inputs: g, x_3, and x_4. Then the decomposed realization of f can be expressed algebraically as

$$f(x_1, x_2, x_3, x_4) = h[g(x_1, x_2), x_3, x_4]$$

The structure of this decomposition can be described in block-diagram form as shown in Figure 4.27.

While not evident from our first example, functional decomposition can lead to great reductions in the complexity and cost of circuits. The reader will get a good indication of this benefit from the next example.

Example 4.7

Figure 4.28*a* defines a five-variable function f in the form of a Karnaugh map. In searching for a good decomposition for this function, it is necessary to first identify the variables that will be used as inputs to a subfunction. We can get a useful clue from the patterns of 1s in

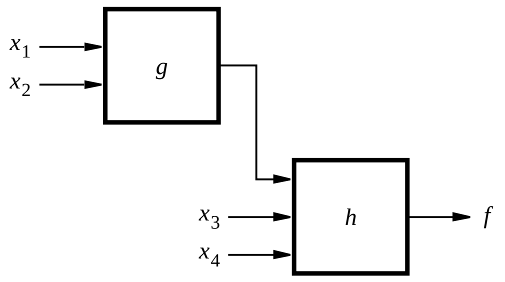
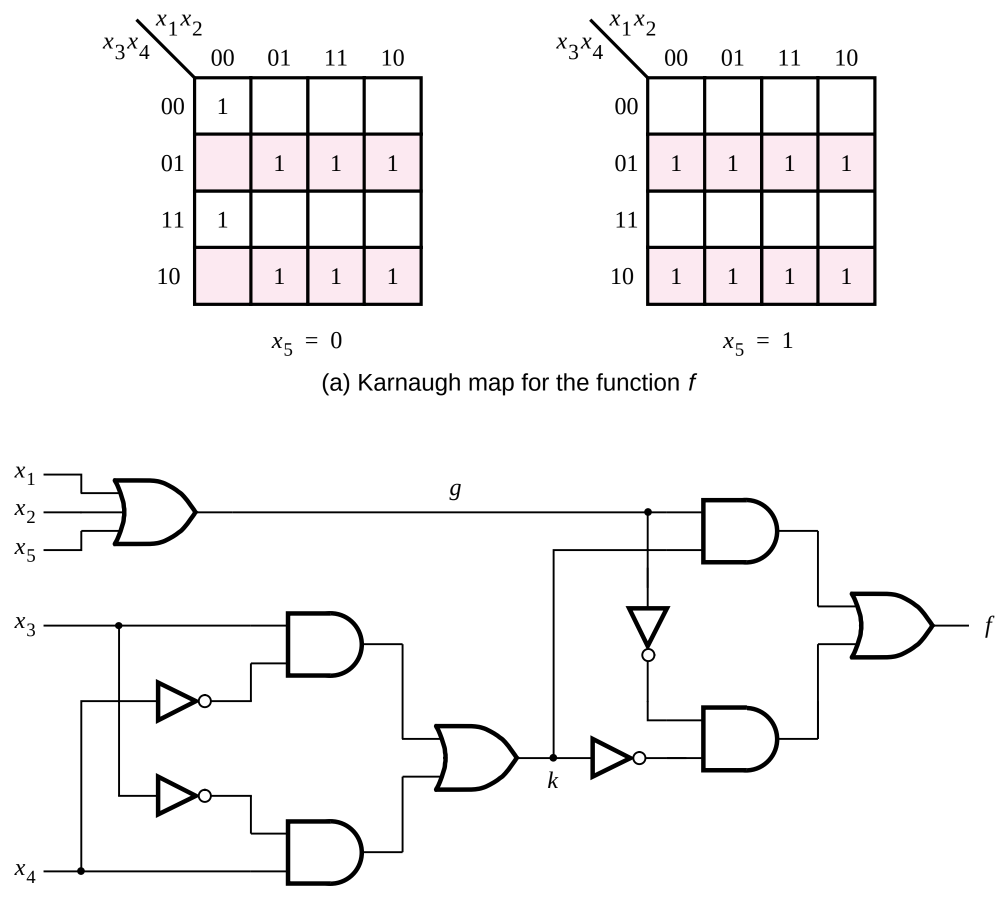


Figure 4.27 The structure of decomposition in Example 4.6.



$x_3x_4 \backslash x_1x_2$	00	01	11	10
00	1			
01		1	1	1
11	1			
10		1	1	1

$x_5 = 0$

$x_3x_4 \backslash x_1x_2$	00	01	11	10
00				
01	1	1	1	1
11				
10	1	1	1	1

$x_5 = 1$

(a) Karnaugh map for the function f

(b) Circuit obtained using decomposition

Figure 4.28 Decomposition for Example 4.7.

the map. Note that there are only two distinct patterns in the rows of the map. The second and fourth rows have one pattern, highlighted in blue, while the first and second rows have the other pattern. Once we specify which row each pattern is in, then the pattern itself depends only on the variables that define columns in each row, namely, x_1, x_2, and x_5. Let a subfunction $g(x_1, x_2, x_5)$ represent the pattern in rows 2 and 4. This subfunction is just

$$g = x_1 + x_2 + x_5$$

because the pattern has a 1 wherever any of these variables is equal to 1. To specify the location of rows where the pattern g occurs, we use the variables x_3 and x_4. The terms $\overline{x}_3x_4$ and $x_3\overline{x}_4$ identify the second and fourth rows, respectively. Thus the expression $(\overline{x}_3x_4 + x_3\overline{x}_4) \cdot g$ represents the part of f that is defined in rows 2 and 4.

Next, we have to find a realization for the pattern in rows 1 and 3. This pattern has a 1 only in the cell where $x_1 = x_2 = x_5 = 0$, which corresponds to the term $\overline{x}_1\overline{x}_2\overline{x}_5$. But we can make a useful observation that this term is just a complement of g. The location of rows 1 and 3 is identified by terms $\overline{x}_3\overline{x}_4$ and x_3x_4, respectively. Thus the expression $(\overline{x}_3\overline{x}_4 + x_3x_4) \cdot \overline{g}$ represents f in rows 1 and 3.

We can make one other useful observation. The expressions $(\overline{x}_3x_4 + x_3\overline{x}_4)$ and $(\overline{x}_3\overline{x}_4 + x_3x_4)$ are complements of each other, as shown in Example 4.6. Therefore, if we let $k(x_3, x_4) = \overline{x}_3x_4 + x_3\overline{x}_4$, the complete decomposition of f can be stated as

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) &= h[g(x_1, x_2, x_5), k(x_3, x_4)] \\ &= kg + \overline{k}\overline{g} \end{aligned}$$

where

$$\begin{aligned} g &= x_1 + x_2 + x_5 \\ k &= \overline{x}_3x_4 + x_3\overline{x}_4 \end{aligned}$$

The resulting circuit is given in Figure 4.28*b*. It requires a total of 11 gates and 19 inputs. The largest fan-in is three.

For comparison, a minimum-cost sum-of-products expression for f is

$$f = x_1\overline{x}_3x_4 + x_1x_3\overline{x}_4 + x_2\overline{x}_3x_4 + x_2x_3\overline{x}_4 + \overline{x}_3x_4x_5 + x_3\overline{x}_4x_5 + \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4\overline{x}_5 + \overline{x}_1\overline{x}_2x_3x_4\overline{x}_5$$

The corresponding circuit requires a total of 14 gates (including the five NOT gates to complement the primary inputs) and 41 inputs. The fan-in for the output OR gate is eight. Obviously, functional decomposition results in a much simpler implementation of this function.

In both of the preceding examples, the decomposition is such that a decomposed subfunction depends on some primary input variables, whereas the remainder of the implementation depends on the rest of the variables. Such decompositions are called *disjoint decompositions* in the technical literature. It is possible to have a *non-disjoint decomposition*, where the variables of the subfunction are also used in realizing the remainder of the circuit. The following example illustrates this possibility.

Example 4.8

Exclusive-OR (XOR) is a very useful function. In section 3.9.1 we showed how it can be realized using a special circuit. It can also be realized using AND and OR gates as shown in Figure 4.29*a*. In section 4.6 we explained how any AND-OR circuit can be realized as a NAND-NAND circuit that has the same structure.

Let us now try to exploit functional decomposition to find a better implementation of XOR using only NAND gates. Let the symbol $\uparrow$ represent the NAND operation so that $x_1 \uparrow x_2 = \overline{x_1 \cdot x_2}$. A sum-of-products expression for the XOR function is

$$x_1 \oplus x_2 = x_1\overline{x}_2 + \overline{x}_1x_2$$

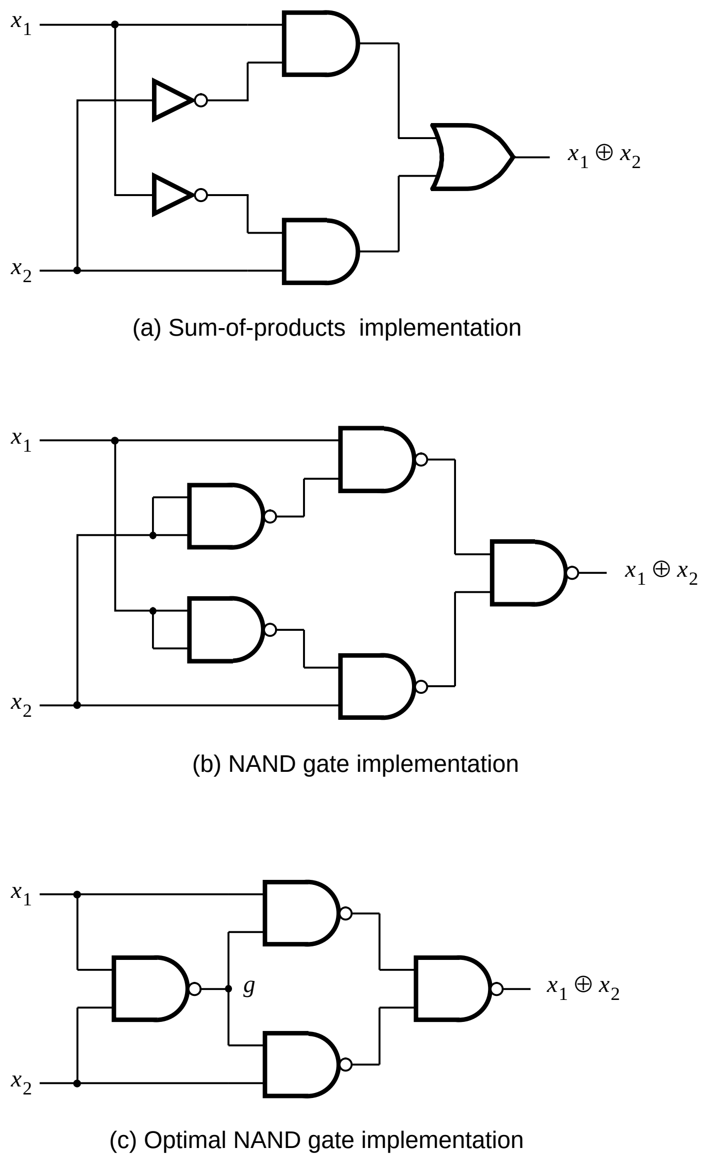

x_1
x_2
$x_1 \oplus x_2$
(a) Sum-of-products implementation
x_1
x_2
$x_1 \oplus x_2$
(b) NAND gate implementation
x_1
x_2
g
$x_1 \oplus x_2$
(c) Optimal NAND gate implementation

Figure 4.29 Implementation of XOR.

From the discussion in section 4.6, this expression can be written in terms of NAND operations as

$$x_1 \oplus x_2 = (x_1 \uparrow \overline{x}_2) \uparrow (\overline{x}_1 \uparrow x_2)$$

This expression requires five NAND gates, and it is implemented by the circuit in Figure 4.29*b*. Observe that an inverter is implemented using a two-input NAND gate by tying the two inputs together.

To find a decomposition, we can manipulate the term $(x_1 \uparrow \overline{x}_2)$ as follows:

$$(x_1 \uparrow \overline{x}_2) = \overline{(x_1\overline{x}_2)} = \overline{(x_1(\overline{x}_1 + \overline{x}_2))} = (x_1 \uparrow (\overline{x}_1 + \overline{x}_2))$$

We can perform a similar manipulation for $(\overline{x}_1 \uparrow x_2)$ to generate

$$x_1 \oplus x_2 = (x_1 \uparrow (\overline{x}_1 + \overline{x}_2)) \uparrow ((\overline{x}_1 + \overline{x}_2) \uparrow x_2)$$

DeMorgan's theorem states that $\overline{x}_1 + \overline{x}_2 = x_1 \uparrow x_2$; hence we can write

$$x_1 \oplus x_2 = (x_1 \uparrow (x_1 \uparrow x_2)) \uparrow ((x_1 \uparrow x_2) \uparrow x_2)$$

Now we have a decomposition

$$x_1 \oplus x_2 = (x_1 \uparrow g) \uparrow (g \uparrow x_2)$$

$$g = x_1 \uparrow x_2$$

The corresponding circuit, which requires only four NAND gates, is given in Figure 4.29*c*.

Practical Issues

Functional decomposition is a powerful technique for reducing the complexity of circuits. It can also be used to implement general logic functions in circuits that have built-in constraints. For example, in programmable logic devices (PLDs) that were introduced in Chapter 3 it is necessary to "fit" a desired logic circuit into logic blocks that are available on these devices. The available blocks are a target for decomposed subfunctions that may be used to realize larger functions.

A big problem in functional decomposition is finding the possible subfunctions. For functions of many variables, an enormous number of possibilities should be tried. This situation precludes attempts at finding optimal solutions. Instead, heuristic approaches that lead to acceptable solutions are used.

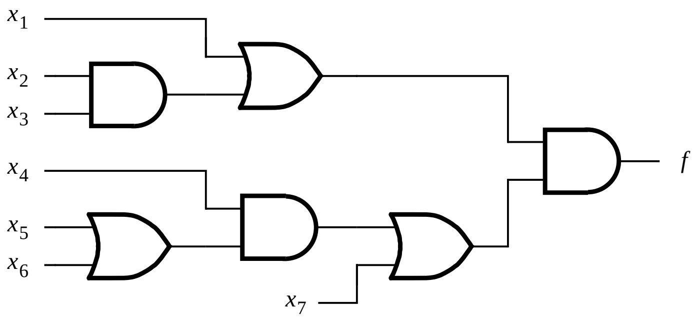
Full discussion of functional decomposition and factoring is beyond the scope of this book. An interested reader may consult other references [2–5]. Modern CAD tools use the concept of decomposition extensively.

4.7.3 MULTILEVEL NAND AND NOR CIRCUITS

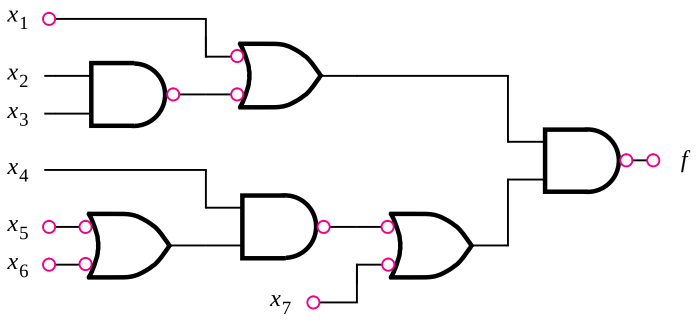
In section 4.6 we showed that two-level circuits consisting of AND and OR gates can be easily converted into circuits that can be realized with NAND and NOR gates, using the same gate arrangement. In particular, an AND-OR (sum-of products) circuit can be realized as a NAND-NAND circuit, while an OR-AND (product-of-sums) circuit becomes a NOR-NOR circuit. The same conversion approach can be used for multilevel circuits. We will illustrate this approach by an example.

Example 4.9

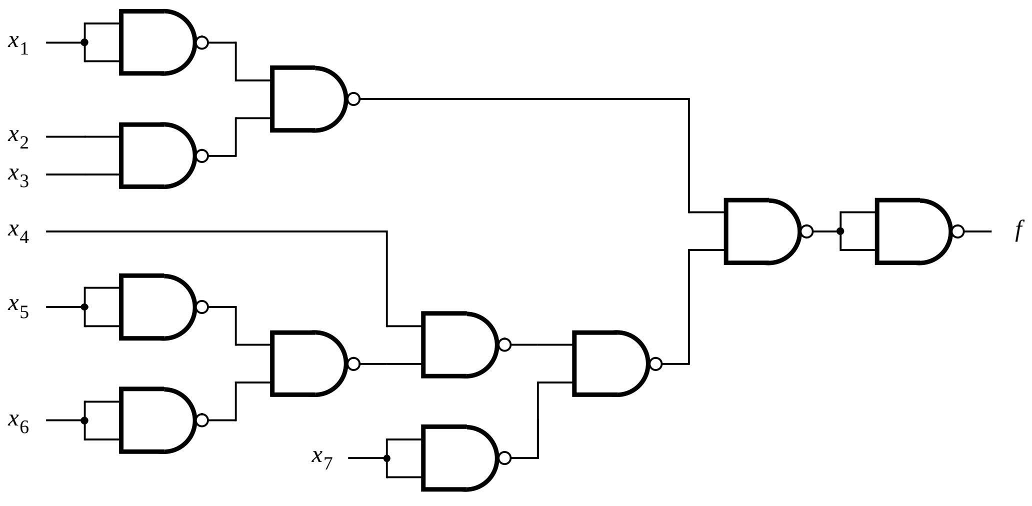
Figure 4.30*a* gives a four-level circuit consisting of AND and OR gates. Let us first derive a functionally equivalent circuit that comprises only NAND gates. Each AND gate is converted to a NAND by inverting its output. Each OR gate is converted to a NAND by



(a) Circuit with AND and OR gates



(b) Inversions needed to convert to NANDs

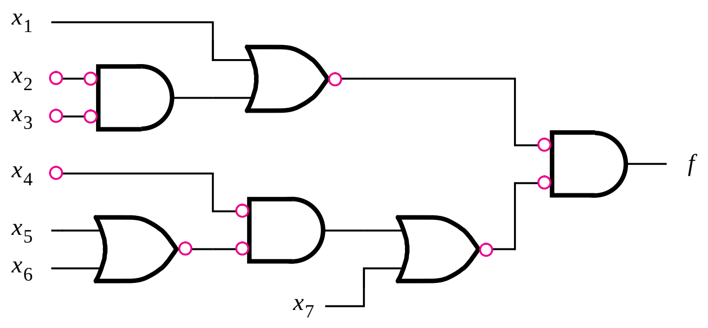


(c) NAND-gate circuit

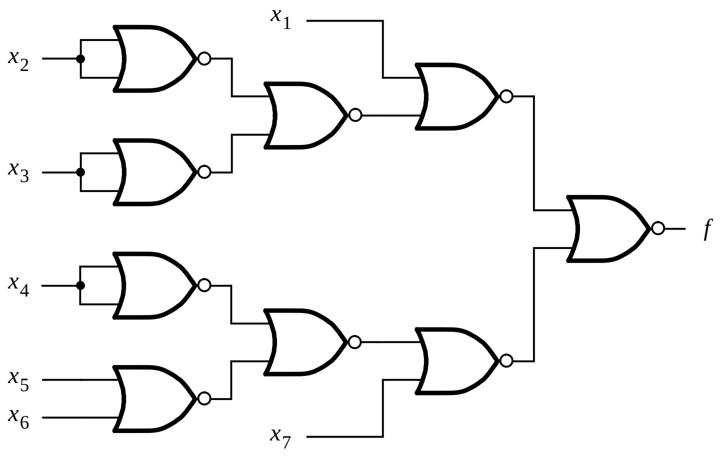
Figure 4.30 Conversion to a NAND-gate circuit.

inverting its inputs. This is just an application of DeMorgan's theorem, as illustrated in Figure 4.18*a*. Figure 4.30*b* shows the necessary inversions in blue. Note that an inversion is applied at both ends of a given wire. Now each gate becomes a NAND gate. This accounts for most of the inversions added to the original circuit. But, there are still four inversions that are not a part of any gate; therefore, they must be implemented separately. These inversions are at inputs x_1, x_5, and x_6 and at the output f. They can be implemented as two-input NAND gates, where the inputs are tied together. The resulting circuit is shown in Figure 4.30*c*.

A similar approach can be used to convert the circuit in Figure 4.30*a* into a circuit that comprises only NOR gates. An OR gate is converted to a NOR gate by inverting its output. An AND becomes a NOR if its inputs are inverted, as indicated in Figure 4.18*b*. Using this approach, the inversions needed for our sample circuit are shown in blue in Figure 4.31*a*.



(a) Inversions needed to convert to NORs



(b) NOR-gate circuit

Figure 4.31 Conversion to a NOR-gate circuit.

Then each gate becomes a NOR gate. The three inversions at inputs x_2, x_3, and x_4 can be realized as two-input NOR gates, where the inputs are tied together. The resulting circuit is presented in Figure 4.31*b*.

It is evident that the basic topology of a circuit does not change substantially when converting from AND and OR gates to either NAND or NOR gates. However, it may be necessary to insert additional gates to serve as NOT gates that implement inversions not absorbed as a part of other gates in the circuit.

4.8 ANALYSIS OF MULTILEVEL CIRCUITS

The preceding section showed that it may be advantageous to implement logic functions using multilevel circuits. It also presented the most commonly used approaches for synthesizing functions in this way. In this section we will consider the task of analyzing an existing circuit to determine the function that it implements.

For two-level circuits the analysis process is simple. If a circuit has an AND-OR (NAND-NAND) structure, then its output function can be written in the SOP form by inspection. Similarly, it is easy to derive a POS expression for an OR-AND (NOR-NOR) circuit. The analysis task is more complicated for multilevel circuits because it is difficult to write an expression for the function by inspection. We have to derive the desired expression by tracing the circuit and determining its functionality. The tracing can be done either starting from the input side and working towards the output, or by starting at the output side and working back towards the inputs. At intermediate points in the circuit, it is necessary to evaluate the subfunctions realized by the logic gates.

Example 4.10 Figure 4.32 replicates the circuit from Figure 4.30*a*. To determine the function f implemented by this circuit, we can consider the functionality at internal points that are the outputs

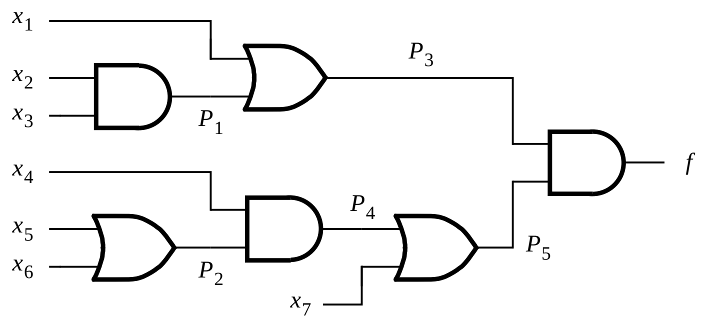


Figure 4.32 Circuit for Example 4.10.

of various gates. These points are labeled P_1 to P_5 in the figure. The functions realized at these points are

$$\begin{aligned}
P_1 &= x_2x_3 \\
P_2 &= x_5 + x_6 \\
P_3 &= x_1 + P_1 = x_1 + x_2x_3 \\
P_4 &= x_4P_2 = x_4(x_5 + x_6) \\
P_5 &= P_4 + x_7 = x_4(x_5 + x_6) + x_7
\end{aligned}$$

Then f can be evaluated as

$$\begin{aligned}
f &= P_3P_5 \\
&= (x_1 + x_2x_3)(x_4(x_5 + x_6) + x_7)
\end{aligned}$$

Applying the distributive property to eliminate the parentheses gives

$$f = x_1x_4x_5 + x_1x_4x_6 + x_1x_7 + x_2x_3x_4x_5 + x_2x_3x_4x_6 + x_2x_3x_7$$

Note that the expression represents a circuit comprising six AND gates, one OR gate, and 25 inputs. The cost of this two-level circuit is higher than the cost of the circuit in Figure 4.32, but the circuit has lower propagation delay.

Example 4.11

In Example 4.7 we derived the circuit in Figure 4.28*b*. In addition to AND gates and OR gates, the circuit has some NOT gates. It is reproduced in Figure 4.33, and the internal points are labeled from P_1 to P_{10} as shown. The following subfunctions occur

$$\begin{aligned}
P_1 &= x_1 + x_2 + x_5 \\
P_2 &= \overline{x}_4 \\
P_3 &= \overline{x}_3 \\
P_4 &= x_3P_2 \\
P_5 &= x_4P_3 \\
P_6 &= P_4 + P_5 \\
P_7 &= \overline{P}_1 \\
P_8 &= \overline{P}_6 \\
P_9 &= P_1P_6 \\
P_{10} &= P_7P_8
\end{aligned}$$

We can derive f by tracing the circuit from the output towards the inputs as follows

$$\begin{aligned}
f &= P_9 + P_{10} \\
&= P_1P_6 + P_7P_8
\end{aligned}$$

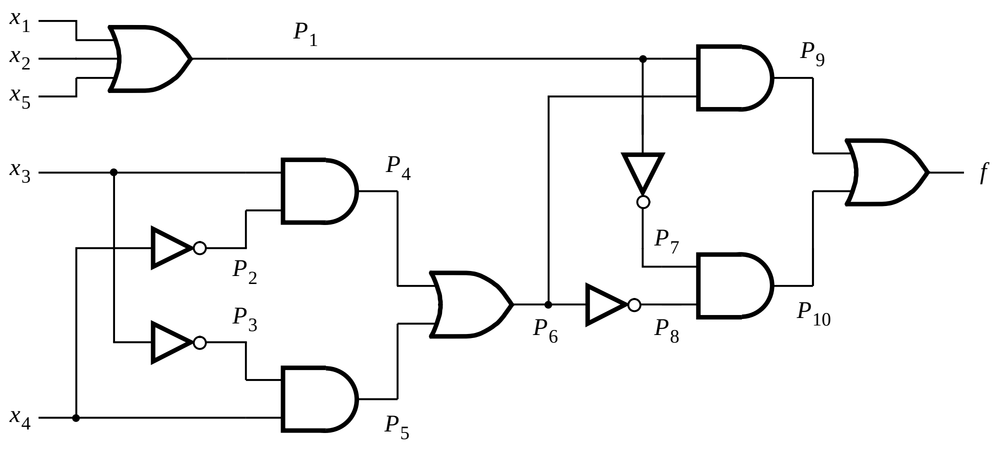


Figure 4.33 Circuit for Example 4.11.

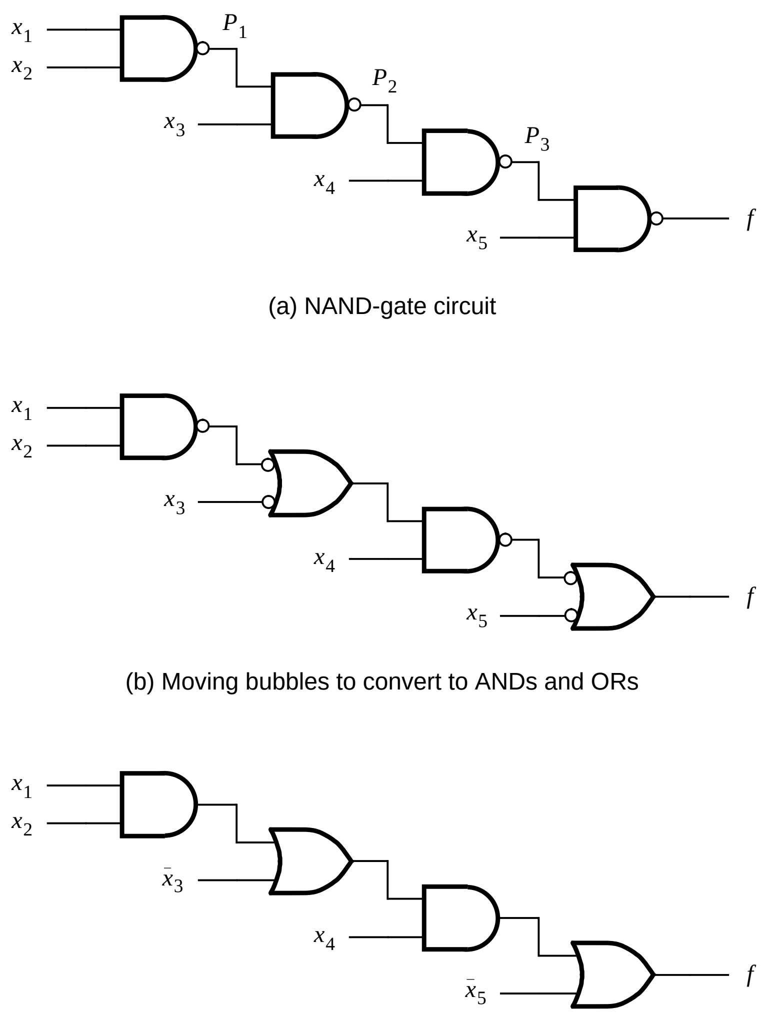
$$\begin{aligned}
&= (x_1 + x_2 + x_5)(P_4 + P_5) + \overline{P}_1\overline{P}_6 \\
&= (x_1 + x_2 + x_5)(x_3P_2 + x_4P_3) + \overline{x}_1\overline{x}_2\overline{x}_5\overline{P}_4\overline{P}_5 \\
&= (x_1 + x_2 + x_5)(x_3\overline{x}_4 + x_4\overline{x}_3) + \overline{x}_1\overline{x}_2\overline{x}_5(\overline{x}_3 + \overline{P}_2)(\overline{x}_4 + \overline{P}_3) \\
&= (x_1 + x_2 + x_5)(x_3\overline{x}_4 + \overline{x}_3x_4) + \overline{x}_1\overline{x}_2\overline{x}_5(\overline{x}_3 + x_4)(\overline{x}_4 + x_3) \\
&= x_1x_3\overline{x}_4 + x_1\overline{x}_3x_4 + x_2x_3\overline{x}_4 + x_2\overline{x}_3x_4 + x_5x_3\overline{x}_4 + x_5\overline{x}_3x_4 + \\
&\quad \overline{x}_1\overline{x}_2\overline{x}_5\overline{x}_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_5x_4x_3
\end{aligned}$$

This is the same expression as stated in Example 4.7.

Example 4.12 Circuits based on NAND and NOR gates are slightly more difficult to analyze because each gate involves an inversion. Figure 4.34*a* depicts a simple NAND-gate circuit that illustrates the effect of inversions. We can convert this circuit into a circuit with AND and OR gates using the reverse of the approach described in Example 4.9. Bubbles that denote inversions can be moved, according to DeMorgan's theorem, as indicated in Figure 4.34*b*. Then the circuit can be converted into the circuit in part (*c*) of the figure, which consists of AND and OR gates. Observe that in the converted circuit, the inputs x_3 and x_5 are complemented. From this circuit the function f is determined as

$$\begin{aligned}
f &= (x_1x_2 + \overline{x}_3)x_4 + \overline{x}_5 \\
&= x_1x_2x_4 + \overline{x}_3x_4 + \overline{x}_5
\end{aligned}$$

It is not necessary to convert a NAND circuit into a circuit with AND and OR gates to determine its functionality. We can use the approach from Examples 4.10 and 4.11 to



(a) NAND-gate circuit

(b) Moving bubbles to convert to ANDs and ORs

(c) Circuit with AND and OR gates

Figure 4.34 Circuit for Example 4.12.

derive f as follows. Let P_1, P_2, and P_3 label the internal points as shown in Figure 4.34*a*. Then

$$P_1 = \overline{x_1 x_2}$$

$$P_2 = \overline{P_1 x_3}$$

$$P_3 = \overline{P_2 x_4}$$

$$\begin{aligned}
f &= \overline{P_3 x_5} = \overline{P}_3 + \overline{x}_5 \\
&= \overline{\overline{P_2 x_4}} + \overline{x}_5 = P_2 x_4 + \overline{x}_5 \\
&= \overline{P_1 x_3} x_4 + \overline{x}_5 = (\overline{P}_1 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (\overline{\overline{x_1 x_2}} + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (x_1 x_2 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= x_1 x_2 x_4 + \overline{x}_3 x_4 + \overline{x}_5
\end{aligned}$$

Example 4.13 The circuit in Figure 4.35 consists of NAND and NOR gates. It can be analyzed as follows.

$$\begin{aligned}
P_1 &= \overline{x_2 x_3} \\
P_2 &= \overline{x_1 P_1} = \overline{x}_1 + \overline{P}_1 \\
P_3 &= \overline{x_3 x_4} = \overline{x}_3 + \overline{x}_4 \\
P_4 &= \overline{P_2 + P_3} \\
f &= \overline{P_4 + x_5} = \overline{P}_4 \overline{x}_5 \\
&= \overline{\overline{P_2 + P_3}} \cdot \overline{x}_5 \\
&= (P_2 + P_3) \overline{x}_5 \\
&= (\overline{x}_1 + \overline{P}_1 + \overline{x}_3 + \overline{x}_4) \overline{x}_5 \\
&= (\overline{x}_1 + x_2 x_3 + \overline{x}_3 + \overline{x}_4) \overline{x}_5 \\
&= (\overline{x}_1 + x_2 + \overline{x}_3 + \overline{x}_4) \overline{x}_5 \\
&= \overline{x}_1 \overline{x}_5 + x_2 \overline{x}_5 + \overline{x}_3 \overline{x}_5 + \overline{x}_4 \overline{x}_5
\end{aligned}$$

Note that in deriving the second to the last line, we used property 16*a* in section 2.5 to simplify $x_2 x_3 + \overline{x}_3$ into $x_2 + \overline{x}_3$.

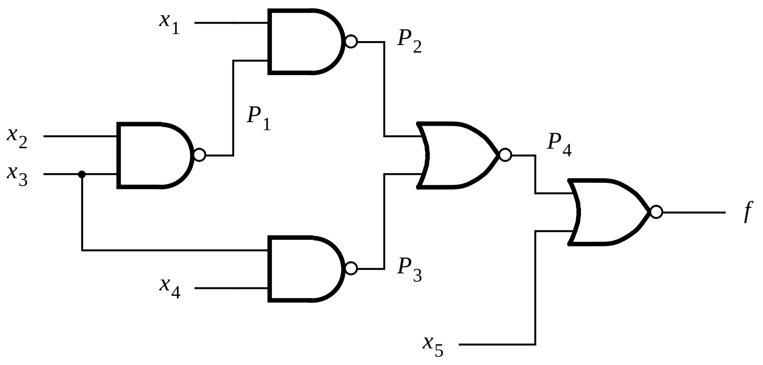


Figure 4.35 Circuit for Example 4.13.

April 13, 1999 12:52 g02-ch4 Sheet number 43 Page number 185 black

Analysis of circuits is much simpler than synthesis. With a little practice one can develop an ability to easily analyze even fairly complex circuits.

We have now covered a considerable amount of material on synthesis and analysis of logic functions. We have used the Karnaugh map as a vehicle for illustrating the concepts involved in finding optimal implementations of logic functions. We have also shown that logic functions can be realized in a variety of forms, both with two levels of logic and with multiple levels. In a modern design environment, logic circuits are synthesized using CAD tools, rather than by hand. The concepts that we have discussed in this chapter are quite general; they are representative of the strategies implemented in CAD algorithms. As we have said before, the Karnaugh map scheme for representing logic functions is not appropriate for use in CAD tools. In the next section we discuss an alternative representation of logic functions, which is suitable for use in CAD algorithms.

4.9 Cubical Representation

The Karnaugh map is an excellent vehicle for illustrating concepts, and it is even useful for manual design if the functions have only a few variables. To deal with larger functions it is necessary to have techniques that are algebraic, rather than graphical, which can be applied to functions of any number of variables.

Many algebraic optimization techniques have been developed. As early as the 1950s, a tabular approach proposed by Willard Quine [6] and Edward McCluskey [7] became popular under the name Quine-McCluskey method. Almost all textbooks on logic design discuss this method at length [8–18]. We will not do so because there exist more attractive alternatives that can be incorporated into CAD tools.

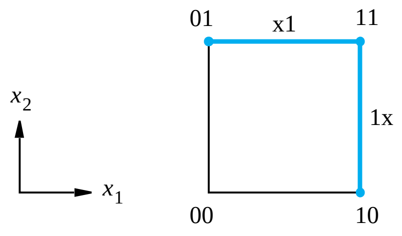
We will not pursue algebraic optimization techniques in great detail, but we will attempt to provide the reader with an appreciation of the tasks involved. This helps in gaining an understanding of what the CAD tools can do and what results can be expected from them. The approach that we will present makes use of a cubical representation of logic functions.

4.9.1 Cubes and Hypercubes

So far in this book, we have encountered four different forms for representing logic functions: truth tables, algebraic expressions, Venn diagrams, and Karnaugh maps. Another possibility is to map a function of n variables onto an n-dimensional cube.

Two-Dimensional Cube

A two-dimensional cube is shown in Figure 4.36. The four corners in the cube are called *vertices*, which correspond to the four rows of a truth table. Each vertex is identified by two coordinates. The horizontal coordinate is assumed to correspond to variable x_1, and vertical coordinate to x_2. Thus vertex 00 is the bottom-left corner, which corresponds to row 0 in the truth table. Vertex 01 is the top-left corner, where $x_1 = 0$ and $x_2 = 1$, which corresponds to row 1 in the truth table, and so on for the other two vertices.



x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4.36 Representation of $f(x_1, x_2) = \sum m(1, 2, 3)$.

We will map a function onto the cube by indicating with blue circles those vertices for which $f = 1$. In Figure 4.36 $f = 1$ for vertices 01, 10, and 11. We can express the function as a set of vertices, using the notation $f = \{01, 10, 11\}$. The function f is also shown in the form of a truth table in the figure.

An edge joins two vertices for which the labels differ in the value of only one variable. Therefore, if two vertices for which $f = 1$ are joined by an edge, then this edge represents that portion of the function just as well as the two individual vertices. For example, $f = 1$ for vertices 10 and 11. They are joined by the edge that is labeled 1x. It is customary to use the letter x to denote the fact that the corresponding variable can be either 0 or 1. Hence 1x means that $x_1 = 1$, while x_2 can be either 0 or 1. Similarly, vertices 01 and 11 are joined by the edge labeled x1, indicating that x_1 can be either 0 or 1, but $x_2 = 1$. The reader must not confuse the use of the letter x for this purpose, in contrast to the subscripted use where x_1 and x_2 refer to the variables.

Two vertices being represented by a single edge is the embodiment of the combining property 14*a* from section 2.5. The edge 1x is the logical sum of vertices 10 and 11. It essentially defines the term x_1, which is the sum of minterms $x_1\overline{x}_2$ and x_1x_2. The property 14*a* indicates that

$$x_1\overline{x}_2 + x_1x_2 = x_1$$

Therefore, finding edges for which $f = 1$ is equivalent to applying the combining property. Of course, this is also analogous to finding pairs of adjacent cells in a Karnaugh map for which $f = 1$.

The edges 1x and x1 define fully the function in Figure 4.36; hence we can represent the function as $f = \{1x, x1\}$. This corresponds to the logic expression

$$f = x_1 + x_2$$

which is also obvious from the truth table in the figure.

Three-Dimensional Cube

Figure 4.37 illustrates a three-dimensional cube. The x_1, x_2, and x_3 coordinates are as shown on the left. Each vertex is identified by a specific valuation of the three variables. The function f mapped onto the cube is the function from Figure 4.1, which was used in Figure 4.5*b*. There are five vertices for which $f = 1$, namely, 000, 010, 100, 101, and 110. These vertices are joined by the five edges shown in blue, namely, x00, 0x0, x10, 1x0,

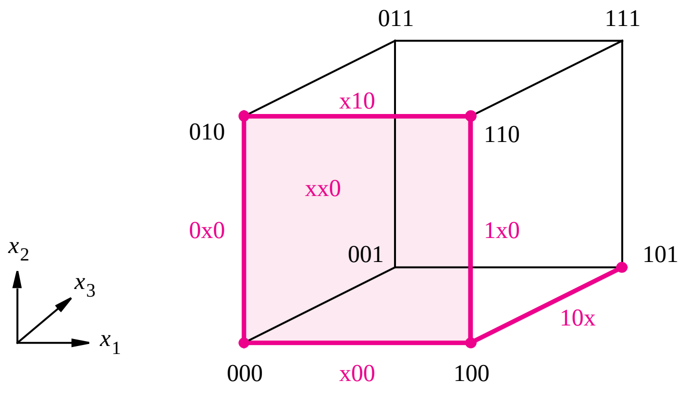


Figure 4.37 Representation of $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

and 10x. Because the vertices 000, 010, 100, and 110 include all valuations of x_1 and x_2, when x_3 is 0, they can be specified by the term xx0. This term means that $f = 1$ if $x_3 = 0$, regardless of the values of x_1 and x_2. Notice that xx0 represents the front side of the cube, which is shaded in blue.

From the preceding discussion it is evident that the function f can be represented in several ways. Some of the possibilities are

$$\begin{aligned} f &= \{000, 010, 100, 101, 110\} \\ &= \{0x0, 1x0, 101\} \\ &= \{x00, x10, 101\} \\ &= \{x00, x10, 10x\} \\ &= \{xx0, 10x\} \end{aligned}$$

In a physical realization each of the above terms is a product term implemented by an AND gate. Obviously, the least-expensive circuit is obtained if $f = \{xx0, 10x\}$, which is equivalent to the logic expression

$$f = \overline{x}_3 + x_1\overline{x}_2$$

This is the expression that we derived using the Karnaugh map in Figure 4.5*b*.

Four-Dimensional Cube

Graphical images of two- and three-dimensional cubes are easy to draw. A four-dimensional cube is more difficult. It consists of 2 three-dimensional cubes with their corners connected. The simplest way to visualize a four-dimensional cube is to have one cube placed inside the other cube, as depicted in Figure 4.38. We have assumed that the x_1, x_2, and x_3 coordinates are the same as in Figure 4.37, while $x_4 = 0$ defines the outer cube and $x_4 = 1$ defines the inner cube. Figure 4.38 indicates how the function f_3 of Figure 4.7 is mapped onto the four-dimensional cube. To avoid cluttering the figure with too many labels, we have labeled only those vertices for which $f_3 = 1$. Again, all edges that connect these vertices are highlighted in blue.

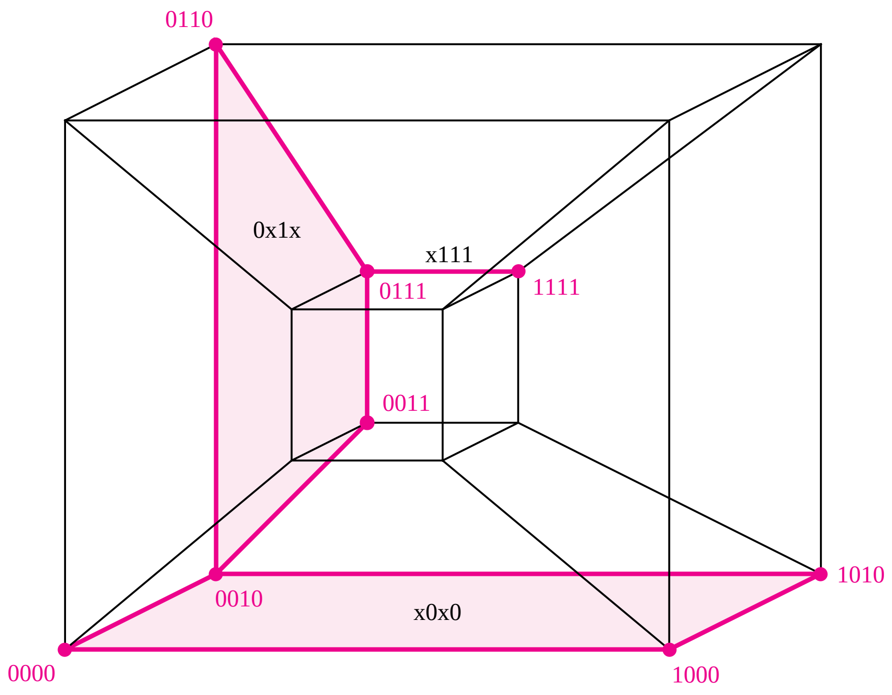


Figure 4.38 Representation of function f_3 from Figure 4.7.

There are two groups of four adjacent vertices for which $f_3 = 1$ that can be represented as planes. The group comprising 0000, 0010, 1000, and 1010 is represented by x0x0. The group 0010, 0011, 0110, and 0111 is represented by 0x1x. These planes are shaded in the figure. The function f_3 can be represented in several ways, for example

$$\begin{aligned} f_3 &= \{0000, 0010, 0011, 0110, 0111, 1000, 1010, 1111\} \\ &= \{00\text{x}0, 10\text{x}0, 0\text{x}10, 0\text{x}11, \text{x}111\} \\ &= \{\text{x}0\text{x}0, 0\text{x}1\text{x}, \text{x}111\} \end{aligned}$$

Since each x indicates that the corresponding variable can be ignored, because it can be either 0 or 1, the simplest circuit is obtained if $f = \{\text{x}0\text{x}0, 0\text{x}1\text{x}, \text{x}111\}$, which is equivalent to the expression

$$f_3 = \overline{x}_2\overline{x}_4 + \overline{x}_1x_3 + x_2x_3x_4$$

We derived the same expression in Figure 4.7.

n-Dimensional Cube

A function that has n variables can be mapped onto an n-dimensional cube. Although it is impractical to draw graphical images of cubes that have more than four variables, it is not difficult to extend the ideas introduced above to a general n-variable case. Because visual interpretation is not possible and because we normally use the word *cube* only for a three-dimensional structure, many people use the word *hypercube* to refer to structures with more than three dimensions. We will continue to use the word *cube* in our discussion.

It is convenient to refer to a cube as being of a certain *size* that reflects the number of vertices in the cube. Vertices have the smallest size. Each variable has a value of 0 or 1 in a vertex. A cube that has an x in one variable position is larger because it consists of two vertices. For example, the cube 1x01 consists of vertices 1001 and 1101. A cube that has two x's consists of four vertices, and so on. A cube that has k x's consists of 2^k vertices.

An n-dimensional cube has 2^n vertices. Two vertices are adjacent if they differ in the value of only one coordinate. Because there are n coordinates (axes in the n-dimensional cube), each vertex is adjacent to n other vertices. The n-dimensional cube contains cubes of lower dimensionality. Cubes of the lowest dimension are vertices. Because their dimension is zero, we will call them 0-*cubes*. Edges are cubes of dimension 1; hence we will call them 1-*cubes*. A side of a three-dimensional cube is a 2-*cube*. An entire three-dimensional cube is a 3-*cube*, and so on. In general, we will refer to a set of 2^k adjacent vertices as a k-*cube*.

From the examples in Figures 4.37 and 4.38, it is apparent that the largest possible k-*cubes* that exist for a given function are equivalent to its prime implicants. Next, we will describe a minimization technique that uses the cubical representation of functions.

4.10 Minimization Using Cubical Representation

Cubical representation of logic functions is well suited for implementation of minimization algorithms that can be programmed and run efficiently on computers. Such algorithms are included in modern CAD tools. While the CAD tools can be used effectively without detailed knowledge of how their minimization algorithms are implemented, the reader may find it interesting to gain some insight into how this may be accomplished. In this section we will outline a relatively simple algorithm, which illustrates the main concepts and indicates some of the problems that arise. A reader who intends to use the CAD tools, but is not interested in the details of automated minimization, may skip this section without loss of continuity.

4.10.1 Generation of Prime Implicants

As mentioned in section 4.9, the prime implicants of a given logic function f are the largest possible k-cubes for which $f = 1$. For incompletely specified functions, which include a set of don't-care vertices, the prime implicants are the largest k-cubes for which either $f = 1$ or f is unspecified.

In section 4.2.2 we presented a strategy for finding the minimum-cost sum-of-products form of a logic function. Assume that the initial specification of a function f is given in terms of implicants that are not necessarily either minterms or prime implicants. Then it is necessary to define an operation that will generate other implicants that are not given explicitly in the initial specification, but which will eventually lead to the prime implicants of f. One such possibility is known as the $*$-*product* operation, which is usually pronounced the "star-product" operation. We will refer to it simply as the $*$-*operation*.

$*$-Operation

The $*$-operation provides a simple way of deriving a new cube by combining two cubes that differ in the value of only one variable. Let $A = A_1A_2 \dots A_n$ and $B = B_1B_2 \dots B_n$ be two cubes that are implicants of an n-variable function. Thus each coordinate A_i and B_i is specified as having the value 0, 1, or x. There are two distinct steps in the $*$-operation. First, the $*$-operation is evaluated for each pair A_i and B_i, in coordinates $i = 1, 2, \dots, n$, according to the table in Figure 4.39. Then based on the results of using the table, a set of rules is applied to determine the overall result of the $*$-operation. The table in Figure 4.39 defines the coordinate $*$-operation, $A_i * B_i$. It specifies the result of $A_i * B_i$ for each possible combination of values of A_i and B_i. This result is the intersection (i.e., the common part) of A and B in this coordinate. Note that when A_i and B_i have the opposite values (0 and 1, or vice versa), the result of the coordinate $*$-operation is indicated by the symbol ø. We say that the intersection of A_i and B_i is empty. Using the table, the complete $*$-operation for A and B is defined as follows:

$$C = A * B, \text{ such that}$$

1. $C = ø$ *if* $A_i * B_i = ø$ for more than one i.
2. Otherwise, $C_i = A_i * B_i$ when $A_i * B_i \neq ø$, and $C_i = \text{x}$ for the coordinate where $A_i * B_i = ø$.

For example, let $A = \{0x0\}$ and $B = \{111\}$. Then $A_1 * B_1 = 0 * 1 = ø$, $A_2 * B_2 = x * 1 = 1$, and $A_3 * B_3 = 0 * 1 = ø$. Because the result is ø in two coordinates, it follows from condition 1 that $A * B = ø$. In other words, these two cubes cannot be combined into another cube, because they differ in two coordinates.

As another example, consider $A = \{11x\}$ and $B = \{10x\}$. In this case $A_1 * B_1 = 1 * 1 = 1$, $A_2 * B_2 = 1 * 0 = ø$, and $A_3 * B_3 = x * x = x$. According to condition 2 above, $C_1 = 1$, $C_2 = x$, and $C_3 = x$, which gives $C = A * B = \{1xx\}$. A larger 2-cube is created from two 1-cubes that differ in one coordinate only.

The result of the $*$-operation may be a smaller cube than the two cubes involved in the operation. Consider $A = \{1x1\}$ and $B = \{11x\}$. Then $C = A * B = \{111\}$. Notice that C is included in both A and B, which means that this cube will not be useful in searching for prime implicants. Therefore, it should be discarded by the minimization algorithm.

A_i \ B_i	0	1	x
0	0	ø	0
1	ø	1	1
x	0	1	x

$A_i * B_i$

Figure 4.39 The coordinate $*$-operation.

As a final example, consider $A = \{\text{x10}\}$ and $B = \{\text{0x1}\}$. Then $C = A * B = \{\text{01x}\}$. All three of these cubes are the same size, but C is not included in either A or B. Hence C has to be considered in the search for prime implicants. The reader may find it helpful to draw a Karnaugh map to see how cube C is related to cubes A and B.

Using the $*$-Operation to Find Prime Implicants

The essence of the $*$-operation is to find new cubes from pairs of existing cubes. In particular, it is of interest to find new cubes that are not included in the existing cubes. A procedure for finding the prime implicants may be organized as follows.

Suppose that a function f is specified by means of a set of implicants that are represented as cubes. Let this set be denoted as the cover C^k of f. Let c^i and c^j be any two cubes in C^k. Then apply the $*$-operation to all pairs of cubes in C^k; let G^{k+1} be the set of newly generated cubes. Hence

$$G^{k+1} = c^i * c^j \quad \text{for all } c^i, c^j \epsilon\ C^k$$

Now a new cover for f may be formed by using the cubes in C^k and G^{k+1}. Some of these cubes may be redundant because they are included in other cubes; they should be removed. Let the new cover be

$$C^{k+1} = C^k \cup G^{k+1} - \text{ redundant cubes}$$

where $\cup$ denotes the logical union of two sets, and the minus sign $(-)$ denotes the removal of elements of a set. If $C^{k+1} \neq C^k$, then a new cover C^{k+2} is generated using the same process. If $C^{k+1} = C^k$, then the cubes in the cover are the prime implicants of f. For an n-variable function, it is necessary to repeat the step at most n times.

Redundant cubes that have to be removed are identified through pairwise comparison of cubes. Cube $A = A_1A_2 \ldots A_n$ should be removed if it is included in some cube $B = B_1B_2 \ldots B_n$, which is the case if $A_i = B_i$ or $B_i = \text{x}$ for every coordinate i.

Example 4.14

Consider the function $f(x_1, x_2, x_3)$ of Figure 4.9. Assume that f is initially specified as a set of vertices that correspond to the minterms, m_0, m_1, m_2, m_3, and m_7. Hence let the initial cover be $C^0 = \{000, 001, 010, 011, 111\}$. Using the $*$-operation to generate a new set of cubes, we obtain $G^1 = \{\text{00x, 0x0, 0x1, 01x, x11}\}$. Then $C^1 = C^0 \cup G^1$ – redundant cubes. Observe that each cube in C^0 is included in one of the cubes in G^1; therefore, all cubes in C^0 are redundant. Thus $C^1 = G^1$.

The next step is to apply the $*$-operation to the cubes in C^1, which yields $G^2 = \{000, 001, \text{0xx, 0x1}, 010, \text{01x}, 011\}$. Note that all of these cubes are included in the cube 0xx; therefore, all but 0xx are redundant. Now it is easy to see that

$$\begin{aligned} C^2 &= C^1 \cup G^2 - \text{redundant terms} \\ &= \{\text{x11, 0xx}\} \end{aligned}$$

since all cubes of C^1, except x11, are redundant because they are covered by 0xx.

Applying the $*$-operation to C^2 yields $G^3 = \{011\}$ and

$$\begin{aligned} C^3 &= C^2 \cup G^3 - \text{redundant terms} \\ &= \{\text{x11, 0xx}\} \end{aligned}$$

Since $C^3 = C^2$, the conclusion is that the prime implicants of f are the cubes {x11, 0xx}, which represent the product terms x_2x_3 and $\overline{x}_1$. This is the same set of prime implicants that we derived using a Karnaugh map in Figure 4.9.

Example 4.15 As another example, consider the four-variable function of Figure 4.10. Assume that this function is initially specified as the cover C^0 = {0101, 1101, 1110, 011x, x01x}. Then successive applications of the $*$-operation and removing the redundant terms gives

$$\begin{aligned} C^1 &= \{\text{x01x, x101, 01x1, x110, 1x10, 0x1x}\} \\ C^2 &= \{\text{x01x, x101, 01x1, 0x1x, xx10}\} \\ C^3 &= C^2 \end{aligned}$$

Therefore, the prime implicants are $\overline{x}_2x_3$, $x_2\overline{x}_3x_4$, $\overline{x}_1x_2x_4$, $\overline{x}_1x_3$, and $x_3\overline{x}_4$.

4.10.2 Determination of Essential Prime Implicants

From a cover that consists of all prime implicants, it is necessary to extract a minimal cover. As we saw in section 4.2.2, all *essential* prime implicants must be included in the minimal cover. To find the essential prime implicants, it is useful to define an operation that determines a part of a cube (implicant) that is *not* covered by another cube. One such operation is called the *#-operation* (pronounced the "sharp operation"), which is defined as follows.

#-Operation

Again, let $A = A_1A_2 \ldots A_n$ and $B = B_1B_2 \ldots B_n$ be two cubes (implicants) of an n-variable function. The sharp operation $A\#B$ leaves as a result "that part of A that is not covered by B." Similar to the $*$-operation, the #-operation has two steps: $A_i\#B_i$ is evaluated for each coordinate i, and then a set of rules is applied to determine the overall result. The sharp operation for each coordinate is defined in Figure 4.40. After this operation is performed for all pairs (A_i, B_i), the complete #-operation is defined as follows:

$$C = A\#B, \text{ such that}$$

1. $C = A$ if $A_i\#B_i = \text{ø}$ for some i.
2. $C = \text{ø}$ if $A_i\#B_i = \varepsilon$ for all i.
3. Otherwise, $C = \bigcup_i (A_1, A_2, \ldots, \overline{B}_i, \ldots, A_n)$, where the union is for all i for which $A_i = \text{x}$ and $B_i \neq \text{x}$.

The first condition corresponds to the case where cubes A and B do not intersect at all; namely, A and B differ in the value of at least one variable, which means that no part of A is covered by B. For example, let A = 0x1 and B = 11x. The coordinate #-products are $A_1\#B_1 = \text{ø}$, $A_2\#B_2 = 0$, and $A_3\#B_3 = \varepsilon$. Then from rule 1 it follows that 0x1 # 11x = 0x1. The second condition reflects the case where A is fully covered by B. For example, 0x1

$A_i \backslash B_i$	0	1	x
0	ε	∅	ε
1	∅	ε	ε
x	1	0	ε

$A_i \# B_i$

Figure 4.40 The coordinate #-operation.

# 0xx = ø. The third condition is for the case where only a part of A is covered by B. In this case the #-operation generates one or more cubes. Specifically, it generates one cube for each coordinate i that is x in A_i, but is not x in B_i. Each cube generated is identical to A, except that A_i is replaced by $\overline{B}_i$. For example, 0xx # 01x = 00x, and 0xx # 010 = {00x, 0x1}.

We will now show how the #-operation can be used to find the essential prime implicants. Let P be the set of all prime implicants of a given function f. Let p^i denote one prime implicant in the set P. Also, let DC denote the don't-care vertices for f. Then p^i is an essential prime implicant if and only if

$$p^i \# (P - p^i) \# DC \neq ø$$

This means that p^i is essential if there exists at least one vertex for which $f = 1$ that is covered by p^i, but not by any other prime implicant. The #-operation is also performed with the set of don't-care cubes because vertices in p^i that correspond to don't-care conditions are not essential to cover. The meaning of $p^i \# (P - p^i)$ is that the #-operation is applied successively to each prime implicant in P. For example, consider $P = \{p^1, p^2, p^3, p^4\}$ and $DC = \{d^1, d^2\}$. To check whether p^3 is essential, we evaluate

$$((((p^3 \# p^1) \# p^2) \# p^4) \# d^1) \# d^2$$

If the result of this expression is not ø, then p^3 is essential.

Example 4.16

In Example 4.14 we determined that the cubes x11 and 0xx are the prime implicants of the function f in Figure 4.9. We can discover whether each of these prime implicants is essential as follows

$$\text{x11 \# 0xx} = 111 \neq ø$$
$$\text{0xx \# x11} = \{00\text{x}, 0\text{x}0\} \neq ø$$

The cube x11 is essential because it is the only prime implicant that covers the vertex 111, for which $f = 1$. The prime implicant 0xx is essential because it is the only one that covers the vertices 000, 001, and 010. This can be seen in the Karnaugh map in Figure 4.9.

Example 4.17

In Example 4.15 we found that the prime implicants of the function in Figure 4.10 are P = {x01x, x101, 01x1, 0x1x, xx10}. Because this function has no don't cares, we compute

$$\text{x01x} \# (P - \text{x01x}) = 1011 \neq ø$$

This is computed in the following steps: x01x # x101 = x01x, then x01x # 01x1 = x01x, then x01x # 0x1x = 101x, and finally 101x # xx10 = 1011. Similarly, we obtain

$$\text{x101} \# (P - \text{x101}) = 1101 \neq ø$$
$$\text{01x1} \# (P - \text{01x1}) = ø$$
$$\text{0x1x} \# (P - \text{0x1x}) = ø$$
$$\text{xx10} \# (P - \text{xx10}) = 1110 \neq ø$$

Therefore, the essential prime implicants are x01x, x101, and xx10 because they are the only ones that cover the vertices 1011, 1101, and 1110, respectively. This is obvious from the Karnaugh map in Figure 4.10.

When checking whether a cube A is essential, the #-operation with one of the cubes in $P - A$ may generate multiple cubes. If so, then each of these cubes has to be checked using the #-operation with all of the remaining cubes in $P - A$.

4.10.3 Complete Procedure for Finding a Minimal Cover

Having introduced the $*$- and #-operations, we can now outline a complete procedure for finding a minimal cover for any n-variable function. Assume that the function f is specified in terms of vertices for which $f = 1$; these vertices are often referred to as the *ON-set* of the function. Also, assume that the don't-care conditions are specified as a *DC-set*. Then the initial cover for f is a union of the ON and DC sets.

Prime implicants of f can be generated using the $*$-operation, as explained in section 4.10.1. Then the #-operation can be used to find the essential prime implicants as presented in section 4.10.2. If the essential prime implicants cover the entire ON-set, then they form the minimum-cost cover for f. Otherwise, it is necessary to include other prime implicants until all vertices in the ON-set are covered.

A nonessential prime implicant p^i should be deleted if there exists a less-expensive prime implicant p^j that covers all vertices of the ON-set that are covered by p^i. If the remaining nonessential prime implicants have the same cost, then a possible heuristic approach is to arbitrarily select one of them, include it in the cover, and determine the rest of the cover. Then an alternative cover is generated by excluding this prime implicant, and the lower-cost cover is chosen for implementation. We already used this approach, which is often referred to as the *branching* heuristic, in section 4.2.2.

The preceding discussion can be summarized in the form of the following minimization procedure:

1. Let $C^0 = ON \cup DC$ be the initial cover of function f and its don't-care conditions.
2. Find all prime implicants of C^0 using the $*$-operation; let P be this set of prime implicants.
3. Find the essential prime implicants using the #-operation. A prime implicant p^i is essential if $p^i \# (P - p^i) \# DC \neq ø$.

If the essential prime implicants cover all vertices of the ON-set, then these implicants form the minimum-cost cover.

4. Delete any nonessential p^i that is more expensive (i.e., a smaller cube) than some other prime implicant p^j if $p^i \# DC \# p^j = \text{ø}$.
5. Choose the lowest-cost prime implicants to cover the remaining vertices of the ON-set. Use the branching heuristic on the prime implicants of equal cost and retain the cover with the lowest cost.

Example 4.18

To illustrate the minimization procedure, we will use the function

$$f(x_1, x_2, x_3, x_4, x_5) = \sum m(0, 1, 4, 8, 13, 15, 20, 21, 23, 26, 31) + D(5, 10, 24, 28)$$

To help the reader follow the discussion, this function is also shown in the form of a Karnaugh map in Figure 4.41.

The initial cover C^0 consists of the ON-set and the DC-set:

$$C^0 = \{00000, 00001, 00100, 01000, 01101, 01111, 10100, 10101, 10111, 11010, 11111, 00101, 01010, 11000, 11100\}$$

Using the $*$-operation, the subsequent covers obtained are

$$C^1 = \{0000x, 00x00, 0x000, 00x01, x0100, 0010x, 010x0, x1000, 011x1, 0x101, x1111, 1010x, 1x100, 101x1, x0101, 1x111, x1010, 110x0, 11x00\}$$

$$C^2 = \{0x000, 011x1, 0x101, x1111, 1x100, 101x1, 1x111, 11x00, 00x0x, x010x, x10x0\}$$

$$C^3 = C^2$$

Therefore, $P = C^2$.

Using the #-operation, we find that there are two essential prime implicants: 00x0x (because it is the only one that covers the vertex 00001) and x10x0 (because it is the only

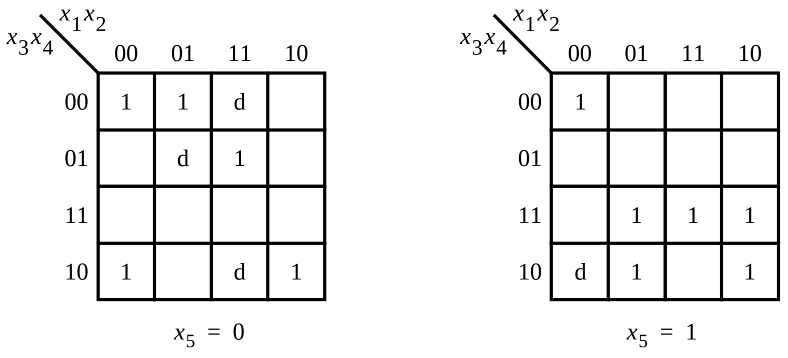


Figure 4.41 The function for Example 4.18.

one that covers the vertex 11010). The minterms of f covered by these two prime implicants are $m(0, 1, 4, 8, 26)$.

Next, we find that 1x100 can be deleted because the only ON-set vertex that it covers is 10100 (m_{20}), which is also covered by x010x and the cost of this prime implicant is lower. Note that having removed 1x100, the prime implicant x010x becomes essential because none of the other remaining prime implicants covers the vertex 10100. Therefore, x010x has to be included in the final cover. It covers $m(20, 21)$.

There remains to find prime implicants to cover $m(13, 15, 23, 31)$. Using the branching heuristic, the lowest-cost cover is obtained by including the prime implicants 011x1 and 1x111. Thus the final cover is

$$C_{minimum} = \{00x0x, x10x0, x010x, 011x1, 1x111\}$$

The corresponding sum-of-products expression is

$$f = \overline{x}_1\overline{x}_2\overline{x}_4 + x_2\overline{x}_3\overline{x}_5 + \overline{x}_2x_3\overline{x}_4 + \overline{x}_1x_2x_3x_5 + x_1x_3x_4x_5$$

Although this procedure is tedious when performed by hand, it is not difficult to write a computer program to implement the algorithm automatically. The reader should check the validity of our solution by finding the optimal realization from the Karnaugh map in Figure 4.41.

4.11 PRACTICAL CONSIDERATIONS

The purpose of the preceding section was to give the reader some idea about how minimization of logic functions may be automated for use in CAD tools. We chose a scheme that is not too difficult to explain. From the practical point of view, this scheme has some drawbacks. The main difficulty is that the number of cubes that must be considered in the process can be extremely large.

If the goal of minimization is relaxed so that it is not imperative to find a minimum-cost implementation, then it is possible to derive heuristic techniques that produce good results in reasonable time. A technique of this type forms the basis of the widely used Espresso program, which is available from the University of California at Berkeley via the World Wide Web. Espresso is a two-level optimization program. Both input to the program and its output are specified in the format of cubes. Instead of using the $*$-operation to find the prime implicants, Espresso uses an implicant-expansion technique. (See problem 4.27 for an illustration of the expansion of implicants.) A comprehensive explanation of Espresso is given in [19], while simplified outlines can be found in [3, 12].

The University of California at Berkeley also provides two software programs that can be used for design of multilevel circuits, called MIS [20] and SIS [21]. They allow a user to apply various multilevel optimization techniques to a logic circuit. The user can experiment with different optimization strategies by applying techniques such as factoring and decomposition to all or part of a circuit. SIS also includes the Espresso algorithm for two-level minimization of functions, as well as many other optimization techniques.

Numerous commercial CAD systems are on the market. Three companies whose products are widely used are Cadence Design Systems, Mentor Graphics, and Synopsys. Information on their products is available on the World Wide Web. Each company provides logic synthesis software that can be used to target various types of chips, such as PLDs, gate arrays, standard cells, and custom chips. Because there are many possible ways to synthesize a given circuit, as we saw in the previous sections, each commercial product uses a proprietary logic optimization strategy based on heuristics.

To describe CAD tools, some new terminology has been invented. In particular, we should mention two terms that are widely used in industry: *technology-independent logic synthesis* and *technology mapping*. The first term refers to techniques that are applied when optimizing a circuit without considering the resources available in the target chip. Most of the techniques presented in this chapter are of this type. The second term, technology mapping, refers to techniques that are used to ensure that the circuit produced by logic synthesis can be realized using the logic resources available in the target chip. A good example of technology mapping is the transformation from a circuit in the form of logic operations such as AND and OR into a circuit that consists of only NAND operations. This type of technology mapping is done when targeting a circuit to a gate array that contains only NAND gates. Another example is the translation from logic operations to lookup tables, which is done when targeting a design to an FPGA. It should be noted that the terminology is sometimes used inconsistently. For instance, some CAD systems consider factoring, which was discussed in section 4.7.1, to be technology independent, whereas other systems consider it to be a part of the technology mapping. Still other systems, such as MAX+plusII, do not use these two terms at all, even though they clearly implement both types of techniques. We will not rely on these terms in this book and have mentioned them only for completeness.

The next section provides a more detailed discussion of CAD tools. To give an example of the features provided in these tools, we use the MAX+plusII system that accompanies the book. Of course, different CAD systems offer different features. MAX+plusII synthesizes designs for implementation in PLDs. It includes all the optimization techniques introduced in this chapter.

4.12 CAD Tools

In section 2.8 we introduced the concept of a CAD system and described CAD tools for performing design entry, initial synthesis, and functional simulation. In this section we introduce the remaining tools in a typical CAD system, which are used for performing logic synthesis and optimization, physical design, and timing simulation. The principles behind such tools are quite general; the details may vary from one system to another. We will discuss the main aspects of the tools in as general a fashion as possible. However, to provide a sufficient degree of reality, we will use illustrative examples based on the Altera MAX+plusII system that is provided with the book. To fully grasp the concepts presented in the following discussion, the reader should go through the material in Tutorials 1 and 2, which are presented in Appendices B and C.

A typical CAD system comprises tools for performing the following tasks:

- *Design entry* allows the designer to enter a description of the desired circuit in the form of truth tables, schematic diagrams, or HDL code.
- *Initial synthesis* generates an initial circuit, based on data entered during the design entry stage.
- *Functional simulation* is used to verify the functionality of the circuit, based on inputs provided by the designer.
- *Logic synthesis and optimization* applies optimization techniques to derive an optimized circuit.
- *Physical design* determines how to implement the optimized circuit in a given target technology, for example, in a PLD chip.
- *Timing simulation* determines the propagation delays that are expected in the implemented circuit.
- *Chip configuration* configures the actual chip to realize the designed circuit.

The first three of these tools are discussed in Chapter 2. The rest are described below.

4.12.1 Logic Synthesis and Optimization

The optimization techniques described in this chapter are automatically applied by CAD tools when synthesizing logic circuits. Consider the VHDL code in Figure 4.42. It describes the function f from Figure 4.5*a* in the canonical form, which consists of minterms. We used the MAX+plusII system to synthesize f for implementation in a FLEX 10K FPGA. The result obtained was

$$f = \overline{x}_2 x_3 + x_1 \overline{x}_3$$

which is the same minimal sum-of-products expression derived in Figure 4.5*a*. This result was displayed in a *report file*, which is produced by the CAD system. The report file includes a set of logic equations that describe the synthesized circuit.

CAD tools often include many optional features that can be invoked by the user. Figure 4.43 shows some of the logic synthesis options provided by MAX+plusII. Although the reader may not recognize all the options shown, the meaning of terms such as minimization, multilevel synthesis, factoring, and decomposition should be obvious at this point. Detailed explanation of various synthesis procedures can be found in specialized texts [5, 22].

The optimized circuit produced by the logic synthesis tools depends both on the type of logic resources available in the target chip and on the particular CAD system that is used. For example, if the target chip is a CPLD, then each logic function in the circuit is expressed in terms of the gates available in a macrocell. For an FPGA that contains lookup tables (LUTs), the number of inputs to each logic function in the circuit is constrained by the size of the LUTs. If the target chip is a gate array, then the logic functions in the optimized circuit are expressed using only the type of logic cells available in the gate array. Finally,

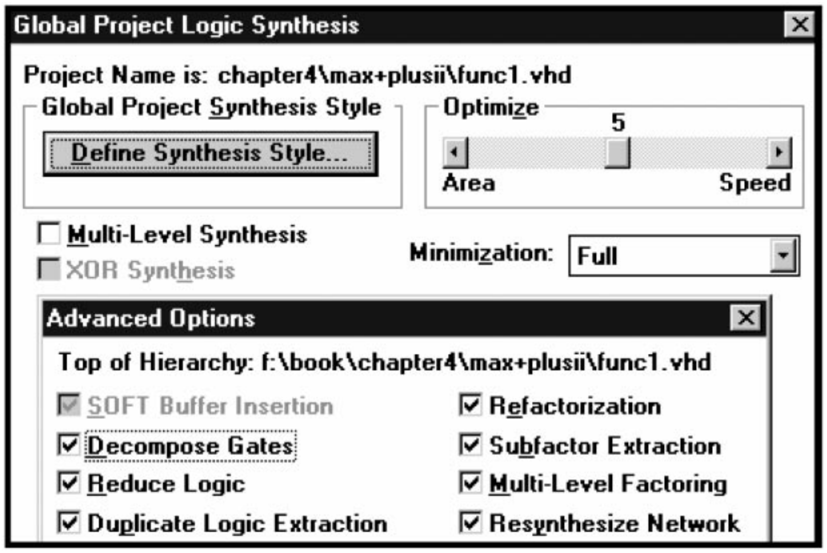
```
ENTITY func1 IS
      PORT ( x1, x2, x3  : IN      BIT ;
                 f               : OUT  BIT ) ;
END func1 ;

ARCHITECTURE LogicFunc OF func1 IS
BEGIN
      f <= (NOT x1 AND NOT x2 AND x3) OR
             (x1 AND NOT x2 AND NOT x3) OR
             (x1 AND NOT x2 AND x3) OR
             (x1 AND x2 AND NOT x3) ;
END LogicFunc ;
```

Figure 4.42 VHDL code for the function in Figure 4.5*a*.

if standard-cell technology is used, then the circuit comprises whatever types of logic cells can be fabricated on the standard-cell chip.

4.12.2 Physical Design

After logic synthesis the next step in the design flow is to determine exactly how to implement the circuit in the target technology. This step is usually called *physical design*, or *layout synthesis*. There are two main parts to physical design: placement and routing.



Global Project Logic Synthesis

Project Name is: chapter4\max+plusii\func1.vhd

Global Project Synthesis Style: Define Synthesis Style...

Optimize: 5 (Area – Speed)

☐ Multi-Level Synthesis
☐ XOR Synthesis

Minimization: Full

Advanced Options

Top of Hierarchy: f:\book\chapter4\max+plusii\func1.vhd

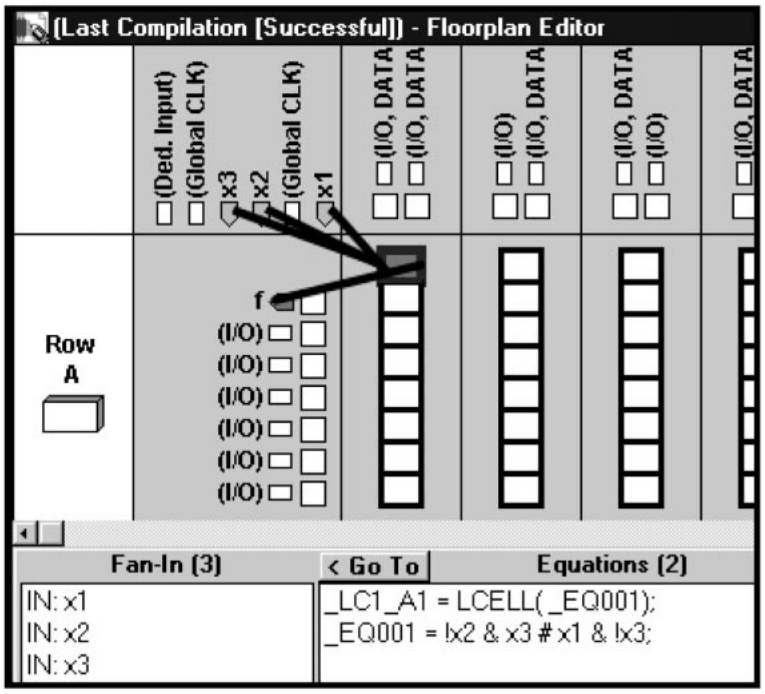
☑ SOFT Buffer Insertion	☑ Refactorization
☑ Decompose Gates	☑ Subfactor Extraction
☑ Reduce Logic	☑ Multi-Level Factoring
☑ Duplicate Logic Extraction	☑ Resynthesize Network

Figure 4.43 Logic synthesis options in MAX+plusII.

A *placement* CAD tool determines where in the target device each logic function in the optimized circuit will be realized. The placement task is highly dependent on the implementation technology. For example, if a PLD is used for implementation, then the structure of the chip is predefined and the placement tool determines which logic resources in the chip are to be used to realize each logic function in the circuit. In the case of a CPLD, the logic functions are assigned to macrocells. For an FPGA each logic function is assigned to a logic cell.

Continuing with our example, the MAX+plusII placement tool realizes the function f from Figure 4.42 in the FLEX 10K FPGA as depicted in Figure 4.44. The figure represents a screen capture of the *Floorplan Editor*, which displays the results generated by the physical design tools. The small squares in the diagram represent the logic cells in the FPGA, which are four-input LUTs (see Appendix E). The logic cell at the top left is used to realize the function f. At the bottom of the window, the Floorplan Editor shows the logic expression contained in the LUT for f. Lines are drawn to indicate the input and output connections of this logic cell. They connect to the I/O cells that are used for inputs x_1, x_2, and x_3, as well as for the output f.

After the placement has been completed, the next step is to decide which of the wires in the chip are to be used to realize the required interconnections. This step is called *routing*. Like the placement task, routing is highly dependent on the implementation technology.



```
IN: x1
IN: x2
IN: x3
```

```
_LC1_A1 = LCELL( _EQ001);
_EQ001 = !x2 & x3 # x1 & !x3;
```

Figure 4.44 The results of physical design for the VHDL code in Figure 4.42.

For a CPLD the programming switches attached to the interconnection wires must be set to connect the macrocells together as needed for the implemented circuit. Similarly, for an FPGA the programming switches are used to connect the logic cells together. If the implementation technology is a gate array or a standard-cell chip, then the routing tool specifies the interconnection wires that are to be fabricated between the rows of logic cells. Some small examples of routing were presented in Chapter 3, in Figures 3.59 and 3.67.

Both the placement and routing tasks can be difficult problems to solve for the CAD tools, especially for the larger devices, such as FPGAs, gate arrays, and standard-cell chips. Much research effort has gone into the development of algorithms for these tasks. Detailed explanations of these algorithms can be found in more specialized books [23, 24].

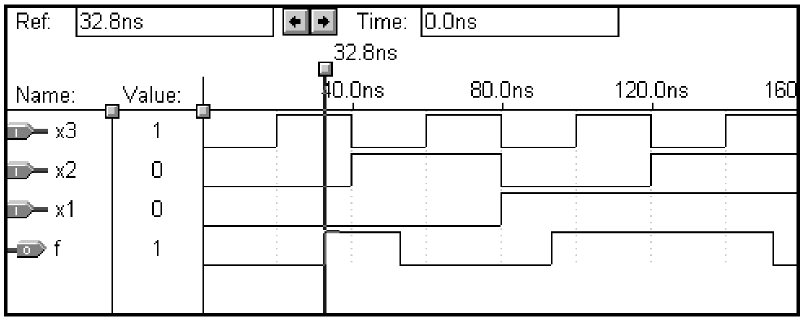
4.12.3 TIMING SIMULATION

In section 2.8.3 we described functional simulation and said that it is used to ensure that a logic circuit description entered into a CAD system functions as expected by the designer. In functional simulation it is assumed that signal propagation delays through logic gates are negligible. In this section we consider *timing simulation*, which simulates the actual propagation delays in the technology chosen for implementation.

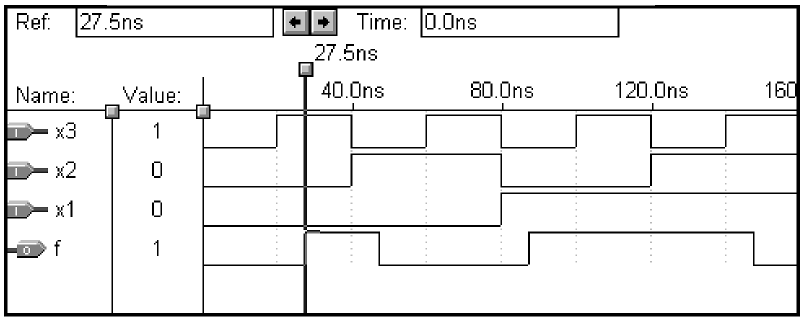
After the physical design tasks are completed, the CAD system has determined exactly how the designed circuit is to be realized in the target technology. It is then possible for the CAD tools to create a model of the circuit that includes all timing aspects of the target chip. The model represents the delays associated with the logic resources in the chip (macrocells or logic cells) and with the interconnection wires.

The results of timing simulation for the function f from Figure 4.42 are shown in Figure 4.45. They were obtained using the timing simulator in MAX+plusII. The simulator allows the designer to specify a waveform for each of the inputs x_1, x_2, and x_3, and the tool generates the corresponding waveform produced at the output f. Part (*a*) of the figure gives the timing expected when the circuit is implemented in the FLEX 10K FPGA. Observe that a heavy vertical line, which is called the *reference line*, is set at the point where f first makes a transition from 0 to 1. The simulator specifies in the box labeled Ref that the reference line is set at 32.8 ns from the start time of the simulation. The change in $x_1x_2x_3$ from 000 to 001 takes place at 20 ns; hence $32.8 - 20 = 12.8$ ns are required for the change in inputs to cause f to change to 1. The reason for the delay at f is that the signals must propagate through the transistor circuits in the FPGA. The timing aspects of transistor circuits are discussed in Chapter 3.

Figure 4.45*b* shows the same simulation for the circuit when it is implemented in a MAX 7000 CPLD. Of course, the circuit implements the same function as when implemented in the FLEX 10K FPGA, but the timing is different. In the MAX 7000 CPLD, f changes 7.5 ns after the inputs change. The speed of a circuit may vary considerably when implemented in different types of chips. Although our example suggests that the CPLD provides much faster speed than the FPGA, the difference is exaggerated because of the small size of the circuit. In general, when larger circuits are implemented, CPLDs and FPGAs provide similar speeds.



(a) Timing in an FPGA



(b) Timing in a CPLD

Figure 4.45 Timing simulation for the VHDL code in Figure 4.42.

4.12.4 Summary of Design Flow

Figure 4.46 summarizes the design flow of a complete CAD system. After initial synthesis the logic synthesis tool automatically optimizes the circuit being designed. The physical design tool then determines exactly how to implement the circuit in the chosen technology. Timing simulation ensures that the implemented circuit meets the required performance. Note that if functional correctness has already been ascertained using functional simulation, as discussed in section 2.8, then the functionality of the circuit need not be verified using timing simulation. However, if functional simulation was not done, then timing simulation can be used to check for proper functionality as well. If timing or functional problems are discovered, they are corrected by returning to the previous steps in the design flow. For functional errors it is necessary to revisit the design entry step. For timing errors it may be possible to correct the problems by using the logic synthesis tool. For example, the window displayed in Figure 4.43 shows a sliding bar that can be used to change the

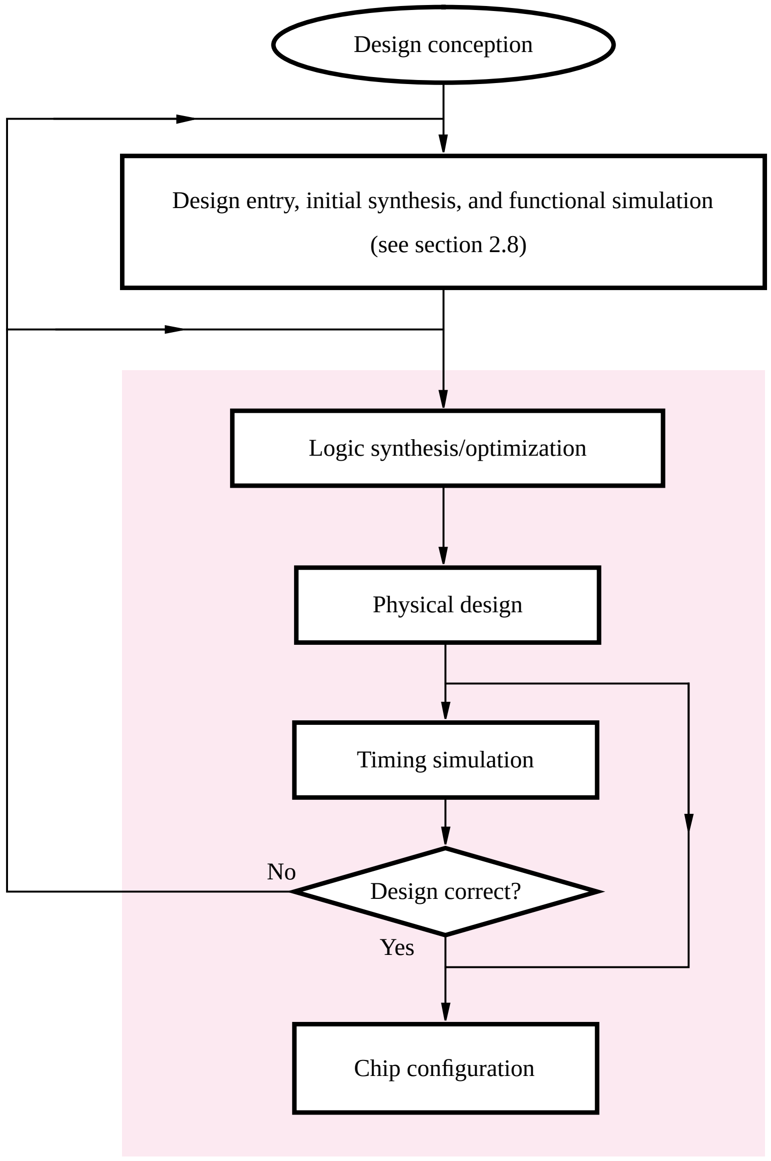


Figure 4.46 A complete CAD system.

emphasis of the logic synthesis algorithms between circuit cost or circuit speed. Cost is optimized by minimizing the amount of area needed on the chip to implement the circuit. Speed is optimized by minimizing the propagation delay of signals in the circuit. It may also be possible to use a faster speed grade of the selected chip or to select a different type of chip that results in a faster circuit, as in the example from Figure 4.45. If the logic synthesis tool cannot resolve the timing problems, then it is necessary to return to the beginning of the design flow to consider other design alternatives. The final step is to configure the target chip to implement the desired circuit.

4.12.5 Examples of Circuits Synthesized from VHDL Code

In section 2.9 we showed how simple VHDL programs can be written to describe logic functions. This section introduces additional features of VHDL and provides some examples of circuits designed using VHDL code.

Recall that a logic signal is represented in VHDL as a data object, and each data object has an associated type. In the examples in section 2.9, all data objects have the type BIT, which means that they can assume only the values 0 and 1. To give more flexibility, VHDL provides another data type called *STD_LOGIC*. Signals represented using this type can have several different values.

As its name implies, STD_LOGIC is meant to serve as the standard data type for representation of logic signals. An example using the STD_LOGIC type is given in Figure 4.47. The VHDL code shown is the same as that given in Figure 4.42 except that here the type STD_LOGIC is used instead of BIT. The VHDL compiler would synthesize this code in exactly the same way as described for the code in Figure 4.42.

To use the STD_LOGIC type, VHDL code must include the two lines given at the beginning of Figure 4.47. These statements serve as directives to the VHDL compiler. They are needed because the original VHDL standard, IEEE 1076, did not include the STD_LOGIC type. The way that the new type was added to the language, in the IEEE 1164 standard, was to provide the definition of STD_LOGIC as a set of files that can be included with VHDL code when compiled. The set of files is called a *library*. The purpose of the first line in Figure 4.47 is to declare that the code will make use of the IEEE library.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func2 IS
    PORT ( x1, x2, x3  : IN   STD_LOGIC ;
           f           : OUT  STD_LOGIC ) ;
END func2 ;

ARCHITECTURE LogicFunc OF func2 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND x3) OR
         (x1 AND NOT x2 AND NOT x3) OR
         (x1 AND NOT x2 AND x3) OR
         (x1 AND x2 AND NOT x3) ;
END LogicFunc ;
```

Figure 4.47 The VHDL code in Figure 4.42 using STD_LOGIC.

In VHDL there are two main aspects to the definition of a new data type. First, the set of values that a data object of the new type can assume must be specified. For STD_LOGIC, there are a number of legal values, but the ones that are the most important for describing logic functions are 0, 1, Z, and −. We introduced the logic value Z, which represents the high-impedance state, in section 3.8.8. The − logic value represents the don't-care condition, which we labeled as *d* in section 4.4. The second requirement is that all legal uses in VHDL code of the new data type must be specified. For example, it is necessary to specify that the type STD_LOGIC is legal for use with Boolean operators.

In the IEEE library one of the files defines the STD_LOGIC data type itself and specifies some basic legal uses, such as for Boolean operations. In Figure 4.47 the second line of code tells the VHDL compiler to use the definitions in this file when compiling the code. The file encapsulates the definition of STD_LOGIC in what is known as a *package*. The package is named std_logic_1164. It is possible to instruct the VHDL compiler to use only a subset of the package, but the normal use is to specify the word *all* to indicate that the entire package is of interest, as we have done in Figure 4.47.

The IEEE library files are plain text files that can be examined with any text editor. Although it is not necessary for purposes of understanding the examples in this book, an interested reader can examine the IEEE library files distributed with the MAX+plusII system that accompanies the book. When the software is installed on a computer running a Microsoft Windows operating system, the IEEE library files are normally installed in the file system in the location C:\maxplus2\max2vhdl\ieee. The file that defines the STD_LOGIC type is named "std1164.vhd."

For the examples of VHDL code given in this book, we will almost always use only the type STD_LOGIC. Besides simplifying the code, using just one data type has another benefit. VHDL is a strongly type-checked language. This means that the VHDL compiler carefully checks all data object assignment statements to ensure that the type of the data object on the left side of the assignment statement is exactly the same as the type of the data object on the right side. Even if two data objects seem compatible from an intuitive point of view, such as an object of type BIT and one of type STD_LOGIC, the VHDL compiler will not allow one to be assigned to the other. Many synthesis tools provide conversion utilities to convert from one type to another, but we will avoid this issue by using only the STD_LOGIC data type in most cases. In the remainder of this section, a few examples of VHDL code are presented. We show the results of synthesizing the code for implementation in two different types of chips, a CPLD and an FPGA.

Example 4.19

Consider the VHDL code in Figure 4.48. The logic expression for f corresponds to the truth table in Figure 4.1. We derived the minimal sum-of-products form, $f = \overline{x}_3 + x_1\overline{x}_2$, using the Karnaugh map in Figure 4.5*b*. If we compile the VHDL code for implementation in a MAX 7000 CPLD, the MAX+plusII tools produce the expression

$$f = \overline{x}_3 + x_1\overline{x}_2x_3$$

It is easy to show that this expression is not fully minimized. Using the identity 16*a* in section 2.5, the expression can be reduced to $f = \overline{x}_3 + x_1\overline{x}_2$, which is the minimal form that we derived manually. However, because the circuit is being implemented in a CPLD, the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func3 IS
    PORT ( x1, x2, x3  : IN    STD_LOGIC ;
             f          : OUT  STD_LOGIC ) ;
END func3 ;

ARCHITECTURE LogicFunc OF func3 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND NOT x3) OR
         (NOT x1 AND x2 AND NOT x3) OR
         (x1 AND NOT x2 AND NOT x3) OR
         (x1 AND NOT x2 AND x3) OR
         (x1 AND x2 AND NOT x3) ;
END LogicFunc ;
```

Figure 4.48 The VHDL code for the function in Figure 4.1.

extra literal in the product term $x_1\overline{x}_2x_3$ does not increase the cost. Figure 4.49 shows the expression for f realized in a macrocell. Observe that since the XOR gate in the macrocell is not used for the circuit, one input to the XOR gate is connected to 0.

As we have said before, CAD tools include many options that can affect the results of the synthesis procedure. Some of the options available in MAX+plusII are shown in the window in Figure 4.43. One of the options is called *XOR synthesis*, which is a synthesis

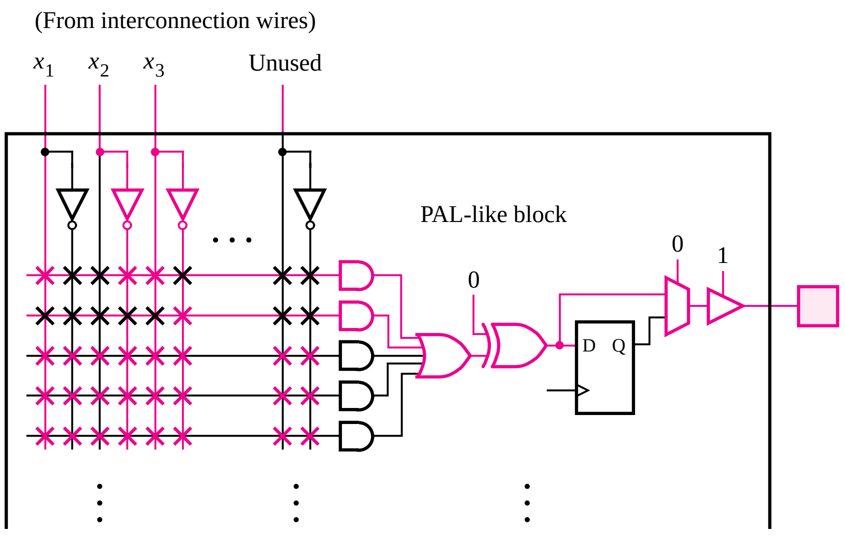


Figure 4.49 Implementation of the VHDL code in Figure 4.48.

technique that attempts to use XOR gates as judiciously as possible. If this option is turned on and the VHDL code in Figure 4.48 is synthesized again, the resulting expression for f becomes

$$f = \overline{x}_3 \oplus x_1\overline{x}_2x_3$$

The reader should verify that this is functionally equivalent to the sum-of-products form given above. The implementation of this expression in a MAX 7000 macrocell is depicted in Figure 4.50. The XOR gate is now used as part of the function, with one input connected to $\overline{x}_3$. Since it occupies a single macrocell, the cost of the implementation is the same as for the circuit in Figure 4.49. Although not true in this example, for some logic functions the XOR gates lead to greatly reduced cost. We should note that it is even possible to realize any arbitrary logic function using only AND and XOR gates [4]. We discuss some typical uses of XOR gates in Chapter 5. As this example illustrates, for any given logic function, several different implementations often have the same cost in a given chip.

Figure 4.51 gives the results of synthesizing the VHDL code in Figure 4.48 into a FLEX 10K FPGA. In this case the compiler generates the same sum-of-products form that we derived manually. Because the logic cells in the FLEX 10K chip are four-input lookup tables, only a single logic cell is needed for this function. The figure shows that the variables x_1, x_2, and x_3 are connected to the LUT inputs called i_2, i_3, and i_4. Input i_1 is not used because the function requires only three inputs. The truth table in the LUT indicates that the unused input is treated as a don't care. Thus only half of the rows in the table are shown, since the other half is identical. The unused LUT input is shown connected to 0 in the figure, but it could just as well be connected to 1.

It is interesting to consider the benefits provided by the optimizations used in logic synthesis. For the implementation in the CPLD, the function was simplified from the original five product terms in the canonical form to just two product terms. However, both

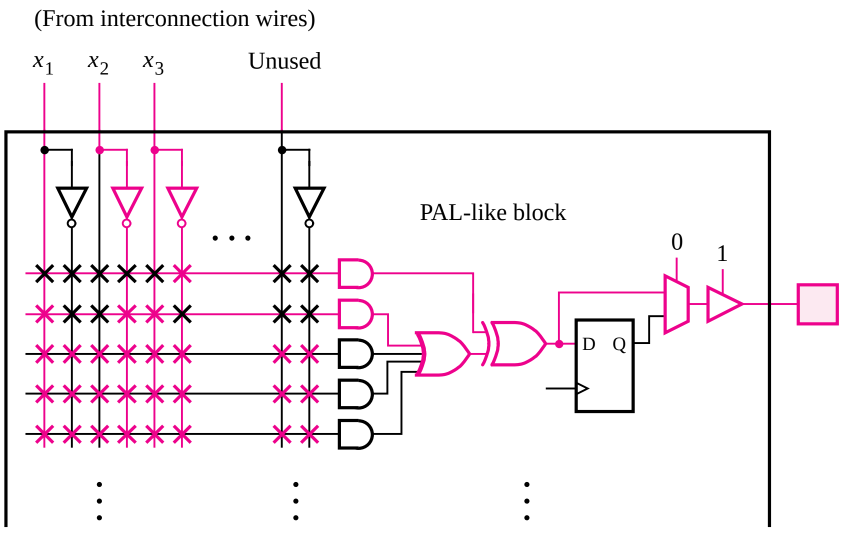
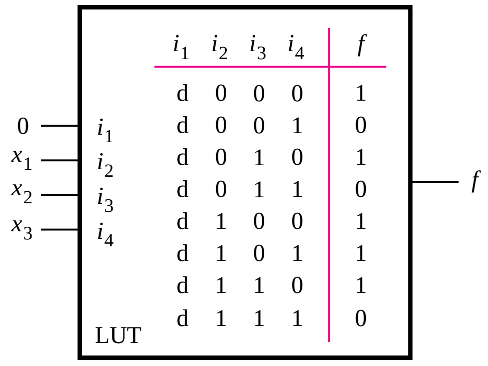


Figure 4.50 Implementation of the VHDL code in Figure 4.48 using XOR synthesis.



i_1	i_2	i_3	i_4	f
d	0	0	0	1
d	0	0	1	0
d	0	1	0	1
d	0	1	1	0
d	1	0	0	1
d	1	0	1	1
d	1	1	0	1
d	1	1	1	0

Figure 4.51 The VHDL code in Figure 4.48 implemented in a LUT.

the optimized and nonoptimized forms fit into a single macrocell in the chip, and thus they have the same cost (Appendix E shows that the MAX 7000 CPLD has five product terms in each macrocell). Similarly, for the FPGA, since a LUT is used for implementation, it does not matter whether the function is minimized, because it fits in a single LUT. The reason is that our example circuit is very small. For large circuits it is essential to perform the optimization. Examples 4.20 and 4.21 illustrate logic functions for which the cost of implementation is reduced when optimized.

Example 4.20 The VHDL code in Figure 4.52 corresponds to the function f_1 in Figure 4.7. Because there are six product terms in the canonical form, two macrocells would be needed in a MAX 7000 CPLD. When synthesized by the CAD tools, the resulting expression is

$$f = \overline{x}_2 x_3 + x_1 \overline{x}_3 x_4$$

which is the same as the expression derived in Figure 4.7. Because the optimized expression has only two product terms, it can be realized using just one macrocell and hence results in a lower cost.

When f_1 is synthesized for implementation in a FLEX 10K FPGA, the expression generated is the same as for the CPLD. Since the function has only four inputs, it needs just one LUT.

Example 4.21 In section 4.7 we used a seven-variable logic function as a motivation for multilevel synthesis. This function is given in the VHDL code in Figure 4.53. The logic expression is in minimal sum-of-products form. When it is synthesized for implementation in a MAX 7000 CPLD, no optimizations are performed by the CAD tools. The function requires one macrocell. This function is more interesting when we consider its implementation in the FLEX 10K FPGA. Because there are seven inputs, more than one LUT is required. If the function is implemented directly as given in the VHDL code, then five LUTs are needed, as depicted in Figure 4.54*a*. Rather than showing the truth table programmed in each LUT, we

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func4 IS
    PORT ( x1, x2, x3, x4  : IN    STD_LOGIC ;
           f               : OUT   STD_LOGIC ) ;
END func4 ;

ARCHITECTURE LogicFunc OF func4 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND x3 AND NOT x4) OR
         (NOT x1 AND NOT x2 AND x3 AND x4) OR
         (x1 AND NOT x2 AND NOT x3 AND x4) OR
         (x1 AND NOT x2 AND x3 AND NOT x4) OR
         (x1 AND NOT x2 AND x3 AND x4) OR
         (x1 AND x2 AND NOT x3 AND x4) ;
END LogicFunc ;
```

Figure 4.52 The VHDL code for f_1 in Figure 4.7.

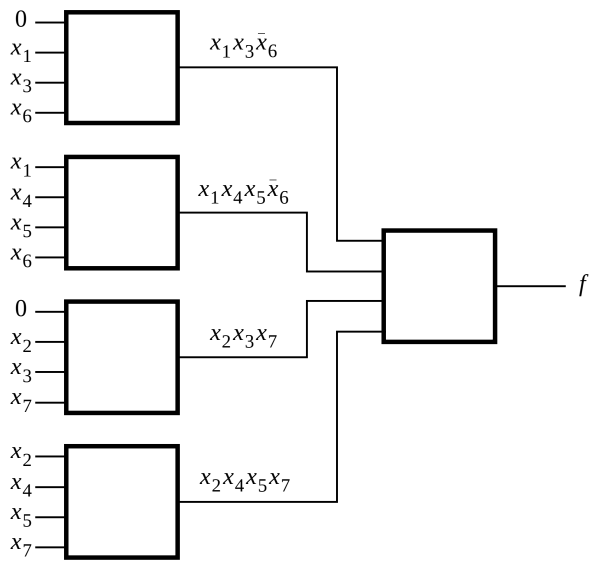
show the logic function that is implemented at the LUT output. Synthesis with MAX+plusII results in the following expression:

$$f = (x_1\bar{x}_6 + x_2x_7)(x_3 + x_4x_5)$$

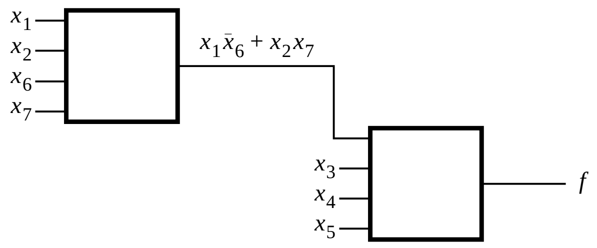
We derived the same expression by using factoring in section 4.7. As illustrated in Figure 4.54*b*, it can be implemented using only two LUTs. One LUT produces the term $S = x_1\bar{x}_6 + x_2x_7$. The other LUT implements the four-input function $f = Sx_3 + Sx_4x_5$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func5 IS
    PORT ( x1, x2, x3, x4, x5, x6, x7  : IN    STD_LOGIC ;
           f                           : OUT   STD_LOGIC ) ;
END func5 ;

ARCHITECTURE LogicFunc OF func5 IS
BEGIN
    f <= (x1 AND x3 AND NOT x6) OR
         (x1 AND x4 AND x5 AND NOT x6) OR
         (x2 AND x3 AND x7) OR
         (x2 AND x4 AND x5 AND x7) ;
END LogicFunc ;
```

Figure 4.53 The VHDL code for the function of section 4.7.



(a) Sum-of-products realization



(b) Factored realization

Figure 4.54 Implementation of the VHDL code in Figure 4.53.

4.13 Concluding Remarks

This chapter has attempted to provide the reader with an understanding of various aspects of synthesis for logic functions and how synthesis is automated using modern CAD tools. Now that the reader is comfortable with the fundamental concepts, we can examine digital circuits of a more sophisticated nature. The next chapter describes circuits that perform arithmetic operations, which are a key part of computers.

PROBLEMS

4.1 Find the minimum-cost SOP and POS forms for the function $f(x_1, x_2, x_3) = \sum m(1, 2, 3, 5)$.

4.2 Repeat problem 4.1 for the function $f(x_1, x_2, x_3) = \sum m(1, 4, 7) + D(2, 5)$.

4.3 Repeat problem 4.1 for the function $f(x_1, \ldots, x_4) = \Pi M(0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 14, 15)$.

4.4 Repeat problem 4.1 for the function $f(x_1, \ldots, x_4) = \sum m(0, 2, 8, 9, 10, 15) + D(1, 3, 6, 7)$.

4.5 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \Pi M(1, 4, 6, 7, 9, 12,15, 17, 20, 21, 22, 23, 28, 31)$.

4.6 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \sum m(0, 1, 3, 4, 6, 8, 9, 11, 13, 14, 16, 19, 20, 21, 22, 24, 25) + D(5, 7, 12, 15, 17, 23)$.

4.7 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \sum m(1, 4, 6, 7, 9, 10, 12, 15, 17, 19, 20, 23, 25, 26, 27, 28, 30, 31) + D(8, 16, 21, 22)$.

4.8 Find 5 three-variable functions for which the product-of-sums form has lower cost than the sum-of-products form.

4.9 A four-variable logic function that is equal to 1 if any three or all four of its variables are equal to 1 is called a *majority* function. Design a minimum-cost circuit that implements this majority function.

4.10 Derive a minimum-cost realization of the four-variable function that is equal to 1 if exactly two or exactly three of its variables are equal to 1; otherwise it is equal to 0.

4.11 Prove or show a counter-example for the statement: If a function f has a unique minimum-cost SOP expression, then it also has a unique minimum-cost POS expression.

4.12 A circuit with two outputs has to implement the following functions

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 6, 7, 9) + D(10, 11)$$

$$g(x_1, \ldots, x_4) = \sum m(2, 4, 9, 10, 15) + D(0, 13, 14)$$

Design the minimum-cost circuit and compare its cost with combined costs of two circuits that implement f and g separately. Assume that the input variables are available in both uncomplemented and complemented forms.

4.13 Repeat problem 4.12 for the following functions

$$f(x_1, \ldots, x_5) = \sum m(1, 4, 5, 11, 27, 28) + D(10, 12, 14, 15, 20, 31)$$

$$g(x_1, \ldots, x_5) = \sum m(0, 1, 2, 4, 5, 8, 14, 15, 16, 18, 20, 24, 26, 28, 31) + D(10, 11, 12, 27)$$

4.14 Implement the logic circuit in Figure 4.26 using NAND gates only.

4.15 Implement the logic circuit in Figure 4.26 using NOR gates only.

4.16 Implement the logic circuit in Figure 4.28 using NAND gates only.

4.17 Implement the logic circuit in Figure 4.28 using NOR gates only.

4.18 Consider the function $f = x_3x_5 + \overline{x}_1x_2x_4 + x_1\overline{x}_2\overline{x}_4 + x_1x_3\overline{x}_4 + \overline{x}_1x_3x_4 + \overline{x}_1x_2x_5 + x_1\overline{x}_2x_5$. Derive a minimum-cost circuit that implements this function using NOT, AND, and OR gates.

4.19 Derive a minimum-cost circuit that implements the function $f(x_1, \ldots, x_4) = \sum m(4, 7, 8, 11) + D(12, 15)$.

4.20 Find the simplest realization of the function $f(x_1, \ldots, x_4) = \sum m(0, 3, 4, 7, 9, 10, 13, 14)$, assuming that the logic gates have a maximum fan-in of two.

4.21 Find the minimum-cost circuit for the function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 13, 14, 15)$. Assume that the input variables are available in uncomplemented form only. (Hint: use functional decomposition.)

4.22 Use functional decomposition to find the best implementation of the function $f(x_1, \ldots, x_5) = \sum m(1, 2, 7, 9, 10, 18, 19, 25, 31) + D(0, 15, 20, 26)$. How does your implementation compare with the lowest-cost SOP implementation? Give the costs.

4.23 Show that the following distributive-like rules are valid

$$(A \cdot B)\#C = (A\#C) \cdot (B\#C)$$

$$(A + B)\#C = (A\#C) + (B\#C)$$

4.24 Use the cubical representation and the method discussed in section 4.10 to find a minimum-cost SOP realization of the function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 7, 8, 9, 15)$.

4.25 Repeat problem 4.24 for the function $f(x_1, \ldots, x_5) = \overline{x}_1\overline{x}_3\overline{x}_5 + x_1x_2\overline{x}_3 + x_2x_3\overline{x}_4x_5 + x_1\overline{x}_2\overline{x}_3x_4 + x_1x_2x_3x_4\overline{x}_5 + \overline{x}_1x_2x_4\overline{x}_5 + \overline{x}_1\overline{x}_3x_4x_5$.

4.26 Use the cubical representation and the method discussed in section 4.10 to find a minimum-cost SOP realization of the function $f(x_1, \ldots, x_4)$ defined by the ON-set ON = {00x0, 100x, x010, 1111} and the don't-care set DC = {00x1, 011x}.

4.27 In section 4.10.1 we showed how the $*$-product operation can be used to find the prime implicants of a given function f. Another possibility is to find the prime implicants by expanding the implicants in the initial cover of the function. An implicant is *expanded* by removing one literal to create a larger implicant (in terms of the number of vertices covered). A larger implicant is valid only if it does not include any vertices for which $f = 0$. The largest valid implicants obtained in the process of expansion are the prime

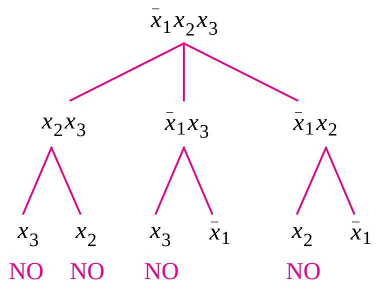


Figure P4.1 Expansion of implicant $\bar{x}_1x_2x_3$.

implicants. Figure P4.1 illustrates the expansion of the implicant $\overline{x}_1x_2x_3$ of the function in Figure 4.9, which is also used in Example 4.14. Note from Figure 4.9 that

$$\overline{f} = x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3$$

In Figure P4.1 the word NO is used to indicate that the expanded term is not valid, because it includes one or more vertices from $\overline{f}$. From the graph it is clear that the largest valid implicants that arise from this expansion are x_2x_3 and $\overline{x}_1$; they are prime implicants of f.

Expand the other four implicants given in the initial cover in Example 4.14 to find all prime implicants of f. What is the relative complexity of this procedure compared to the $*$-product technique?

Note: A technique based on such expansion of implicants is used to find the prime implicants in the Espresso CAD program [19].

4.28 Repeat problem 4.27 for the function in Example 4.15. Expand the implicants given in the initial cover C^0.

4.29 Consider the logic expressions

$$f = x_1\overline{x}_2\overline{x}_5+\overline{x}_1\overline{x}_2\overline{x}_4\overline{x}_5+x_1x_2x_4x_5+\overline{x}_1\overline{x}_2x_3\overline{x}_4+x_1\overline{x}_2x_3x_5+\overline{x}_2\overline{x}_3x_4\overline{x}_5+x_1x_2x_3x_4\overline{x}_5$$

$$g = \overline{x}_2x_3\overline{x}_4+\overline{x}_2\overline{x}_3\overline{x}_4\overline{x}_5+x_1x_3x_4\overline{x}_5+x_1\overline{x}_2x_4\overline{x}_5+x_1x_3x_4x_5+\overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_5+x_1x_2\overline{x}_3x_4x_5$$

Prove or disprove that $f = g$.

4.30 Consider the circuit in Figure P4.2, which implements functions f and g. What is the cost of this circuit, assuming that the input variables are available in both true and complemented forms? Redesign the circuit to implement the same functions, but at as low a cost as possible. What is the cost of your circuit?

4.31 Repeat problem 4.30 for the circuit in Figure P4.3. Use only NAND gates in your circuit.

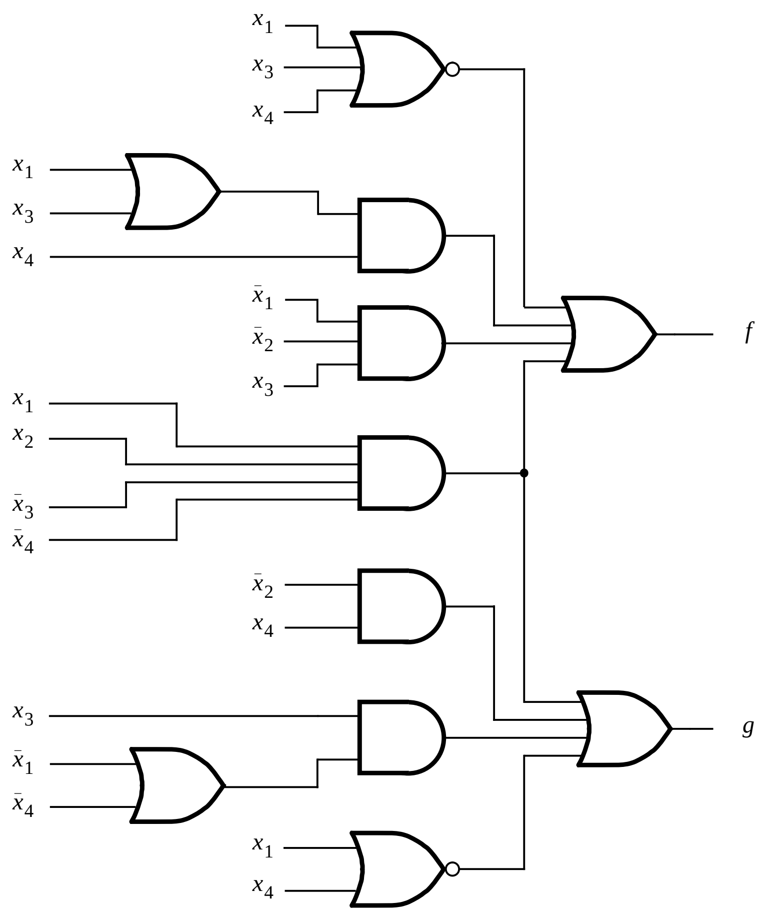


Figure P4.2 Circuit for problem 4.30.

References

1. M. Karnaugh, "A Map Method for Synthesis of Combinatorial Logic Circuits," *Transactions of AIEE, Communications and Electronics* 72, part 1, November 1953, pp. 593–599.
2. R. L. Ashenhurst, "The Decomposition of Switching Functions," Proc. of the Symposium on the Theory of Switching, 1957, *Vol. 29 of Annals of Computation Laboratory* (Harvard University: Cambridge, MA, 1959), pp. 74–116.
3. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed. (Wiley: New York, 1993).
4. T. Sasao, *Logic Synthesis and Optimization* (Kluwer: Boston, MA, 1993).

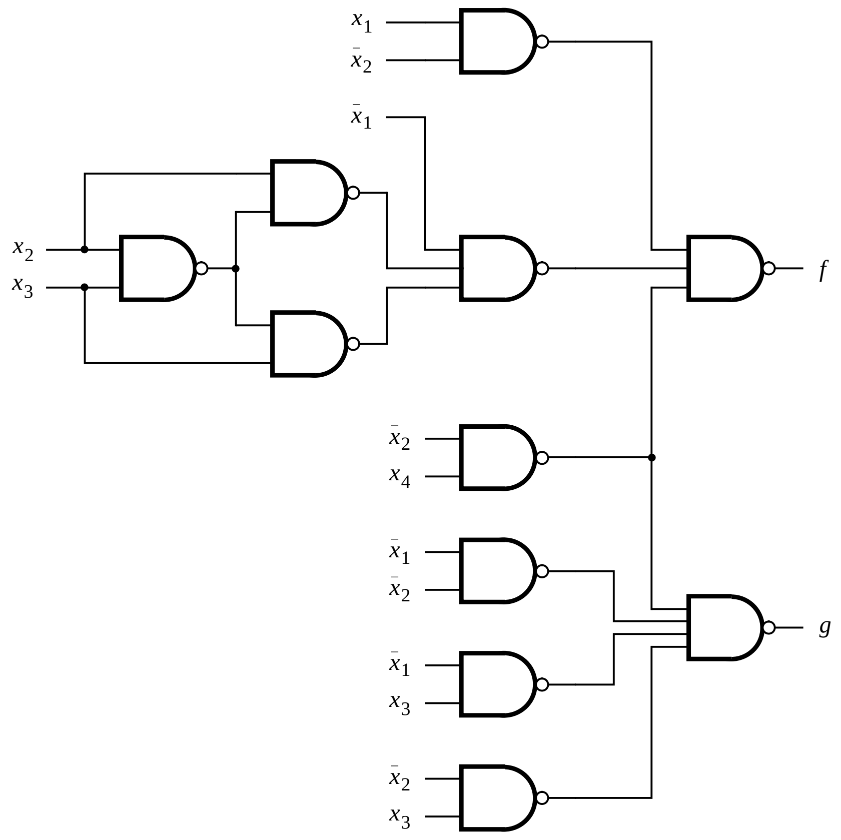


Figure P4.3 Circuit for problem 4.31.

5. S. Devadas, A. Gosh, and K. Keutzer, *Logic Synthesis* (McGraw-Hill: New York, 1994).
6. W. V. Quine, "The Problem of Simplifying Truth Functions," *Amer. Math. Monthly* 59 (1952), pp. 521–31.
7. E. J. McCluskey Jr., "Minimization of Boolean Functions," *Bell System Tech. Journal*, November 1956, pp. 521–31.
8. E. J. McCluskey, *Logic Design Principles* (Prentice-Hall: Englewood Cliffs, NJ, 1986).
9. J. F. Wakerly, *Digital Design Principles and Practices* (Prentice-Hall: Englewood Cliffs, NJ, 1990).
10. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
11. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed. (West: St. Paul, MN, 1993).
12. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).

13. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
14. J. P. Daniels, *Digital Design from Zero to One* (Wiley: New York, 1996).
15. P. K. Lala, *Practical Digital Logic Design and Testing* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
16. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
17. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals* (Prentice-Hall: Upper Saddle River, NJ, 1997).
18. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).
19. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer: Boston, MA, 1984).
20. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Synthesis Optimization System," *IEEE Transactions on Computer-Aided Design*, CAD-6, November 1987, pp. 1062–81.
21. E. M. Sentovic, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Technical Report UCB/ERL M92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.
22. G. De Micheli, *Synthesis and Optimization of Digital Circuits* (McGraw-Hill: New York, 1994).
23. N. Sherwani, *Algorithms for VLSI Physical Design Automation* (Kluwer: Boston, MA, 1995).
24. B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems* (Benjamin/Cummings: Redwood City, CA, 1988).

chapter

6

Combinational-Circuit Building Blocks

Chapter Objectives

In this chapter you will learn about:

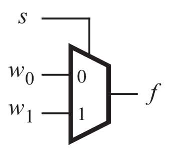
- Commonly used combinational subcircuits
- Multiplexers, which can be used for selection of signals and for implementation of general logic functions
- Circuits used for encoding, decoding, and code-conversion purposes
- Key VHDL constructs used to define combinational circuits

Previous chapters have introduced the basic techniques for design of logic circuits. In practice, a few types of logic circuits are often used as building blocks in larger designs. This chapter discusses a number of these blocks and gives examples of their use. The chapter also includes a major section on VHDL, which describes several key features of the language.

6.1 Multiplexers

Multiplexers were introduced briefly in Chapters 2 and 3. A multiplexer circuit has a number of data inputs, one or more select inputs, and one output. It passes the signal value on one of the data inputs to the output. The data input is selected by the values of the select inputs. Figure 6.1 shows a 2-to-1 multiplexer. Part (*a*) gives the symbol commonly used. The *select* input, s, chooses as the output of the multiplexer either input w_0 or w_1. The multiplexer's functionality can be described in the form of a truth table as shown in part (*b*) of the figure. Part (*c*) gives a sum-of-products implementation of the 2-to-1 multiplexer, and part (*d*) illustrates how it can be constructed with transmission gates.

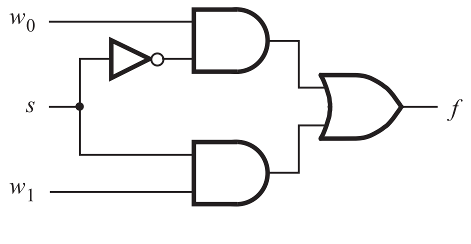
Figure 6.2*a* depicts a larger multiplexer with four data inputs, $w_0, \ldots, w_3$, and two select inputs, s_1 and s_0. As shown in the truth table in part (*b*) of the figure, the two-bit number represented by s_1s_0 selects one of the data inputs as the output of the multiplexer.



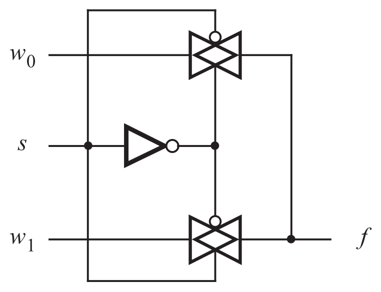
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

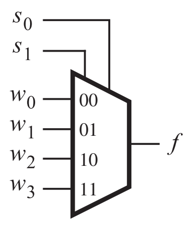


(c) Sum-of-products circuit



(d) Circuit with transmission gates

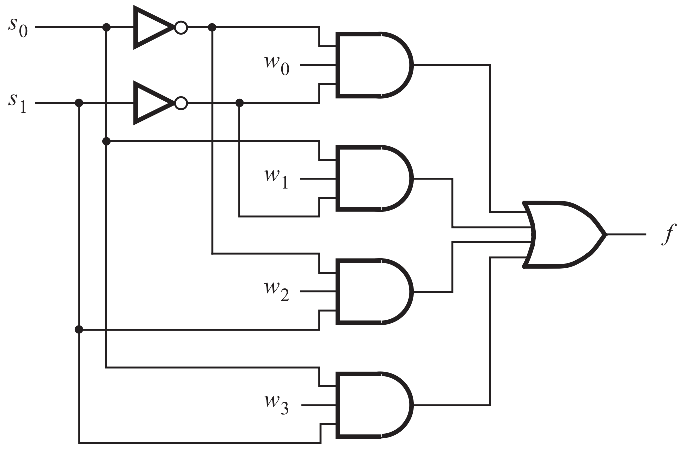
Figure 6.1 A 2-to-1 multiplexer.



(a) Graphical symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table



(c) Circuit

Figure 6.2 A 4-to-1 multiplexer.

A sum-of-products implementation of the 4-to-1 multiplexer appears in Figure 6.2*c*. It realizes the multiplexer function

$$f = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1\overline{s}_0w_2 + s_1s_0w_3$$

It is possible to build larger multiplexers using the same approach. Usually, the number of data inputs, n, is an integer power of two. A multiplexer that has n data inputs, $w_0, \ldots, w_{n-1}$, requires $\lceil \log_2 n \rceil$ select inputs. Larger multiplexers can also be constructed from smaller multiplexers. For example, the 4-to-1 multiplexer can be built using three 2-to-1 multiplexers as illustrated in Figure 6.3. If the 4-to-1 multiplexer is implemented using transmission gates, then the structure in this figure is always used. Figure 6.4 shows how a 16-to-1 multiplexer is constructed with five 4-to-1 multiplexers.

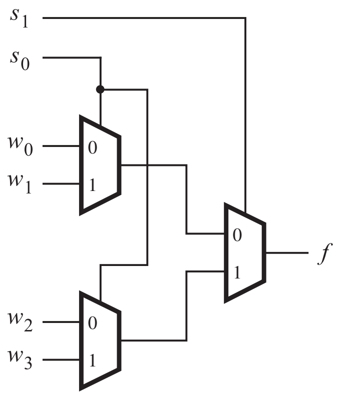


Figure 6.3 Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

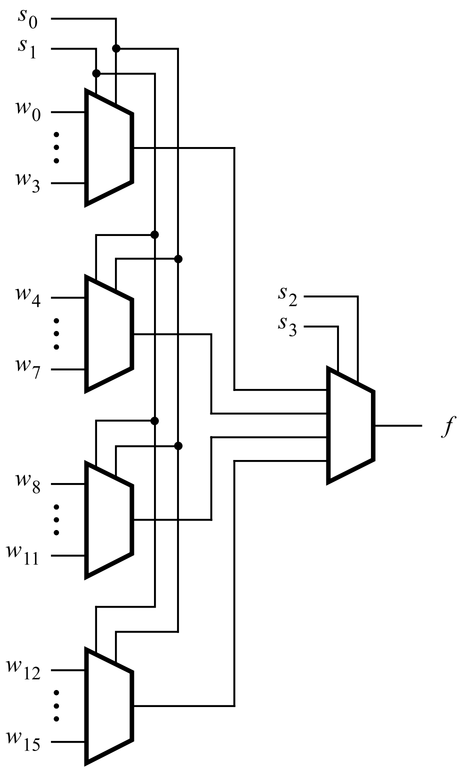


Figure 6.4 A 16-to-1 multiplexer.

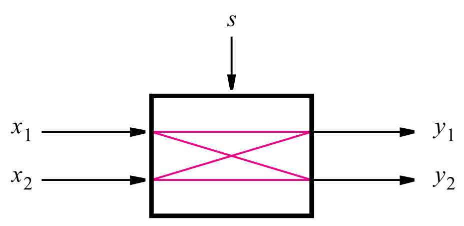
Example 6.1

Figure 6.5 shows a circuit that has two inputs, x_1 and x_2, and two outputs, y_1 and y_2. As indicated by the blue lines, the function of the circuit is to allow either of its inputs to be connected to either of its outputs, under the control of another input, s. A circuit that has n inputs and k outputs, whose sole function is to provide a capability to connect any input to any output, is usually referred to as an $n \times k$ crossbar switch. Crossbars of various sizes can be created, with different numbers of inputs and outputs. When there are two inputs and two outputs, it is called a 2×2 crossbar.

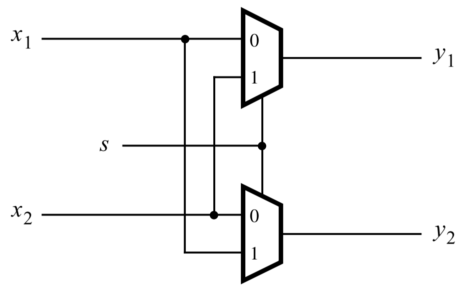
Figure 6.5*b* shows how the 2×2 crossbar can be implemented using 2-to-1 multiplexers. The multiplexer select inputs are controlled by the signal s. If $s = 0$, the crossbar connects x_1 to y_1 and x_2 to y_2, while if $s = 1$, the crossbar connects x_1 to y_2 and x_2 to y_1. Crossbar switches are useful in many practical applications in which it is necessary to be able to connect one set of wires to another set of wires, where the connection pattern changes from time to time.

Example 6.2

We introduced field-programmable gate array (FPGA) chips in section 3.6.5. Figure 3.39 depicts a small FPGA that is programmed to implement a particular circuit. The logic blocks in the FPGA have two inputs, and there are four tracks in each routing channel. Each of the programmable switches that connects a logic block input or output to an interconnection wire is shown as an X. A small part of Figure 3.39 is reproduced in Figure 6.6*a*. For clarity,

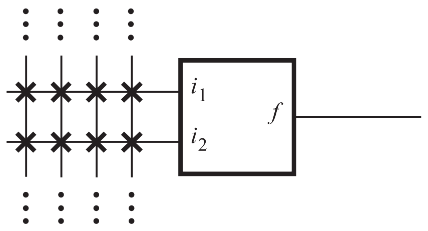


(a) A 2x2 crossbar switch

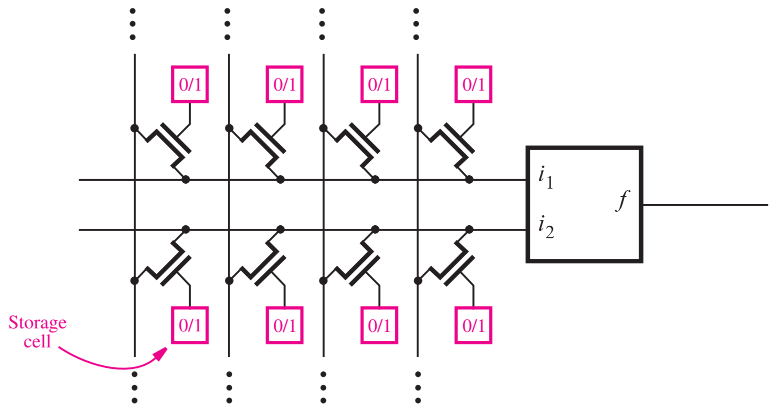


(b) Implementation using multiplexers

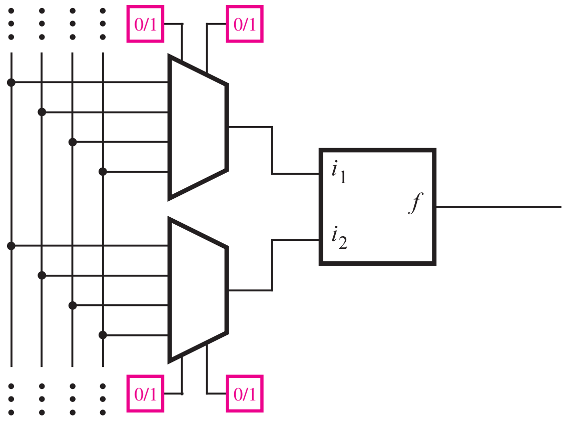
Figure 6.5 A practical application of multiplexers.



(a) Part of the FPGA in Figure 3.39



(b) Implementation using pass transistors



(c) Implementation using multiplexers

Figure 6.6 Implementing programmable switches in an FPGA.

the figure shows only a single logic block and the interconnection wires and switches associated with its input terminals.

One way in which the programmable switches can be implemented is illustrated in Figure 6.6*b*. Each X in part (*a*) of the figure is realized using an NMOS transistor controlled by a storage cell. This type of programmable switch was also shown in Figure 3.68. We described storage cells briefly in section 3.6.5 and will discuss them in more detail in section 10.1. Each cell stores a single logic value, either 0 or 1, and provides this value as the output of the cell. Each storage cell is built by using several transistors. Thus the eight cells shown in the figure use a significant amount of chip area.

The number of storage cells needed can be reduced by using multiplexers, as shown in Figure 6.6*c*. Each logic block input is fed by a 4-to-1 multiplexer, with the select inputs controlled by storage cells. This approach requires only four storage cells, instead of eight. In commercial FPGAs the multiplexer-based approach is usually adopted.

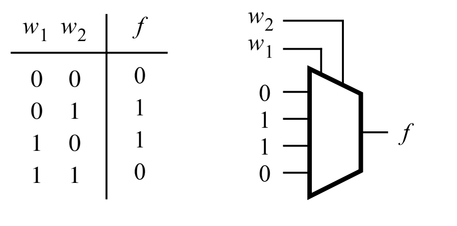
6.1.1 SYNTHESIS OF LOGIC FUNCTIONS USING MULTIPLEXERS

Multiplexers are useful in many practical applications, such as those described above. They can also be used in a more general way to synthesize logic functions. Consider the example in Figure 6.7*a*. The truth table defines the function $f = w_1 \oplus w_2$. This function can be implemented by a 4-to-1 multiplexer in which the values of f in each row of the truth table are connected as constants to the multiplexer data inputs. The multiplexer select inputs are driven by w_1 and w_2. Thus for each valuation of $w_1 w_2$, the output f is equal to the function value in the corresponding row of the truth table.

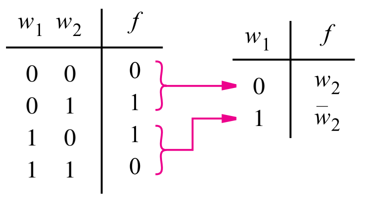
The above implementation is straightforward, but it is not very efficient. A better implementation can be derived by manipulating the truth table as indicated in Figure 6.7*b*, which allows f to be implemented by a single 2-to-1 multiplexer. One of the input signals, w_1 in this example, is chosen as the select input of the 2-to-1 multiplexer. The truth table is redrawn to indicate the value of f for each value of w_1. When $w_1 = 0, f$ has the same value as input w_2, and when $w_1 = 1, f$ has the value of $\overline{w}_2$. The circuit that implements this truth table is given in Figure 6.7*c*. This procedure can be applied to synthesize a circuit that implements any logic function.

Example 6.3

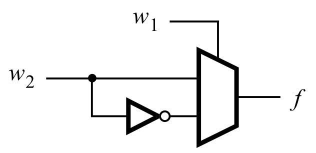
Figure 6.8*a* gives the truth table for the three-input majority function, and it shows how the truth table can be modified to implement the function using a 4-to-1 multiplexer. Any two of the three inputs may be chosen as the multiplexer select inputs. We have chosen w_1 and w_2 for this purpose, resulting in the circuit in Figure 6.8*b*.



(a) Implementation using a 4-to-1 multiplexer



(b) Modified truth table



(c) Circuit

Figure 6.7 Synthesis of a logic function using mutiplexers.

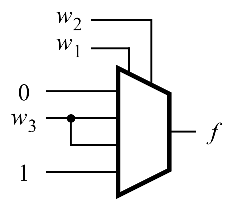
Example 6.4 Figure 6.9*a* indicates how the function $f = w_1 \oplus w_2 \oplus w_3$ can be implemented using 2-to-1 multiplexers. When $w_1 = 0, f$ is equal to the XOR of w_2 and w_3, and when $w_1 = 1, f$ is the XNOR of w_2 and w_3. The left multiplexer in the circuit produces $w_2 \oplus w_3$, using the result from Figure 6.7, and the right multiplexer uses the value of w_1 to select either $w_2 \oplus w_3$ or its complement. Note that we could have derived this circuit directly by writing the function as $f = (w_2 \oplus w_3) \oplus w_1$.

Figure 6.10 gives an implementation of the three-input XOR function using a 4-to-1 multiplexer. Choosing w_1 and w_2 for the select inputs results in the circuit shown.

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

(a) Modified truth table



(b) Circuit

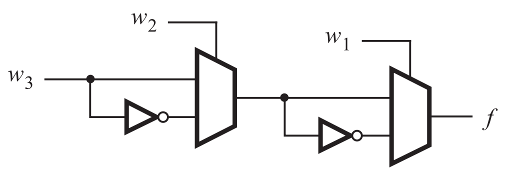
Figure 6.8 Implementation of the three-input majority function using a 4-to-1 multiplexer.

w_1	w_2	w_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$w_2 \oplus w_3$

$\overline{w_2 \oplus w_3}$

(a) Truth table

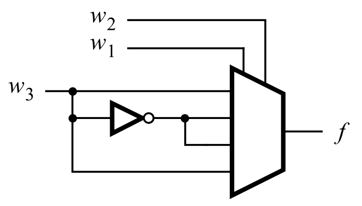


(b) Circuit

Figure 6.9 Three-input XOR implemented with 2-to-1 multiplexers.

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\overline{w}_3$
0	1	1	0	
1	0	0	1	$\overline{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

(a) Truth table



(b) Circuit

Figure 6.10 Three-input XOR implemented with a 4-to-1 multiplexer.

6.1.2 MULTIPLEXER SYNTHESIS USING SHANNON'S EXPANSION

Figures 6.8 through 6.10 illustrate how truth tables can be interpreted to implement logic functions using multiplexers. In each case the inputs to the multiplexers are the constants 0 and 1, or some variable or its complement. Besides using such simple inputs, it is possible to connect more complex circuits as inputs to a multiplexer, allowing functions to be synthesized using a combination of multiplexers and other logic gates. Suppose that we want to implement the three-input majority function in Figure 6.8 using a 2-to-1 multiplexer in this way. Figure 6.11 shows an intuitive way of realizing this function. The truth table can be modified as shown on the right. If $w_1 = 0$, then $f = w_2w_3$, and if $w_1 = 1$, then $f = w_2 + w_3$. Using w_1 as the select input for a 2-to-1 multiplexer leads to the circuit in Figure 6.11*b*.

This implementation can be derived using algebraic manipulation as follows. The function in Figure 6.11*a* is expressed in sum-of-products form as

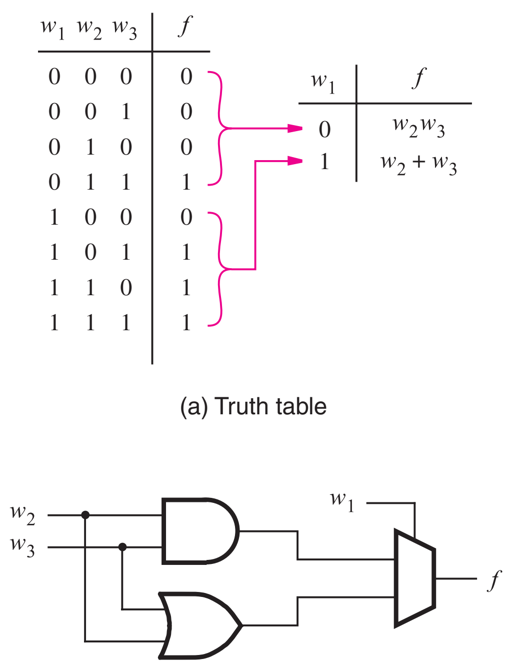
$$f = \overline{w}_1w_2w_3 + w_1\overline{w}_2w_3 + w_1w_2\overline{w}_3 + w_1w_2w_3$$

It can be manipulated into

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

which corresponds to the circuit in Figure 6.11*b*.

Multiplexer implementations of logic functions require that a given function be decomposed in terms of the variables that are used as the select inputs. This can be accomplished by means of a theorem proposed by Claude Shannon [1].



w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	$w_2 w_3$
1	$w_2 + w_3$

(a) Truth table

(b) Circuit

Figure 6.11 The three-input majority function implemented using a 2-to-1 multiplexer.

Shannon's Expansion Theorem

Any Boolean function $f(w_1, \ldots, w_n)$ can be written in the form

$$f(w_1, w_2, \ldots, w_n) = \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$

This expansion can be done in terms of any of the n variables. We will leave the proof of the theorem as an exercise for the reader (see problem 6.9).

To illustrate its use, we can apply the theorem to the three-input majority function, which can be written as

$$f(w_1, w_2, w_3) = w_1 w_2 + w_1 w_3 + w_2 w_3$$

Expanding this function in terms of w_1 gives

$$f = \overline{w}_1(w_2 w_3) + w_1(w_2 + w_3)$$

which is the expression that we derived above.

For the three-input XOR function, we have

$$\begin{aligned} f &= w_1 \oplus w_2 \oplus w_3 \\ &= \overline{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot (\overline{w_2 \oplus w_3}) \end{aligned}$$

which gives the circuit in Figure 6.9*b*.

In Shannon's expansion the term $f(0, w_2, \ldots, w_n)$ is called the *cofactor* of f with respect to $\overline{w}_1$; it is denoted in shorthand notation as $f_{\overline{w}_1}$. Similarly, the term $f(1, w_2, \ldots, w_n)$ is called the cofactor of f with respect to w_1, written f_{w_1}. Hence we can write

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

In general, if the expansion is done with respect to variable w_i, then f_{w_i} denotes $f(w_1, \ldots, w_{i-1}, 1, w_{i+1}, \ldots, w_n)$ and

$$f(w_1, \ldots, w_n) = \overline{w}_i f_{\overline{w}_i} + w_i f_{w_i}$$

The complexity of the logic expression may vary, depending on which variable, w_i, is used, as illustrated in Example 6.5.

Example 6.5 For the function $f = \overline{w}_1 w_3 + w_2 \overline{w}_3$, decomposition using w_1 gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1 (w_3 + w_2) + w_1 (w_2 \overline{w}_3) \end{aligned}$$

Using w_2 instead of w_1 produces

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2 (\overline{w}_1 w_3) + w_2 (\overline{w}_1 + \overline{w}_3) \end{aligned}$$

Finally, using w_3 gives

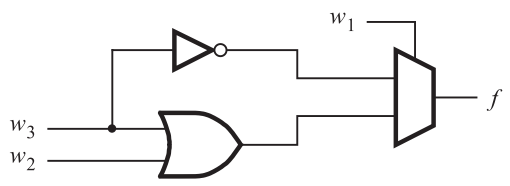
$$\begin{aligned} f &= \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3} \\ &= \overline{w}_3 (w_2) + w_3 (\overline{w}_1) \end{aligned}$$

The results generated using w_1 and w_2 have the same cost, but the expression produced using w_3 has a lower cost. In practice, the CAD tools that perform decompositions of this type try a number of alternatives and choose the one that produces the best result.

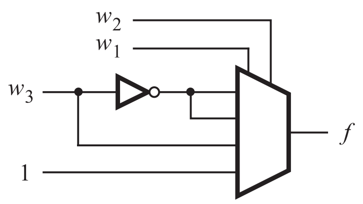
Shannon's expansion can be done in terms of more than one variable. For example, expanding a function in terms of w_1 and w_2 gives

$$\begin{aligned} f(w_1, \ldots, w_n) = {} & \overline{w}_1 \overline{w}_2 \cdot f(0, 0, w_3, \ldots, w_n) + \overline{w}_1 w_2 \cdot f(0, 1, w_3, \ldots, w_n) \\ & + w_1 \overline{w}_2 \cdot f(1, 0, w_3, \ldots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n) \end{aligned}$$

This expansion gives a form that can be implemented using a 4-to-1 multiplexer. If Shannon's expansion is done in terms of all n variables, then the result is the canonical sum-of-products form, which was defined in section 2.6.1.



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

Figure 6.12 The circuits synthesized in Example 6.6.

Example 6.6

Assume that we wish to implement the function

$$f = \overline{w}_1\overline{w}_3 + w_1w_2 + w_1w_3$$

using a 2-to-1 multiplexer and any other necessary gates. Shannon's expansion using w_1 gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3) \end{aligned}$$

The corresponding circuit is shown in Figure 6.12*a*. Assume now that we wish to use a 4-to-1 multiplexer instead. Further decomposition using w_2 gives

$$\begin{aligned} f &= \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1w_2 f_{w_1w_2} \\ &= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1w_2(1) \end{aligned}$$

The circuit is shown in Figure 6.12*b*.

Example 6.7

Consider the three-input majority function

$$f = w_1w_2 + w_1w_3 + w_2w_3$$

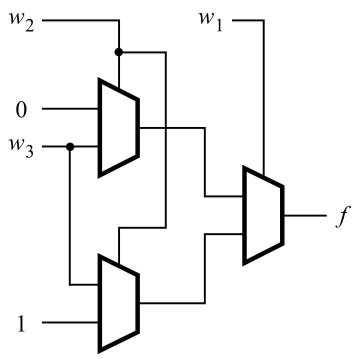


Figure 6.13 The circuit synthesized in Example 6.7.

We wish to implement this function using only 2-to-1 multiplexers. Shannon's expansion using w_1 yields

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

Let $g = w_2w_3$ and $h = w_2 + w_3$. Expansion of both g and h using w_2 gives

$$\begin{aligned} g &= \overline{w}_2(0) + w_2(w_3) \\ h &= \overline{w}_2(w_3) + w_2(1) \end{aligned}$$

The corresponding circuit is shown in Figure 6.13. It is equivalent to the 4-to-1 multiplexer circuit derived using a truth table in Figure 6.8.

Example 6.8 In section 3.6.5 we said that most FPGAs use lookup tables for their logic blocks. Assume that an FPGA exists in which each logic block is a three-input lookup table (3-LUT). Because it stores a truth table, a 3-LUT can realize any logic function of three variables. Using Shannon's expansion, any four-variable function can be realized with at most three 3-LUTs. Consider the function

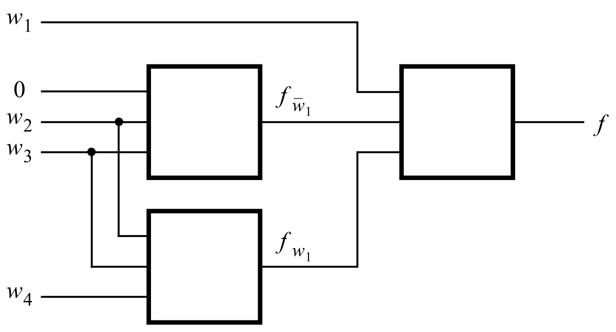
$$f = \overline{w}_2w_3 + \overline{w}_1w_2\overline{w}_3 + w_2\overline{w}_3w_4 + w_1\overline{w}_2\overline{w}_4$$

Expansion in terms of w_1 produces

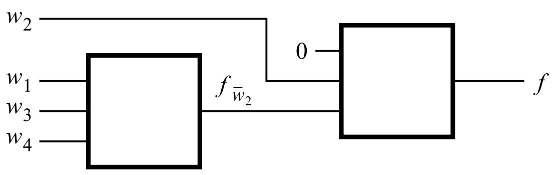
$$\begin{aligned} f &= \overline{w}_1f_{\overline{w}_1} + w_1f_{w_1} \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2\overline{w}_3w_4) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \end{aligned}$$

A circuit with three 3-LUTs that implements this expression is shown in Figure 6.14*a*. Decomposition of the function using w_2, instead of w_1, gives

$$\begin{aligned} f &= \overline{w}_2f_{\overline{w}_2} + w_2f_{w_2} \\ &= \overline{w}_2(w_3 + w_1\overline{w}_4) + w_2(\overline{w}_1\overline{w}_3 + \overline{w}_3w_4) \end{aligned}$$



(a) Using three 3-LUTs



(b) Using two 3-LUTs

Figure 6.14 Circuits synthesized in Example 6.8.

Observe that $\overline{f}_{\overline{w}_2} = f_{w_2}$; hence only two 3-LUTs are needed, as illustrated in Figure 6.14*b*. The LUT on the right implements the two-variable function $\overline{w}_2 f_{\overline{w}_2} + w_2 \overline{f}_{\overline{w}_2}$.

Since it is possible to implement any logic function using multiplexers, general-purpose chips exist that contain multiplexers as their basic logic resources. Both Actel Corporation [2] and QuickLogic Corporation [3] offer FPGAs in which the logic block comprises an arrangement of multiplexers. Texas Instruments offers gate array chips that have multiplexer-based logic blocks [4].

6.2 Decoders

Decoder circuits are used to decode encoded information. A binary decoder, depicted in Figure 6.15, is a logic circuit with n inputs and 2^n outputs. Only one output is asserted at a time, and each output corresponds to one valuation of the inputs. The decoder also has an enable input, *En*, that is used to disable the outputs; if $En = 0$, then none of the decoder outputs is asserted. If $En = 1$, the valuation of $w_{n-1} \cdots w_1 w_0$ determines which of the outputs is asserted. An n-bit binary code in which exactly one of the bits is set to 1 at a

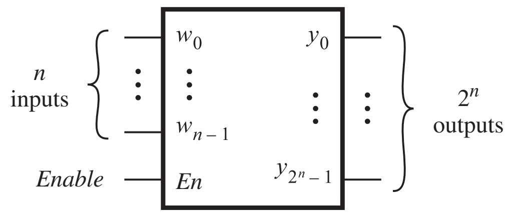


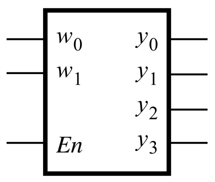
Figure 6.15 An n-to-2^n binary decoder.

time is referred to as *one-hot encoded*, meaning that the single bit that is set to 1 is deemed to be "hot." The outputs of a binary decoder are one-hot encoded.

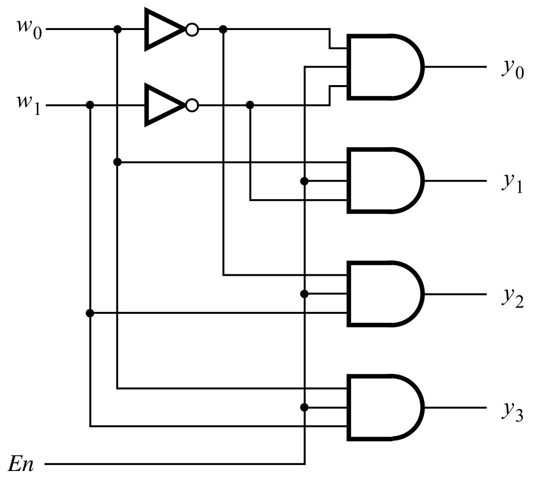
A 2-to-4 decoder is given in Figure 6.16. The two data inputs are w_1 and w_0. They represent a two-bit number that causes the decoder to assert one of the outputs $y_0, \ldots, y_3$. Although a decoder can be designed to have either active-high or active-low outputs, in

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



(b) Graphical symbol



(c) Logic circuit

Figure 6.16 A 2-to-4 decoder.

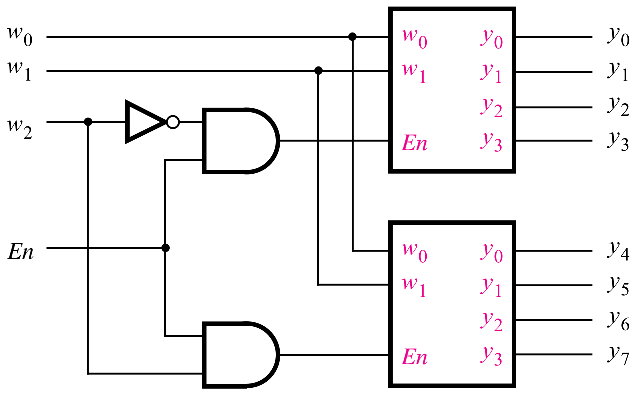


Figure 6.17 A 3-to-8 decoder using two 2-to-4 decoders.

Figure 6.16 active-high outputs are assumed. Setting the inputs w_1w_0 to 00, 01, 10, or 11 causes the output y_0, y_1, y_2, or y_3 to be set to 1, respectively. A graphical symbol for the decoder is given in part (*b*) of the figure, and a logic circuit is shown in part (*c*).

Larger decoders can be built using the sum-of-products structure in Figure 6.16*c*, or else they can be constructed from smaller decoders. Figure 6.17 shows how a 3-to-8 decoder is built with two 2-to-4 decoders. The w_2 input drives the enable inputs of the two decoders. The top decoder is enabled if $w_2 = 0$, and the bottom decoder is enabled if $w_2 = 1$. This concept can be applied for decoders of any size. Figure 6.18 shows how five 2-to-4 decoders can be used to construct a 4-to-16 decoder. Because of its treelike structure, this type of circuit is often referred to as a *decoder tree*.

Example 6.9

Decoders are useful for many practical purposes. In Figure 6.2*c* we showed the sum-of-products implementation of the 4-to-1 multiplexer, which requires AND gates to distinguish the four different valuations of the select inputs s_1 and s_0. Since a decoder evaluates the values on its inputs, it can be used to build a multiplexer as illustrated in Figure 6.19. The enable input of the decoder is not needed in this case, and it is set to 1. The four outputs of the decoder represent the four valuations of the select inputs.

Example 6.10

In Figure 3.59 we showed how a 2-to-1 multiplexer can be constructed using two tri-state buffers. This concept can be applied to any size of multiplexer, with the addition of a decoder. An example is shown in Figure 6.20. The decoder enables one of the tri-state buffers for each valuation of the select lines, and that tri-state buffer drives the output, f, with the selected data input. We have now seen that multiplexers can be implemented in various ways. The choice of whether to employ the sum-of-products form, transmission gates, or tri-state buffers depends on the resources available in the chip being used. For instance, most FPGAs that use lookup tables for their logic blocks do not contain tri-state

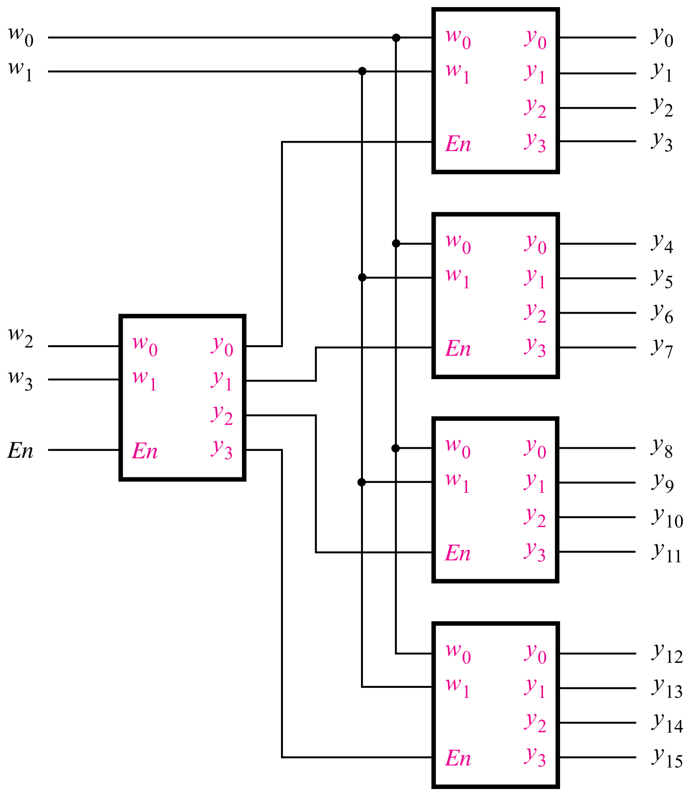


Figure 6.18 A 4-to-16 decoder built using a decoder tree.

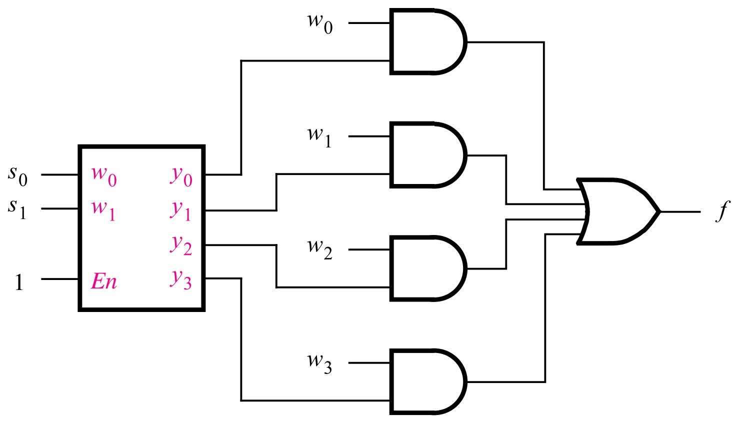


Figure 6.19 A 4-to-1 multiplexer built using a decoder.

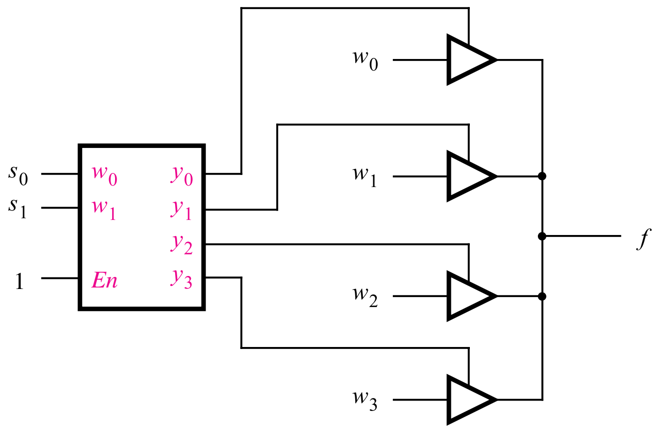


Figure 6.20 A 4-to-1 multiplexer built using a decoder and tri-state buffers.

buffers. Hence multiplexers must be implemented in the sum-of-products form using the lookup tables (see Example 6.30).

6.2.1 DEMULTIPLEXERS

We showed in section 6.1 that a multiplexer has one output, n data inputs, and $\lceil \log_2 n \rceil$ select inputs. The purpose of the multiplexer circuit is to *multiplex* the n data inputs onto the single data output under control of the select inputs. A circuit that performs the opposite function, namely, placing the value of a single data input onto multiple data outputs, is called a *demultiplexer*. The demultiplexer can be implemented using a decoder circuit. For example, the 2-to-4 decoder in Figure 6.16 can be used as a 1-to-4 demultiplexer. In this case the En input serves as the data input for the demultiplexer, and the y_0 to y_3 outputs are the data outputs. The valuation of w_1w_0 determines which of the outputs is set to the value of En. To see how the circuit works, consider the truth table in Figure 6.16*a*. When $En = 0$, all the outputs are set to 0, including the one selected by the valuation of w_1w_0. When $En = 1$, the valuation of w_1w_0 sets the appropriate output to 1.

In general, an n-to-2^n decoder circuit can be used as a 1-to-n demultiplexer. However, in practice decoder circuits are used much more often as decoders rather than as demultiplexers. In many applications the decoder's En input is not actually needed; hence it can be omitted. In this case the decoder always asserts one of its data outputs, $y_0, \ldots, y_{2^n-1}$, according to the valuation of the data inputs, $w_{n-1} \cdots w_0$. Example 6.11 uses a decoder that does not have the En input.

Example 6.11 One of the most important applications of decoders is in memory blocks, which are used to store information. Such memory blocks are included in digital systems, such as computers, where there is a need to store large amounts of information electronically. One type of memory block is called a *read-only memory* (ROM). A ROM consists of a collection of storage cells, where each cell permanently stores a single logic value, either 0 or 1. Figure 6.21 shows an example of a ROM block. The storage cells are arranged in 2^m rows with n cells per row. Thus each row stores n bits of information. The location of each row in the ROM is identified by its *address*. In the figure the row at the top of the ROM has address 0, and the row at the bottom has address $2^m - 1$. The information stored in the rows can be accessed by asserting the select lines, Sel_0 to Sel_{2^m-1}. As shown in the figure, a decoder with m inputs and 2^m outputs is used to generate the signals on the select lines. Since the inputs to the decoder choose the particular address (row) selected, they are called the *address* lines. The information stored in the row appears on the data outputs of the ROM, $d_{n-1}, \ldots, d_0$, which are called the *data* lines. Figure 6.21 shows that each data line has an associated tri-state buffer that is enabled by the ROM input named *Read*. To access, or *read*, data from the ROM, the address of the desired row is placed on the address lines and *Read* is set to 1.

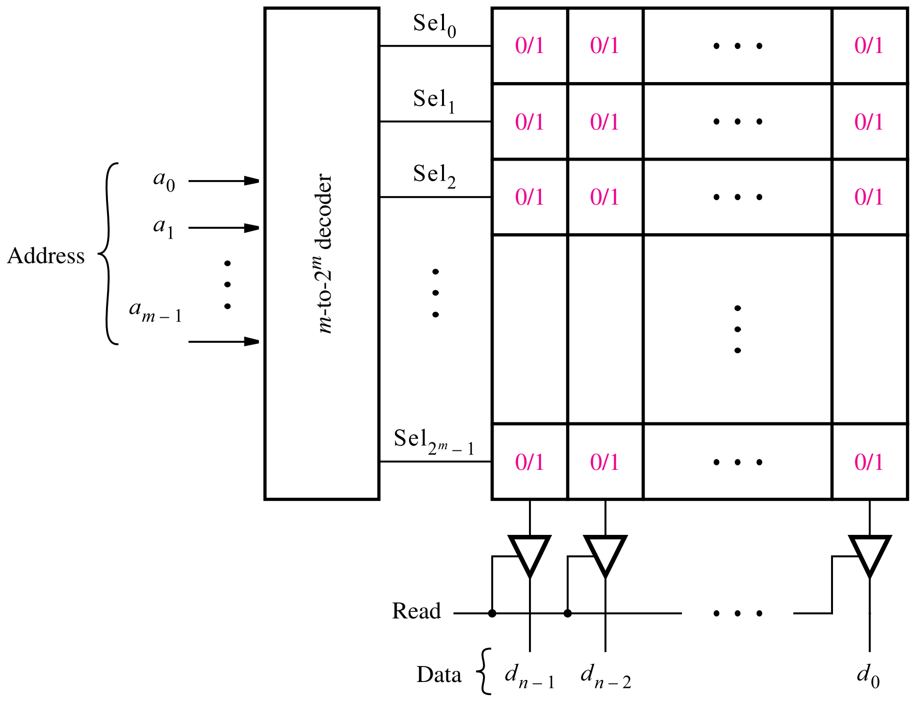


Figure 6.21 A $2^m \times n$ read-only memory (ROM) block.

Many different types of memory blocks exist. In a ROM the stored information can be read out of the storage cells, but it cannot be changed (see problem 6.32). Another type of ROM allows information to be both read out of the storage cells and stored, or *written*, into them. Reading its contents is the normal operation, whereas writing requires a special procedure. Such a memory block is called a programmable ROM (PROM). The storage cells in a PROM are usually implemented using EEPROM transistors. We discussed EEPROM transistors in section 3.10 to show how they are used in PLDs. Other types of memory blocks are discussed in section 10.1.

6.3 ENCODERS

An encoder performs the opposite function of a decoder. It encodes given information into a more compact form.

6.3.1 BINARY ENCODERS

A *binary encoder* encodes information from 2^n inputs into an n-bit code, as indicated in Figure 6.22. Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1. The truth table for a 4-to-2 encoder is provided in Figure 6.23*a*. Observe that the output y_0 is 1 when either input w_1 or w_3 is 1, and output y_1 is 1 when input w_2 or w_3 is 1. Hence these outputs can be generated by the circuit in Figure 6.23*b*. Note that we assume that the inputs are one-hot encoded. All input patterns that have multiple inputs set to 1 are not shown in the truth table, and they are treated as don't-care conditions.

Encoders are used to reduce the number of bits needed to represent given information. A practical use of encoders is for transmitting information in a digital system. Encoding the information allows the transmission link to be built using fewer wires. Encoding is also useful if information is to be stored for later use because fewer bits need to be stored.

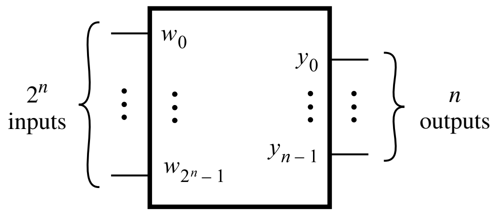
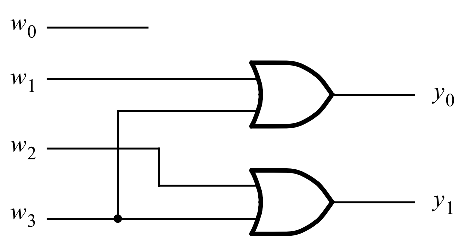


Figure 6.22 A 2^n-to-n binary encoder.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

Figure 6.23 A 4-to-2 binary encoder.

6.3.2 Priority Encoders

Another useful class of encoders is based on the priority of input signals. In a *priority encoder* each input has a priority level associated with it. The encoder outputs indicate the active input that has the highest priority. When an input with a high priority is asserted, the other inputs with lower priority are ignored. The truth table for a 4-to-2 priority encoder is shown in Figure 6.24. It assumes that w_0 has the lowest priority and w_3 the highest. The outputs y_1 and y_0 represent the binary number that identifies the highest priority input set to 1. Since it is possible that none of the inputs is equal to 1, an output, z, is provided to indicate this condition. It is set to 1 when at least one of the inputs is equal to 1. It is set to

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figure 6.24 Truth table for a 4-to-2 priority encoder.

0 when all inputs are equal to 0. The outputs y_1 and y_0 are not meaningful in this case, and hence the first row of the truth table can be treated as a don't-care condition for y_1 and y_0.

The behavior of the priority encoder is most easily understood by first considering the last row in the truth table. It specifies that if input w_3 is 1, then the outputs are set to $y_1y_0 = 11$. Because w_3 has the highest priority level, the values of inputs w_2, w_1, and w_0 do not matter. To reflect the fact that their values are irrelevant, w_2, w_1, and w_0 are denoted by the symbol x in the truth table. The second-last row in the truth table stipulates that if $w_2 = 1$, then the outputs are set to $y_1y_0 = 10$, but only if $w_3 = 0$. Similarly, input w_1 causes the outputs to be set to $y_1y_0 = 01$ only if both w_3 and w_2 are 0. Input w_0 produces the outputs $y_1y_0 = 00$ only if w_0 is the only input that is asserted.

A logic circuit that implements the truth table can be synthesized by using the techniques developed in Chapter 4. However, a more convenient way to derive the circuit is to define a set of intermediate signals, $i_0, \ldots, i_3$, based on the observations above. Each signal, i_k, is equal to 1 only if the input with the same index, w_k, represents the highest-priority input that is set to 1. The logic expressions for $i_0, \ldots, i_3$ are

$$\begin{aligned} i_0 &= \overline{w}_3\overline{w}_2\overline{w}_1w_0 \\ i_1 &= \overline{w}_3\overline{w}_2w_1 \\ i_2 &= \overline{w}_3w_2 \\ i_3 &= w_3 \end{aligned}$$

Using the intermediate signals, the rest of the circuit for the priority encoder has the same structure as the binary encoder in Figure 6.23, namely

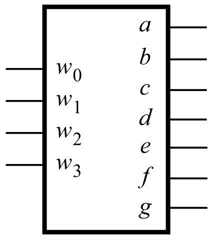
$$\begin{aligned} y_0 &= i_1 + i_3 \\ y_1 &= i_2 + i_3 \end{aligned}$$

The output z is given by

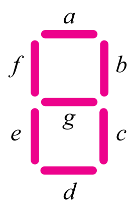
$$z = i_0 + i_1 + i_2 + i_3$$

6.4 CODE CONVERTERS

The purpose of the decoder and encoder circuits is to convert from one type of input encoding to a different output encoding. For example, a 3-to-8 binary decoder converts from a binary number on the input to a one-hot encoding at the output. An 8-to-3 binary encoder performs the opposite conversion. There are many other possible types of code converters. One common example is a BCD-to-7-segment decoder, which converts one binary-coded decimal (BCD) digit into information suitable for driving a digit-oriented display. As illustrated in Figure 6.25*a*, the circuit converts the BCD digit into seven signals that are used to drive the segments in the display. Each segment is a small light-emitting diode (LED), which glows when driven by an electrical signal. The segments are labeled from *a* to *g* in the figure. The truth table for the BCD-to-7-segment decoder is given in Figure 6.25*c*. For each valuation of the inputs $w_3, \ldots, w_0$, the seven outputs are set to



(a) Code converter



(b) 7-segment display

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
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	1	0	1	1

(c) Truth table

Figure 6.25 A BCD-to-7-segment display code converter.

display the appropriate BCD digit. Note that the last 6 rows of a complete 16-row truth table are not shown. They represent don't-care conditions because they are not legal BCD codes and will never occur in a circuit that deals with BCD data. A circuit that implements the truth table can be derived using the synthesis techniques discussed in Chapter 4. Finally, we should note that although the word *decoder* is traditionally used for this circuit, a more appropriate term is *code converter*. The term *decoder* is more appropriate for circuits that produce one-hot encoded outputs.

6.5 Arithmetic Comparison Circuits

Chapter 5 presented arithmetic circuits that perform addition, subtraction, and multiplication of binary numbers. Another useful type of arithmetic circuit compares the relative sizes of two binary numbers. Such a circuit is called a *comparator*. This section considers the design of a comparator that has two n-bit inputs, A and B, which represent unsigned binary numbers. The comparator produces three outputs, called $AeqB$, $AgtB$, and $AltB$. The $AeqB$ output is set to 1 if A and B are equal. The $AgtB$ output is 1 if A is greater than B, and the $AltB$ output is 1 if A is less than B.

The desired comparator can be designed by creating a truth table that specifies the three outputs as functions of A and B. However, even for moderate values of n, the truth table is large. A better approach is to derive the comparator circuit by considering the bits of A and B in pairs. We can illustrate this by a small example, where $n = 4$.

Let $A = a_3a_2a_1a_0$ and $B = b_3b_2b_1b_0$. Define a set of intermediate signals called i_3, i_2, i_1, and i_0. Each signal, i_k, is 1 if the bits of A and B with the same index are equal. That is, $i_k = \overline{a_k \oplus b_k}$. The comparator's $AeqB$ output is then given by

$$AeqB = i_3i_2i_1i_0$$

An expression for the $AgtB$ output can be derived by considering the bits of A and B in the order from the most-significant bit to the least-significant bit. The first bit-position, k, at which a_k and b_k differ determines whether A is less than or greater than B. If $a_k = 0$ and $b_k = 1$, then $A < B$. But if $a_k = 1$ and $b_k = 0$, then $A > B$. The $AgtB$ output is defined by

$$AgtB = a_3\overline{b}_3 + i_3a_2\overline{b}_2 + i_3i_2a_1\overline{b}_1 + i_3i_2i_1a_0\overline{b}_0$$

The i_k signals ensure that only the first digits, considered from the left to the right, of A and B that differ determine the value of $AgtB$.

The $AltB$ output can be derived by using the other two outputs as

$$AltB = \overline{AeqB + AgtB}$$

A logic circuit that implements the four-bit comparator circuit is shown in Figure 6.26. This approach can be used to design a comparator for any value of n.

Comparator circuits, like most logic circuits, can be designed in different ways. Another approach for designing a comparator circuit is presented in Example 5.10 in Chapter 5.

6.6 VHDL for Combinational Circuits

Having presented a number of useful circuits that can be used as building blocks in larger circuits, we will now consider how such circuits can be described in VHDL. Rather than relying on the simple VHDL statements used in previous examples, such as logic expressions, we will specify the circuits in terms of their behavior. We will also introduce a number of new VHDL constructs.

6.6.1 Assignment Statements

VHDL provides several types of statements that can be used to assign logic values to signals. In the examples of VHDL code given so far, only simple assignment statements have been used, either for logic or arithmetic expressions. This section introduces other types of

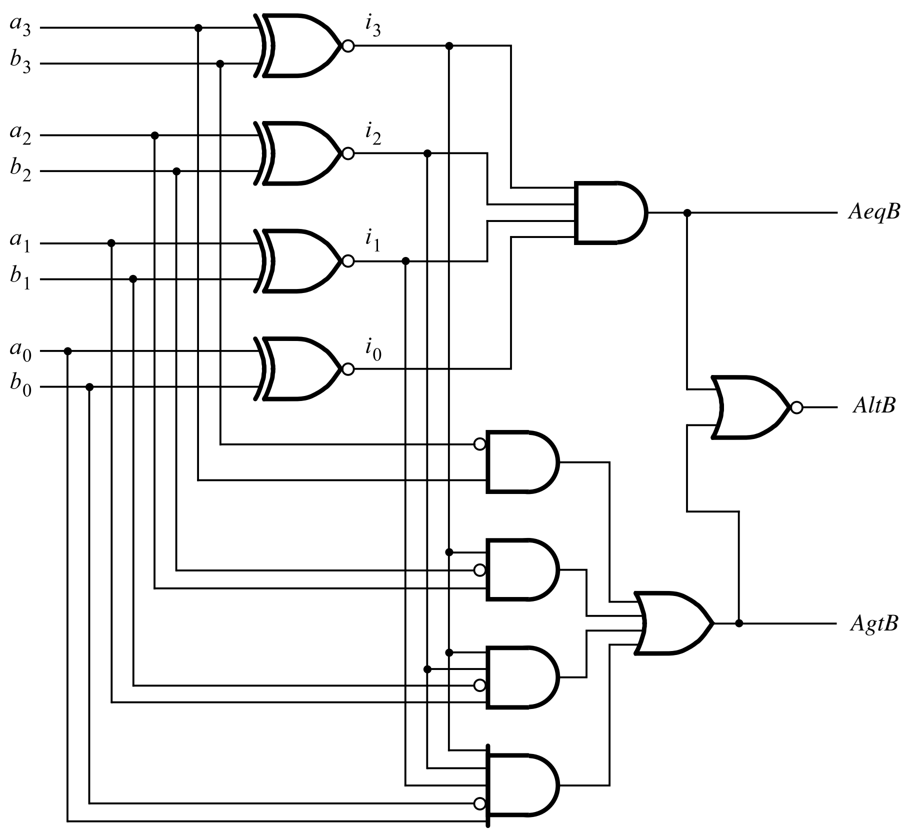


Figure 6.26 A four-bit comparator circuit.

assignment statements, which are called selected signal assignments, conditional signal assignments, generate statements, if-then-else statements, and case statements.

6.6.2 Selected Signal Assignment

A selected signal assignment allows a signal to be assigned one of several values, based on a selection criterion. Figure 6.27 shows how it can be used to describe a 2-to-1 multiplexer. The entity, named *mux2to1*, has the inputs w_0, w_1, and s, and the output f. The selected signal assignment begins with the keyword WITH, which specifies that s is to be used for the selection criterion. The two WHEN clauses state that f is assigned the value of w_0 when $s = 0$; otherwise, f is assigned the value of w_1. The WHEN clause that selects w_1 uses the word OTHERS, instead of the value 1. This is required because the VHDL syntax specifies that a WHEN clause must be included for every possible value of the selection signal s.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
              f        : OUT  STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    WITH s SELECT
        f <=  w0 WHEN '0',
              w1 WHEN OTHERS ;
END Behavior ;
```

Figure 6.27 VHDL code for a 2-to-1 multiplexer.

Since it has the STD_LOGIC type, discussed in section 4.12, s can take the values 0, 1, Z, −, and others. The keyword OTHERS provides a convenient way of accounting for all logic values that are not explicitly listed in a WHEN clause.

Example 6.12

A 4-to-1 multiplexer is described by the entity named *mux4to1*, shown in Figure 6.28. The two select inputs, which are called s_1 and s_0 in Figure 6.2, are represented by the two-bit STD_LOGIC_VECTOR signal s. The selected signal assignment sets f to the value of one of the inputs $w_0, \ldots, w_3$, depending on the valuation of s. Compiling the code results in the circuit shown in Figure 6.2c. At the end of Figure 6.28, the *mux4to1* entity is defined as a component in the package named *mux4to1_package*. We showed in section 5.5.2 that the component declaration allows the entity to be used as a subcircuit in other VHDL code.

Example 6.13

Figure 6.4 showed how a 16-to-1 multiplexer is built using five 4-to-1 multiplexers. Figure 6.29 presents VHDL code for this circuit, using the *mux4to1* component. The lines of code are numbered so that we can easily refer to them. The *mux4to1_package* is included in the code, because it provides the component declaration for *mux4to1*.

The data inputs to the *mux16to1* entity are the 16-bit signal named w, and the select inputs are the four-bit signal named s. In the VHDL code signal names are needed for the outputs of the four 4-to-1 multiplexers on the left of Figure 6.4. Line 11 defines a four-bit signal named m for this purpose, and lines 13 to 16 instantiate the four multiplexers. For instance, line 13 corresponds to the multiplexer at the top left of Figure 6.4. Its first four ports, which correspond to $w_0, \ldots, w_3$ in Figure 6.28, are driven by the signals $w(0), \ldots, w(3)$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT ( w0, w1, w2, w3   : IN    STD_LOGIC ;
           s                : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           f                : OUT   STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Behavior OF mux4to1 IS
BEGIN
    WITH s SELECT
        f <=  w0 WHEN "00",
              w1 WHEN "01",
              w2 WHEN "10",
              w3 WHEN OTHERS ;
END Behavior ;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
PACKAGE mux4to1_package IS
    COMPONENT mux4to1
        PORT ( w0, w1, w2, w3   : IN    STD_LOGIC ;
               s                : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               f                : OUT   STD_LOGIC ) ;
    END COMPONENT ;
END mux4to1_package ;
```

Figure 6.28 VHDL code for a 4-to-1 multiplexer.

The syntax $s(1$ DOWNTO $0)$ is used to attach the signals $s(1)$ and $s(0)$ to the two-bit s port of the *mux4to1* component. The $m(0)$ signal is connected to the multiplexer's output port.

Line 17 instantiates the multiplexer on the right of Figure 6.4. The signals $m_0, \ldots, m_3$ are connected to its data inputs, and bits $s(3)$ and $s(2)$, which are specified by the syntax $s(3$ DOWNTO $2)$, are attached to the select inputs. The output port generates the *mux16to1* output f. Compiling the code results in the multiplexer function

$$f = \overline{s}_3\overline{s}_2\overline{s}_1\overline{s}_0 w_0 + \overline{s}_3\overline{s}_2\overline{s}_1 s_0 w_1 + \overline{s}_3\overline{s}_2 s_1\overline{s}_0 w_2 + \cdots + s_3 s_2 s_1 \overline{s}_0 w_{14} + s_3 s_2 s_1 s_0 w_{15}$$

Example 6.14 The selected signal assignments can also be used to describe other types of circuits. Figure 6.30 shows how a selected signal assignment can be used to describe the truth table for a 2-to-4 binary decoder. The entity is called *dec2to4*. The data inputs are the two-bit signal

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY work ;
USE work.mux4to1_package.all ;

ENTITY mux16to1 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(0 TO 15) ;
           s   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f   : OUT   STD_LOGIC ) ;
END mux16to1 ;

ARCHITECTURE Structure OF mux16to1 IS
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    Mux1: mux4to1 PORT MAP
        ( w(0), w(1), w(2), w(3), s(1 DOWNTO 0), m(0) ) ;
    Mux2: mux4to1 PORT MAP
        ( w(4), w(5), w(6), w(7), s(1 DOWNTO 0), m(1) ) ;
    Mux3: mux4to1 PORT MAP
        ( w(8), w(9), w(10), w(11), s(1 DOWNTO 0), m(2) ) ;
    Mux4: mux4to1 PORT MAP
        ( w(12), w(13), w(14), w(15), s(1 DOWNTO 0), m(3) ) ;
    Mux5: mux4to1 PORT MAP
        ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```

Figure 6.29 Hierarchical code for a 16-to-1 multiplexer.

named *w*, and the enable input is *En*. The four outputs are represented by the four-bit signal *y*.

In the truth table for the decoder in Figure 6.16*a*, the inputs are listed in the order *En* $w_1 w_0$. To represent these three signals, the VHDL code defines the three-bit signal named *Enw*. The statement *Enw* <= *En* & *w* uses the VHDL concatenate operator, which was discussed in section 5.5.4, to combine the *En* and *w* signals into the *Enw* signal. Hence $Enw(2) = En$, $Enw(1) = w_1$, and $Enw(0) = w_0$. The *Enw* signal is used as the selection signal in the selected signal assignment statement. It describes the truth table in Figure 6.16*a*. In the first four WHEN clauses, $En = 1$, and the decoder outputs have the same patterns as in the first four rows of the truth table. The last WHEN clause uses the OTHERS keyword and sets the decoder outputs to 0000, because it represents the cases where $En = 0$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w    : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En   : IN    STD_LOGIC ;
           y    : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <= "1000" WHEN "100",
             "0100" WHEN "101",
             "0010" WHEN "110",
             "0001" WHEN "111",
             "0000" WHEN OTHERS ;
END Behavior ;
```

Figure 6.30 VHDL code for a 2-to-4 binary decoder.

6.6.3 Conditional Signal Assignment

Similar to the selected signal assignment, a conditional signal assignment allows a signal to be set to one of several values. Figure 6.31 shows a modified version of the 2-to-1 multiplexer entity from Figure 6.27. It uses a conditional signal assignment to specify that f is assigned the value of w_0 when $s = 0$, or else f is assigned the value of w_1. Compiling

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    f <= w0 WHEN s = '0' ELSE w1 ;
END Behavior ;
```

Figure 6.31 Specification of a 2-to-1 multiplexer using a conditional signal assignment.

the code generates the same circuit as the code in Figure 6.27. In this small example the conditional signal assignment has only one WHEN clause. A more complex example, which better illustrates the features of the conditional signal assignment, is given in Example 6.15.

Example 6.15

Figure 6.24 gives the truth table for a 4-to-2 priority encoder. VHDL code that describes this truth table is shown in Figure 6.32. The inputs to the encoder are represented by the four-bit signal named w. The encoder has the outputs y, which is a two-bit signal, and z.

The conditional signal assignment specifies that y is assigned the value 11 when input $w(3) = 1$. If this condition is true, then the other WHEN clauses that follow the ELSE keyword do not affect the value of f. Hence the values of $w(2)$, $w(1)$, and $w(0)$ do not matter, which implements the desired priority scheme. The second WHEN clause states that when $w(2) = 1$, then y is assigned the value 10. This can occur only if $w(3) = 0$. Each successive WHEN clause can affect y only if none of the conditions associated with the preceding WHEN clauses are true. Figure 6.32 includes a second conditional signal assignment for the output z. It states that when all four inputs are 0, z is assigned the value 0; else z is assigned the value 1.

The priority level associated with each WHEN clause in the conditional signal assignment is a key difference from the selected signal assignment, which has no such priority scheme. It is possible to describe the priority encoder using a selected signal assignment, but the code is more awkward. One possibility is shown by the architecture in Figure 6.33. The first WHEN clause sets y to 00 when w_0 is the only input that is 1. The next two clauses state that y should be 01 when $w_3 = w_2 = 0$ and $w_1 = 1$. The next four clauses specify that y should be 10 if $w_3 = 0$ and $w_2 = 1$. Finally, the last WHEN clause states that y should be 1 for all other input valuations, which includes all valuations for which w_3 is 1. Note that

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    y <= "11" WHEN w(3) = '1' ELSE
         "10" WHEN w(2) = '1' ELSE
         "01" WHEN w(1) = '1' ELSE
         "00" ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Figure 6.32 VHDL code for a priority encoder.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    WITH w SELECT
        y <= "00" WHEN "0001",
             "01" WHEN "0010",
             "01" WHEN "0011",
             "10" WHEN "0100",
             "10" WHEN "0101",
             "10" WHEN "0110",
             "10" WHEN "0111",
             "11" WHEN OTHERS ;
    WITH w SELECT
        z <= '0' WHEN "0000",
             '1' WHEN OTHERS ;
END Behavior ;
```

Figure 6.33 Less efficient code for a priority encoder.

the OTHERS clause includes the input valuation 0000. This pattern results in $z = 0$, and the value of y does not matter in this case.

Example 6.16 We derived the circuit for a comparator in Figure 6.26. Figure 6.34 shows how this circuit can be described with VHDL code. Each of the three conditional signal assignments determines the value of one of the comparator outputs. The package named *std_logic_unsigned* is included in the code because it specifies that STD_LOGIC_VECTOR signals, namely, A and B, can be used as unsigned binary numbers with VHDL relational operators. The relational operators provide a convenient way of specifying the desired functionality.

The circuit generated from the code in Figure 6.34 is similar, but not identical, to the circuit in Figure 6.26. The VHDL compiler instantiates a predefined module to implement each of the comparison operations. In Quartus II the modules that are instantiated are from the LPM library, which was introduced in section 5.5.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY compare IS
      PORT ( A, B                     : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                AeqB, AgtB, AltB  : OUT  STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
      AeqB <= '1' WHEN A = B ELSE '0' ;
      AgtB <= '1' WHEN A > B ELSE '0' ;
      AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Figure 6.34 VHDL code for a four-bit comparator.

Instead of using the *std_logic_unsigned* library, another way to specify that the generated circuit should use unsigned numbers is to include the library named *std_logic_arith.* In this case the signals *A* and *B* should be defined with the type UNSIGNED, rather than STD_LOGIC_VECTOR. If we want the circuit to work with signed numbers, signals *A* and *B* should be defined with the type SIGNED. This code is given in Figure 6.35.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY compare IS
      PORT ( A, B                     : IN     SIGNED(3 DOWNTO 0) ;
                AeqB, AgtB, AltB  : OUT  STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
      AeqB <= '1' WHEN A = B ELSE '0' ;
      AgtB <= '1' WHEN A > B ELSE '0' ;
      AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Figure 6.35 The code from Figure 6.34 for signed numbers.

6.6.4 GENERATE STATEMENTS

Figure 6.29 gives VHDL code for a 16-to-1 multiplexer using five instances of a 4-to-1 multiplexer subcircuit. The regular structure of the code suggests that it could be written in a more compact form using a loop. VHDL provides a feature called the FOR GENERATE statement for describing regularly structured hierarchical code.

Figure 6.36 shows the code from Figure 6.29 rewritten using a FOR GENERATE statement. The generate statement must have a label, so we have used the label *G1* in the code. The loop instantiates four copies of the *mux4to1* component, using the loop index *i* in the range from 0 to 3. The variable *i* is not explicitly declared in the code; it is automatically defined as a local variable whose scope is limited to the FOR GENERATE statement. The first loop iteration corresponds to the instantiation statement labeled *Mux1* in Figure 6.29. The * operator represents multiplication; hence for the first loop iteration the VHDL compiler translates the signal names $w(4 * i)$, $w(4 * i + 1)$, $w(4 * i + 2)$, and $w(4 * i + 3)$ into signal names $w(0)$, $w(1)$, $w(2)$, and $w(3)$. The loop iterations for $i = 1$, $i = 2$, and $i = 3$ correspond to the statements labeled *Mux2*, *Mux3*, and *Mux4* in Figure 6.29. The statement labeled *Mux5* in Figure 6.29 does not fit within the loop, so it is included as a separate statement in Figure 6.36. The circuit generated from the code in Figure 6.36 is identical to the circuit produced by using the code in Figure 6.29.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.mux4to1_package.all ;

ENTITY mux16to1 IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(0 TO 15) ;
           s  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f  : OUT  STD_LOGIC ) ;
END mux16to1 ;

ARCHITECTURE Structure OF mux16to1 IS
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1:  FOR i IN 0 TO 3 GENERATE
         Muxes: mux4to1 PORT MAP (
              w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 DOWNTO 0), m(i) ) ;
    END GENERATE ;
    Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```

Figure 6.36 Code for a 16-to-1 multiplexer using a generate statement.

Example 6.17

In addition to the FOR GENERATE statement, VHDL provides another type of generate statement called IF GENERATE. Figure 6.37 illustrates the use of both types of generate statements. The code shown is a hierarchical description of the 4-to-16 decoder given in Figure 6.18, using five instances of the *dec2to4* component defined in Figure 6.30. The decoder inputs are the four-bit signal w, the enable is En, and the outputs are the 16-bit signal y.

Following the component declaration for the *dec2to4* subcircuit, the architecture defines the signal m, which represents the outputs of the 2-to-4 decoder on the left of Figure 6.18. The five copies of the *dec2to4* component are instantiated by the FOR GENERATE statement. In each iteration of the loop, the statement labeled *Dec_ri* instantiates a *dec2to4* component that corresponds to one of the 2-to-4 decoders on the right side of Figure 6.18. The first loop iteration generates the *dec2to4* component with data inputs w_1 and w_0, enable input m_0, and outputs y_0, y_1, y_2, y_3. The other loop iterations also use data inputs w_1w_0, but use different bits of m and y.

The IF GENERATE statement, labeled *G2*, instantiates a *dec2to4* component in the last loop iteration, for which the condition $i = 3$ is true. This component represents the 2-to-4 decoder on the left of Figure 6.18. It has the two-bit data inputs w_3 and w_2, the enable En, and

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec4to16 IS
    PORT ( w    : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           En   : IN    STD_LOGIC ;
           y    : OUT   STD_LOGIC_VECTOR(0 TO 15) ) ;
END dec4to16 ;

ARCHITECTURE Structure OF dec4to16 IS
    COMPONENT dec2to4
        PORT ( w    : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En   : IN    STD_LOGIC ;
               y    : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
    END COMPONENT ;
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Dec_ri: dec2to4 PORT MAP ( w(1 DOWNTO 0), m(i), y(4*i TO 4*i+3) );
        G2: IF i=3 GENERATE
            Dec_left: dec2to4 PORT MAP ( w(i DOWNTO i-1), En, m ) ;
        END GENERATE ;
    END GENERATE ;
END Structure ;
```

Figure 6.37 Hierarchical code for a 4-to-16 binary decoder.

the outputs m_0, m_1, m_2, and m_3. Note that instead of using the IF GENERATE statement, we could have instantiated this component outside the FOR GENERATE statement. We have written the code as shown simply to give an example of the IF GENERATE statement.

The generate statements in Figures 6.36 and 6.37 are used to instantiate components. Another use of generate statements is to generate a set of logic equations. An example of this use will be given in Figure 7.73.

6.6.5 Concurrent and Sequential Assignment Statements

We have introduced several types of assignment statements: simple assignment statements, which involve logic or arithmetic expressions, selected assignment statements, and conditional assignment statements. All of these statements share the property that the order in which they appear in VHDL code does not affect the meaning of the code. Because of this property, these statements are called the *concurrent assignment statements*.

VHDL also provides a second category of statements, called *sequential assignment statements*, for which the ordering of the statements may affect the meaning of the code. We will discuss two types of sequential assignment statements, called if-then-else statements and case statements. VHDL requires that the sequential assignment statements be placed inside another type of statement, called a process statement.

6.6.6 Process Statement

Figures 6.27 and 6.31 show two ways of describing a 2-to-1 multiplexer, using the selected and conditional signal assignments. The same circuit can also be described using an if-then-else statement, but this statement must be placed inside a process statement. Figure 6.38 shows such code. The process statement, or simply *process*, begins with the PROCESS keyword, followed by a parenthesized list of signals, called the *sensitivity list*. For a combinational circuit like the multiplexer, the sensitivity list includes all input signals that are used inside the process. The process statement is translated by the VHDL compiler into logic equations. In the figure the process consists of the single if-then-else statement that describes the multiplexer function. Thus the sensitivity list comprises the data inputs, w_0 and w_1, and the select input s.

In general, there can be a number of statements inside a process. These statements are considered as follows. Using VHDL jargon, we say that when there is a change in the value of any signal in the process's sensitivity list, then the process becomes *active*. Once active, the statements inside the process are evaluated in sequential order. Any assignments made to signals inside the process are not visible outside the process until all of the statements in the process have been evaluated. If there are multiple assignments to the same signal, only the last one has any visible effect. This is illustrated in Example 6.18.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        IF s = '0' THEN
            f <= w0 ;
        ELSE
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.38 A 2-to-1 multiplexer specified using the if-then-else statement.

Example 6.18

The code in Figure 6.39 is equivalent to the code in Figure 6.38. The first statement in the process assigns the value of w_0 to f. This provides a *default* value for f but the assignment does not actually take place until the end of the process. In VHDL jargon we say that the assignment is *scheduled* to occur after all of the statements in the process have been evaluated. If another assignment to f takes place while the process is active, the default assignment will be overridden. The second statement in the process assigns the value of w_1 to f if the value of s is equal to 1. If this condition is true, then the default assignment is overridden. Thus if $s = 0$, then $f = w_0$, and if $s = 1$, then $f = w_1$, which defines the 2-to-1 multiplexer. Compiling this code results in the same circuit as for Figures 6.27, 6.31, and 6.38, namely, $f = \bar{s}w_0 + sw_1$.

The process statement in Figure 6.39 illustrates that the ordering of the statements in a process can affect the meaning of the code. Consider reversing the order of the two statements so that the if-then-else statement is evaluated first. If $s = 1$, f is assigned the value of w_1. This assignment is scheduled and does not take place until the end of the process. However, the statement $f <= w_0$ is evaluated last. It overrides the first assignment, and f is assigned the value of w_0 regardless of the value of s. Hence instead of describing a multiplexer, when the statements inside the process are reversed, the code represents the trivial circuit $f = w_0$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
              f       : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        f <= w0 ;
        IF s = '1' THEN
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.39 Alternative code for the 2-to-1 multiplexer using an if-then-else statement.

Example 6.19 Figure 6.40 gives an example that contains both a concurrent assignment statement and a process statement. It describes a priority encoder and is equivalent to the code in Figure 6.32. The process describes the desired priority scheme using an if-then-else statement. It specifies that if the input w_3 is 1, then the output is set to $y = 11$. This assignment does not depend on the values of inputs w_2, w_1, or w_0; hence their values do not matter. The other clauses in the if-then-else statement are evaluated only if $w_3 = 0$. The first ELSIF clause states that if w_2 is 1, then $y = 10$. If $w_2 = 0$, then the next ELSIF clause results in $y = 01$ if $w_1 = 1$. If $w_3 = w_2 = w_1 = 0$, then the ELSE clause results in $y = 00$. This assignment is done whether or not w_0 is 1; Figure 6.24 indicates that y can be set to any pattern when $w = 0000$ because z will be set to 0 in this case.

The priority encoder's output z must be set to 1 whenever at least one of the data inputs is 1. This output is defined by the conditional assignment statement at the end of Figure 6.40. The VHDL syntax does not allow a conditional assignment statement (or a selected assignment statement) to appear inside a process. An alternative would be to specify the value of z by using an if-then-else statement inside the process. The reason that we have written the code as given in the figure is to illustrate that concurrent assignment statements can be used in conjunction with process statements. The process statement serves the purpose of separating the sequential statements from the concurrent statements. Note that the ordering of the process statement and the conditional assignment statement does not matter. VHDL stipulates that while the statements inside a process are sequential statements, the process statement itself is a concurrent statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        IF w(3) = '1' THEN
            y <= "11" ;
        ELSIF w(2) = '1' THEN
            y <= "10" ;
        ELSIF w(1) = '1' THEN
            y <= "01" ;
        ELSE
            y <= "00" ;
        END IF ;
    END PROCESS ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Figure 6.40 A priority encoder specified using the if-then-else statement.

Example 6.20

Figure 6.41 shows an alternative style of code for the priority encoder, using if-then-else statements. The first statement in the process provides the default value of 00 for y_1y_0. The second statement overrides this if w_1 is 1, and sets y_1y_0 to 01. Similarly, the third and fourth statements override the previous ones if w_2 or w_3 are 1, and set y_1y_0 to 10 and 11, respectively. These four statements are equivalent to the single if-then-else statement in Figure 6.40 that describes the priority scheme. The value of z is specified using a default assignment statement, followed by an if-then-else statement that overrides the default if $w = 0000$. Although the examples in Figures 6.40 and 6.41 are equivalent, the meaning of the code in Figure 6.40 is probably easier to understand.

Example 6.21

Figure 6.34 specifies a four-bit comparator that produces the three outputs $AeqB$, $AgtB$, and $AltB$. Figure 6.42 shows how such specification can be written using if-then-else statements. For simplicity, one-bit numbers are used for the inputs A and B, and only the code for the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y   : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z   : OUT   STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        y <= "00" ;
        IF w(1) = '1' THEN y <= "01" ; END IF ;
        IF w(2) = '1' THEN y <= "10" ; END IF ;
        IF w(3) = '1' THEN y <= "11" ; END IF ;

        z <= '1' ;
        IF w = "0000" THEN z <= '0' ; END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.41 Alternative code for the priority encoder.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY compare1 IS
    PORT ( A, B   : IN    STD_LOGIC ;
           AeqB   : OUT   STD_LOGIC ) ;
END compare1 ;

ARCHITECTURE Behavior OF compare1 IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        AeqB <= '0' ;
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.42 Code for a one-bit equality comparator.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
    PORT ( A, B   : IN    STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.43 An example of code that results in implied memory.

AeqB output is shown. The process assigns the default value of 0 to *AeqB* and then the if-then-else statement changes *AeqB* to 1 if *A* and *B* are equal. It is instructive to consider the effect on the semantics of the code if the default assignment statement is removed, as illustrated in Figure 6.43.

With only the if-then-else statement, the code does not specify what value *AeqB* should have if the condition $A = B$ is not true. The VHDL semantics stipulate that in cases where the code does not specify the value of a signal, the signal should retain its current value. For the code in Figure 6.43, once *A* and *B* are equal, resulting in $AeqB = 1$, then *AeqB* will remain set to 1 indefinitely, even if *A* and *B* are no longer equal. In the VHDL jargon, the *AeqB* output is said to have *implied memory* because the circuit synthesized from the code will "remember," or store the value $AeqB = 1$. Figure 6.44 shows the circuit synthesized from the code. The XOR gate produces a 1 when *A* and *B* are equal, and the OR gate ensures that *AeqB* remains set to 1 indefinitely.

The implied memory that results from the code in Figure 6.43 is not useful, because it generates a comparator circuit that does not function correctly. However, we will show

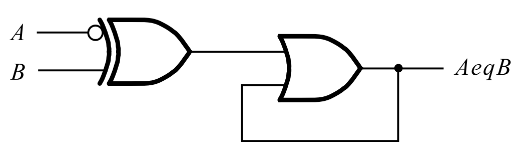


Figure 6.44 The circuit generated from the code in Figure 6.43.

in Chapter 7 that the semantics of implied memory are useful for other types of circuits, which have the capability to store logic signal values in memory elements.

6.6.7 CASE STATEMENT

A case statement is similar to a selected signal assignment in that the case statement has a selection signal and includes WHEN clauses for various valuations of this selection signal. Figure 6.45 shows how the case statement can be used as yet another way of describing the 2-to-1 multiplexer circuit. The case statement begins with the CASE keyword, which specifies that s is to be used as the selection signal. The first WHEN clause specifies, following the => symbol, the statements that should be evaluated when $s = 0$. In this example the only statement evaluated when $s = 0$ is $f <= w_0$. The case statement must include a WHEN clause for all possible valuations of the selection signal. Hence the second WHEN clause, which contains $f <= w_1$, uses the OTHERS keyword.

Example 6.22 Figure 6.30 gives the code for a 2-to-4 decoder. A different way of describing this circuit, using sequential assignment statements, is shown in Figure 6.46. The process first uses an if-then-else statement to check the value of the decoder enable signal *En*. If $En = 1$, the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        CASE s IS
            WHEN '0' =>
                f <= w0 ;
            WHEN OTHERS =>
                f <= w1 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.45 A case statement that represents a 2-to-1 multiplexer.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w    : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En   : IN    STD_LOGIC ;
           y    : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
    PROCESS ( w, En )
    BEGIN
        IF En = '1' THEN
            CASE w IS
                WHEN "00" =>
                    y <= "1000" ;
                WHEN "01" =>
                    y <= "0100" ;
                WHEN "10" =>
                    y <= "0010" ;
                WHEN OTHERS =>
                    y <= "0001" ;
            END CASE ;
        ELSE
            y <= "0000" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.46 A process statement that describes a 2-to-4 binary decoder.

case statement sets the output y to the appropriate value based on the input w. The case statement represents the first four rows of the truth table in Figure 6.16a. If $En = 0$, the ELSE clause sets y to 0000, as specified in the bottom row of the truth table.

Example 6.23

Another example of a case statement is given in Figure 6.47. The entity is named *seg7*, and it represents the BCD-to-7-segment decoder in Figure 6.25. The BCD input is represented by the four-bit signal named *bcd*, and the seven outputs are the seven-bit signal named *leds*. The case statement is formatted so that it resembles the truth table in Figure 6.25c. Note that there is a comment to the right of the case statement, which labels the seven outputs

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY seg7 IS
    PORT ( bcd   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           leds  : OUT   STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;

ARCHITECTURE Behavior OF seg7 IS
BEGIN
    PROCESS ( bcd )
    BEGIN
        CASE bcd IS                    --    abcdefg
            WHEN "0000"   => leds <= "1111110" ;
            WHEN "0001"   => leds <= "0110000" ;
            WHEN "0010"   => leds <= "1101101" ;
            WHEN "0011"   => leds <= "1111001" ;
            WHEN "0100"   => leds <= "0110011" ;
            WHEN "0101"   => leds <= "1011011" ;
            WHEN "0110"   => leds <= "1011111" ;
            WHEN "0111"   => leds <= "1110000" ;
            WHEN "1000"   => leds <= "1111111" ;
            WHEN "1001"   => leds <= "1110011" ;
            WHEN OTHERS => leds <= "-------" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.47 Code that represents a BCD-to-7-segment decoder.

with the letters from *a* to *g*. These labels indicate to the reader the correlation between the seven-bit *leds* signal in the VHDL code and the seven segments in Figure 6.25*b*. The final WHEN clause in the case statement sets all seven bits of *leds* to −. Recall that − is used in VHDL to denote a don't-care condition. This clause represents the don't-care conditions discussed for Figure 6.25, which are the cases where the *bcd* input does not represent a valid BCD digit.

Example 6.24 An arithmetic logic unit (ALU) is a logic circuit that performs various Boolean and arithmetic operations on *n*-bit operands. In section 3.5 we discussed a family of standard chips called the 7400-series chips. We said that some of these chips contain basic logic gates, and others provide commonly used logic circuits. One example of an ALU is the standard chip called the 74381. Table 6.1 specifies the functionality of this chip. It has 2 four-bit data inputs, named *A* and *B*; a three-bit select input *s*; and a four-bit output *F*. As the table shows,

Table 6.1 The functionality of the 74381 ALU.

Operation	Inputs $s_2\ s_1\ s_0$	Outputs F
Clear	0 0 0	0 0 0 0
B–A	0 0 1	$B - A$
A–B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

F is defined by various arithmetic or Boolean operations on the inputs A and B. In this table + means arithmetic addition, and − means arithmetic subtraction. To avoid confusion, the table uses the words XOR, OR, and AND for the Boolean operations. Each Boolean operation is done in a bit-wise fashion. For example, $F = A$ AND B produces the four-bit result $f_0 = a_0b_0, f_1 = a_1b_1, f_2 = a_2b_2$, and $f_3 = a_3b_3$.

Figure 6.48 shows how the functionality of the 74381 ALU can be described using VHDL code. The std_logic_unsigned package, introduced in section 5.5.4, is included so that the STD_LOGIC_VECTOR signals A and B can be used in unsigned arithmetic operations. The case statement shown corresponds directly to Table 6.1. To check the functionality of the code, we synthesized a circuit for implementation in a CPLD. An example of a timing simulation is illustrated in Figure 6.49. For each valuation of s, the circuit generates the appropriate Boolean or arithmetic operation.

6.6.8 VHDL OPERATORS

In this section we discuss the VHDL operators that are useful for synthesizing logic circuits. Table 6.2 lists these operators in groups that reflect the type of operation performed.

To illustrate the results produced by the various operators, we will use three-bit vectors A(2 DOWNTO 0), B(2 DOWNTO 0), and C(2 DOWNTO 0).

Logical Operators

The logical operators can be used with bit and boolean types of operands. The operands can be either single-bit scalars or multibit vectors. For example, the statement

```
C <= NOT A;
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY alu IS
    PORT ( s      : IN   STD_LOGIC_VECTOR(2 DOWNTO 0) ;
           A, B   : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           F      : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END alu ;

ARCHITECTURE Behavior OF alu IS
BEGIN
    PROCESS ( s, A, B )
    BEGIN
        CASE s IS
            WHEN "000" =>
                F <= "0000" ;
            WHEN "001" =>
                F <= B - A ;
            WHEN "010" =>
                F <= A - B ;
            WHEN "011" =>
                F <= A + B ;
            WHEN "100" =>
                F <= A XOR B ;
            WHEN "101" =>
                F <= A OR B ;
            WHEN "110" =>
                F <= A AND B ;
            WHEN OTHERS =>
                F <= "1111" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.48 Code that represents the functionality of the 74381 ALU chip.

produces the result $c_2 = \overline{a}_2$, $c_1 = \overline{a}_1$, and $c_0 = \overline{a}_0$, where a_i and c_i are the bits of the vectors A and C.

The statement

C <= A AND B;

generates $c_2 = a_2 \cdot b_2$, $c_1 = a_1 \cdot b_1$, and $c_0 = a_0 \cdot b_0$. The other operators lead to similar evaluations.

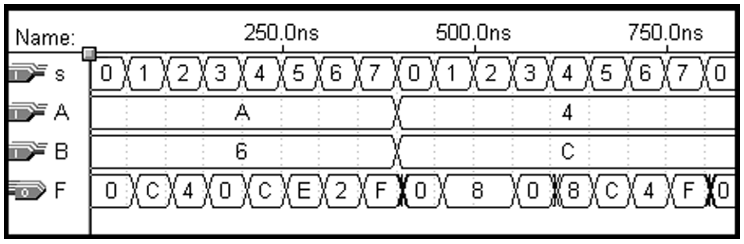


Figure 6.49 Timing simulation for the code in Figure 6.48.

Table 6.2 VHDL operators (used for synthesis).

Operator category	Operator symbol	Operation performed
Logical	AND	AND
	OR	OR
	NAND	Not AND
	NOR	Not OR
	XOR	XOR
	XNOR	Not XOR
	NOT	NOT
Relational	=	Equality
	/=	Inequality
	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
Arithmetic	+	Addition
	−	Subtraction
	∗	Multiplication
	/	Division
Concatenation	&	Concatenation
Shift and Rotate	SLL	Shift left logical
	SRL	Shift right logical
	SLA	Shift left arithmetic
	SRA	Shift right arithmetic
	ROL	Rotate left
	ROR	Rotate right

Relational Operators

The relational operators are used to compare expressions. The result of the comparison is TRUE or FALSE. The expressions that are compared must be of the same type. For example, if $A = 011$ and $B = 010$ then A > B evaluates to TRUE, and B /= ”010” evaluates to FALSE.

Arithmetic Operators

We have already encountered the arithmetic operators in Chapter 5. They perform standard arithmetic operations. Thus

$$\text{C} <= \text{A} + \text{B};$$

puts the three-bit sum of A plus B into C, while

$$\text{C} <= \text{A} - \text{B};$$

puts the difference of A and B into C. The operation

$$\text{C} <= -\text{A};$$

places the 2's complement of A into C.

The addition, subtraction, and multiplication operations are supported by most CAD synthesis tools. However, the division operation is often not supported. When the VHDL compiler encounters an arithmetic operator, it usually synthesizes it by using an appropriate module from a library.

Concatenate Operator

This operator concatenates two or more vectors to create a larger vector. For example,

$$\text{D} <= \text{A} \; \& \; \text{B};$$

defines the six-bit vector $D = a_2a_1a_0b_2b_1b_0$. Similarly, the concatenation

$$\text{E} <= \text{"111"} \; \& \; \text{A} \; \& \; \text{"00"};$$

produces the eight-bit vector $E = 111a_2a_1a_000$.

Shift and Rotate Operators

A vector operand can be shifted to the right or left by a number of bits specified as a constant. When bits are shifted, the vacant bit positions are filled with 0s. For example,

$$\text{B} <= \text{A SLL 1};$$

results in $b_2 = a_1$, $b_1 = a_0$, and $b_0 = 0$. Similarly,

$$\text{B} <= \text{A SRL 2};$$

yields $b_2 = b_1 = 0$ and $b_0 = a_2$.

The arithmetic shift left, SLA, has the same effect as SLL. But, the arithmetic shift right, SRA, performs the sign extension by replicating the sign bit into the positions left vacant after shifting. Hence

$$\text{B} <= \text{A SRA 1};$$

gives $b_2 = a_2$, $b_1 = a_2$, and $b_0 = a_1$.

An operand can also be rotated, in which case the bits shifted out from one end are placed into the vacated positions at the other end. For example,

$$\text{B} <= \text{A ROR 2};$$

produces $b_2 = a_1$, $b_1 = a_0$, and $b_0 = a_2$.

May 19, 2004 11:40 vra60857_ch06 Sheet number 49 Page number 363 black

Operator Precedence

Operators in different categories have different precedence. Operators in the same category have the same precedence, and are evaluated from left to right in a given expression. It is a good practice to use parentheses to indicate the desired order of operations in the expression. To illustrate this point, consider the statement

$$S <= A + B + C + D;$$

which defines the addition of four vector operands. The VHDL compiler will synthesize a circuit as if the expression was written in the form ((A + B) + C) + D, which gives a cascade of three adders so that the final sum will be available after a propagation delay through three adders. By writing the statement as

$$S <= (A + B) + (C + D);$$

the synthesized circuit will still have three adders, but since the sums $A + B$ and $C + D$ are generated in parallel, the final sum will be available after a propagation delay through only two adders.

Table 6.2 groups the operators informally according to their functionality. It shows only those operators that are used to synthesize logic circuits. The VHDL Standard specifies additional operators, which are useful for simulation and documentation purposes. All operators are grouped into different classes, with a defined precedence ordering between classes. We discuss this issue in Appendix A, section A.3.

6.7 Concluding Remarks

This chapter has introduced a number of circuit building blocks. Examples using these blocks to construct larger circuits will be presented in Chapters 7 and 10. To describe the building block circuits efficiently, several VHDL constructs have been introduced. In many cases a given circuit can be described in various ways, using different constructs. A circuit that can be described using a selected signal assignment can also be described using a case statement. Circuits that fit well with conditional signal assignments are also well-suited to if-then-else statements. In general, there are no clear rules that dictate when one type of assignment statement should be preferred over another. With experience the user develops a sense for which types of statements work well in a particular design situation. Personal preference also influences how the code is written.

VHDL is not a programming language, and VHDL code should not be written as if it were a computer program. The concurrent and sequential assignment statements discussed in this chapter can be used to create large, complex circuits. A good way to design such circuits is to construct them using well-defined modules, in the manner that we illustrated for the multiplexers, decoders, encoders, and so on. Additional examples using the VHDL statements introduced in this chapter are given in Chapters 7 and 8. In Chapter 10 we provide a number of examples of using VHDL code to describe larger digital systems. For more information on VHDL, the reader can consult more specialized books [5–10].

In the next chapter we introduce logic circuits that include the ability to store logic signal values in memory elements.

6.8 EXAMPLES OF SOLVED PROBLEMS

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example 6.25 **Problem:** Implement the function $f(w_1, w_2, w_3) = \sum m(0, 1, 3, 4, 6, 7)$ by using a 3-to-8 binary decoder and an OR gate.

Solution: The decoder generates a separate output for each minterm of the required function. These outputs are then combined in the OR gate, giving the circuit in Figure 6.50.

Example 6.26 **Problem:** Derive a circuit that implements an 8-to-3 binary encoder.

Solution: The truth table for the encoder is shown in Figure 6.51. Only those rows for which a single input variable is equal to 1 are shown; the other rows can be treated as don't care cases. From the truth table it is seen that the desired circuit is defined by the equations

$$\begin{aligned} y_2 &= w_4 + w_5 + w_6 + w_7 \\ y_1 &= w_2 + w_3 + w_6 + w_7 \\ y_0 &= w_1 + w_3 + w_5 + w_7 \end{aligned}$$

Example 6.27 **Problem:** Implement the function

$$f(w_1, w_2, w_3, w_4) = \overline{w}_1\overline{w}_2\overline{w}_4\overline{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5$$

by using a 4-to-1 multiplexer and as few other gates as possible. Assume that only the uncomplemented inputs w_1, w_2, w_3, and w_4 are available.

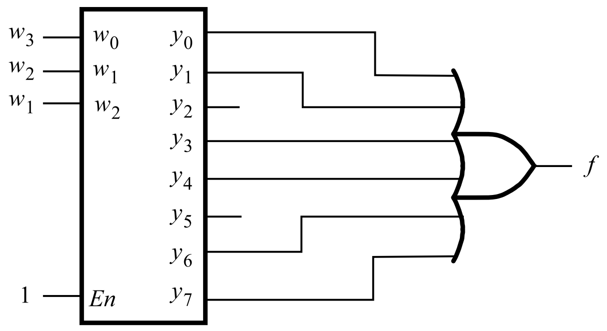


Figure 6.50 Circuit for Example 6.25.

w_7	w_6	w_5	w_4	w_3	w_2	w_1	w_0	y_2	y_1	y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Figure 6.51 Truth table for an 8-to-3 binary encoder.

Solution: Since variables w_1 and w_4 appear in more product terms in the expression for f than the other three variables, let us perform Shannon's expansion with respect to these two variables. The expansion gives

$$\begin{aligned} f &= \overline{w}_1\overline{w}_4 f_{\overline{w}_1\overline{w}_4} + \overline{w}_1 w_4 f_{\overline{w}_1 w_4} + w_1\overline{w}_4 f_{w_1\overline{w}_4} + w_1 w_4 f_{w_1 w_4} \\ &= \overline{w}_1\overline{w}_4(\overline{w}_2\overline{w}_5) + \overline{w}_1 w_4(w_3 w_5) + w_1\overline{w}_4(w_2 + w_3) + w_1 w_2(1) \end{aligned}$$

We can use a NOR gate to implement $\overline{w}_2\overline{w}_5 = \overline{w_2 + w_5}$. We also need an AND gate and an OR gate. The complete circuit is presented in Figure 6.52.

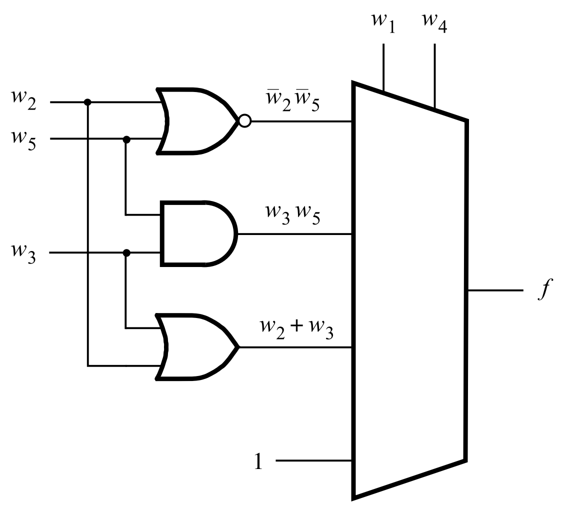


Figure 6.52 Circuit for Example 6.27.

b_2	b_1	b_0	g_2	g_1	g_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

Figure 6.53 Binary to Gray code coversion.

Example 6.28 **Problem:** In Chapter 4 we pointed out that the rows and columns of a Karnaugh map are labeled using Gray code. This is a code in which consecutive valuations differ in one variable only. Figure 6.53 depicts the conversion between three-bit binary and Gray codes. Design a circuit that can convert a binary code into a Gray according the figure.

Solution: From the figure it follows that

$$\begin{aligned} g_2 &= b_2 \\ g_1 &= b_1\overline{b}_2 + \overline{b}_1 b_2 \\ &= b_1 \oplus b_2 \\ g_0 &= b_0\overline{b}_1 + \overline{b}_0 b_1 \\ &= b_0 \oplus b_1 \end{aligned}$$

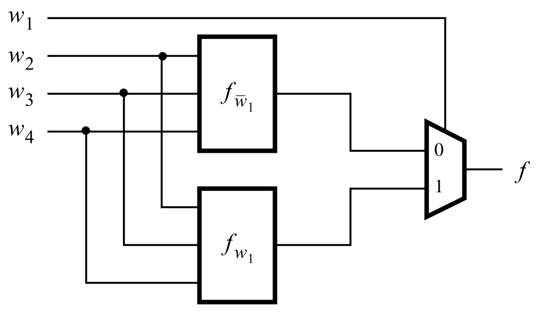
Example 6.29 **Problem:** In section 6.1.2 we showed that any logic function can be decomposed using Shannon's expansion theorem. For a four-variable function, $f(w_1, \ldots, w_4)$, the expansion with respect to w_1 is

$$f(w_1, \ldots, w_4) = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

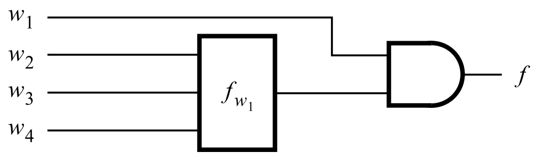
A circuit that implements this expression is given in Figure 6.54*a*.
(a) If the decomposition yields $f_{\overline{w}_1} = 0$, then the multiplexer in the figure can be replaced by a single logic gate. Show this circuit.
(b) Repeat part (*a*) for the case where $f_{w_1} = 1$.

Solution: The desired circuits are shown in parts (*b*) and (*c*) of Figure 6.54.

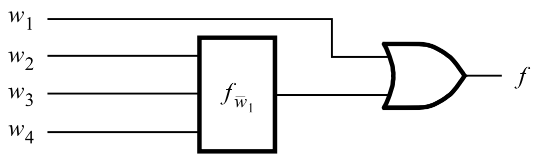
Example 6.30 **Problem:** In several commercial FPGAs the logic blocks are 4-LUTs. What is the minimum number of 4-LUTs needed to construct a 4-to-1 multiplexer with select inputs s_1 and s_0 and data inputs w_3, w_2, w_1, and w_0?



(a) Shannon's expansion of the function *f*.



(b) Solution for part *a*



(c) Solution for part *b*

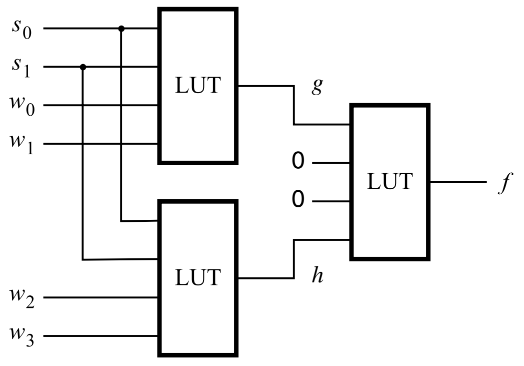
Figure 6.54 Circuits for Example 6.29.

Solution: A straightforward attempt is to use directly the expression that defines the 4-to-1 multiplexer

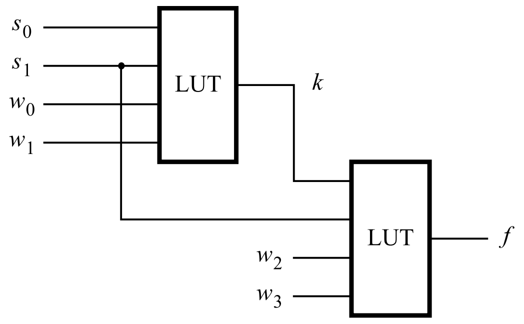
$$f = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1\overline{s}_0w_2 + s_1s_0w_3$$

Let $g = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1$ and $h = s_1\overline{s}_0w_2 + s_1s_0w_3$, so that $f = g + h$. This decomposition leads to the circuit in Figure 6.55*a*, which requires three LUTs.

When designing logic circuits, one can sometimes come up with a clever idea which leads to a superior implementation. Figure 6.55*b* shows how it is possible to implement the multiplexer with just two LUTs, based on the following observation. The truth table in Figure 6.2*b* indicates that when $s_1 = 0$ the output must be either w_0 or w_1, as determined by the value of s_0. This can be generated by the first LUT. The second LUT must make the choice between w_2 and w_3 when $s_1 = 1$. But, the choice can be made only by knowing the value of s_0. Since it is impossible to have five inputs in the LUT, more information has to



(a) Using three LUTs



(b) Using two LUTs

Figure 6.55 Circuits for Example 6.30.

be passed from the first to the second LUT. Observe that when $s_1 = 1$ the output f will be equal to either w_2 or w_3, in which case it is not necessary to know the values of w_0 and w_1. Hence, in this case we can pass on the value of s_0 through the first LUT, rather than w_0 or w_1. This can be done by making the function of this LUT

$$k = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1 + s_1s_0$$

Then, the second LUT performs the function

$$f = \bar{s}_1k + s_1\bar{k}w_3 + s_1kw_4$$

Example 6.31 **Problem:** In digital systems it is often necessary to have circuits that can shift the bits of a vector by one or more bit positions to the left or right. Design a circuit that can shift a

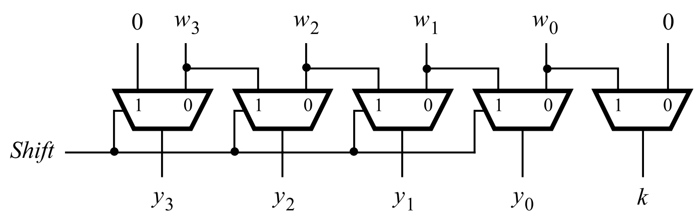


Figure 6.56 A shifter circuit.

four-bit vector $W = w_3w_2w_1w_0$ one bit position to the right when a control signal *Shift* is equal to 1. Let the outputs of the circuit be a four-bit vector $Y = y_3y_2y_1y_0$ and a signal k, such that if $Shift = 1$ then $y_3 = 0$, $y_2 = w_3$, $y_1 = w_2$, $y_0 = w_1$, and $k = w_0$. If $Shift = 0$ then $Y = W$ and $k = 0$.

Solution: The required circuit can be implemented with five 2-to-1 multiplexers as shown in Figure 6.56. The *Shift* signal is used as the select input to each multiplexer.

Example 6.32

Problem: The shifter circuit in Example 6.31 shifts the bits of an input vector by one bit position to the right. It fills the vacated bit on the left side with 0. A more versatile shifter circuit may be able to shift by more bit positions at a time. If the bits that are shifted out are placed into the vacated positions on the left, then the circuit effectively rotates the bits of the input vector by a specified number of bit positions. Such a circuit is often called a *barrel shifter*. Design a four-bit barrel shifter that rotates the bits by 0, 1, 2, or 3 bit positions as determined by the valuation of two control signals s_1 and s_0.

Solution: The required action is given in Figure 6.57*a*. The barrel shifter can be implemented with four 4-to-1 multiplexers as shown in Figure 6.57*b*. The control signals s_1 and s_0 are used as the select inputs to the multiplexers.

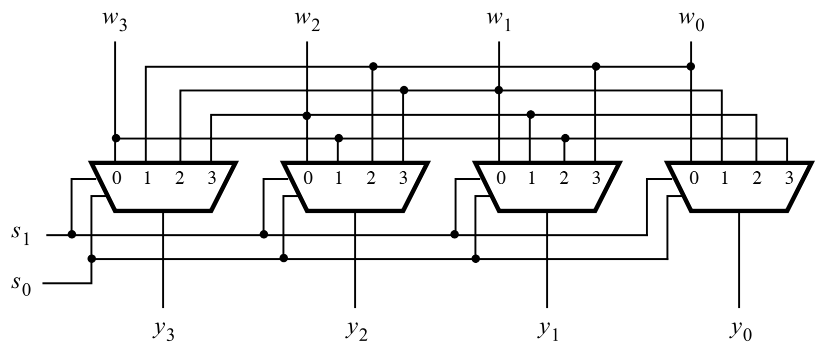
Example 6.33

Problem: Write VHDL code that represents the circuit in Figure 6.19. Use the *dec2to4* entity in Figure 6.30 as a subcircuit in your code.

Solution: The code is shown in Figure 6.58. Note that the *dec2to4* entity can be included in the same file as we have done in the figure, but it can also be in a separate file in the project directory.

s_1	s_0	y_3	y_2	y_1	y_0
0	0	w_3	w_2	w_1	w_0
0	1	w_0	w_3	w_2	w_1
1	0	w_1	w_0	w_3	w_2
1	1	w_2	w_1	w_0	w_3

(a) Truth table



(b) Circuit

Figure 6.57 A barrel shifter circuit.

Example 6.34 **Problem:** Write VHDL code that represents the shifter circuit in Figure 6.56.

Solution: There are two possible approaches: structural and behavioral. A structural description is given in Figure 6.59. The IF construct is used to define the desired shifting of individual bits. A typical VHDL compiler will implement this code with 2-to-1 multiplexers as depicted in Figure 6.56.

A behavioral specification is given in Figure 6.60. It makes use of the shift operator SRL. Since the shift and rotate operators are supported in the *ieee.numeric_std.all* library, this library must be included in the code. Note that the vectors *w* and *y* are defined to be of UNSIGNED type.

Example 6.35 **Problem:** Write VHDL code that defines the barrel shifter in Figure 6.57.

Solution: The easiest way to specify the barrel shifter is by using the VHDL rotate operator. The complete code is presented in Figure 6.61.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT ( s    : IN    STD_LOGIC_VECTOR( 1 DOWNTO 0 ) ;
           w    : IN    STD_LOGIC_VECTOR( 3 DOWNTO 0 ) ;
           f    : OUT   STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Structure OF mux4to1 IS
    COMPONENT dec2to4
        PORT ( w    : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En   : IN    STD_LOGIC ;
               y    : OUT   STD_LOGIC_VECTOR(0 TO 3) );
    END COMPONENT;
    SIGNAL High : STD_LOGIC ;
    SIGNAL y : STD_LOGIC_VECTOR( 3 DOWNTO 0 ) ;
BEGIN
    decoder: dec2to4 PORT MAP ( s, '1', y ) ;
    f <= (w(0) AND y(0)) OR (w(1) AND y(1)) OR
         (w(2) AND y(2)) OR w(3) AND y(3) ) ;
END Structure ;


LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w    : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En   : IN    STD_LOGIC ;
           y    : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <=  "1000" WHEN "100",
              "0100" WHEN "101",
              "0010" WHEN "110",
              "0001" WHEN "111",
              "0000" WHEN OTHERS ;
END Behavior ;
```

Figure 6.58 VHDL code for Example 6.38.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shifter IS
    PORT ( w      : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Shift  : IN   STD_LOGIC ;
           y      : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           k      : OUT  STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = '1' THEN
            y(3) <= '0' ;
            y(2 DOWNTO 0) <= w(3 DOWNTO 1) ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= '0' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.59 Structural VHDL code that specifies the shifter circuit in Figure 6.56.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

6.1 Show how the function $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 4, 5, 7)$ can be implemented using a 3-to-8 binary decoder and an OR gate.

6.2 Show how the function $f(w_1, w_2, w_3) = \sum m(1, 2, 3, 5, 6)$ can be implemented using a 3-to-8 binary decoder and an OR gate.

***6.3** Consider the function $f = \overline{w}_1\overline{w}_3 + w_2\overline{w}_3 + \overline{w}_1 w_2$. Use the truth table to derive a circuit for f that uses a 2-to-1 multiplexer.

6.4 Repeat problem 6.3 for the function $f = \overline{w}_2\overline{w}_3 + w_1 w_2$.

***6.5** For the function $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 6)$, use Shannon's expansion to derive an implementation using a 2-to-1 multiplexer and any other necessary gates.

6.6 Repeat problem 6.5 for the function $f(w_1, w_2, w_3) = \sum m(0, 4, 6, 7)$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all ;

ENTITY shifter IS
    PORT ( w     : IN    UNSIGNED(3 DOWNTO 0) ;
           Shift : IN    STD_LOGIC ;
           y     : OUT   UNSIGNED(3 DOWNTO 0) ;
           k     : OUT   STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = "1" THEN
            y <= w SRL 1 ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= "0" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.60 Behavioral VHDL code that specifies the shifter circuit in Figure 6.56.

6.7 Consider the function $f = \overline{w}_2 + \overline{w}_1\overline{w}_3 + w_1w_3$. Show how repeated application of Shannon's expansion can be used to derive the minterms of f.

6.8 Repeat problem 6.7 for $f = w_2 + \overline{w}_1\overline{w}_3$.

6.9 Prove Shannon's expansion theorem presented in section 6.1.2.

***6.10** Section 6.1.2 shows Shannon's expansion in sum-of-products form. Using the principle of duality, derive the equivalent expression in product-of-sums form.

6.11 Consider the function $f = \overline{w}_1\overline{w}_2 + \overline{w}_2\overline{w}_3 + w_1w_2w_3$. Give a circuit that implements f using the minimal number of two-input LUTs. Show the truth table implemented inside each LUT.

***6.12** For the function in problem 6.11, the cost of the minimal sum-of-products expression is 14, which includes four gates and 10 inputs to the gates. Use Shannon's expansion to derive a multilevel circuit that has a lower cost and give the cost of your circuit.

6.13 Consider the function $f(w_1, w_2, w_3, w_4) = \sum m(0, 1, 3, 6, 8, 9, 14, 15)$. Derive an implementation using the minimum possible number of three-input LUTs.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all ;

ENTITY barrel IS
    PORT ( w  : IN   UNSIGNED(3 DOWNTO 0) ;
           s  : IN   UNSIGNED(1 DOWNTO 0) ) ;
           y  : OUT  UNSIGNED(3 DOWNTO 0) ) ;
END barrel ;

ARCHITECTURE Behavior OF barrel IS
BEGIN
    PROCESS (s, w)
    BEGIN
        CASE s IS
            WHEN "00" =>
                y <= w ;
            WHEN "01" =>
                y <= w ROR 1 ;
            WHEN "10" =>
                y <= w ROR 2 ;
            WHEN OTHERS =>
                y <= w ROR 3 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.61 VHDL code that specifies the barrel shifter circuit in Figure 6.57.

***6.14** Give two examples of logic functions with five inputs, $w_1, \ldots, w_5$, that can be realized using 2 four-input LUTs.

6.15 For the function, f, in Example 6.27 perform Shannon's expansion with respect to variables w_1 and w_2, rather than w_1 and w_4. How does the resulting circuit compare with the circuit in Figure 6.52?

6.16 Actel Corporation manufactures an FPGA family called Act 1, which has the multiplexer-based logic block illustrated in Figure P6.1. Show how the function $f = w_2\overline{w}_3 + w_1w_3 + \overline{w}_2w_3$ can be implemented using only one Act 1 logic block.

6.17 Show how the function $f = w_1\overline{w}_3 + \overline{w}_1w_3 + w_2\overline{w}_3 + w_1\overline{w}_2$ can be realized using Act 1 logic blocks. Note that there are no NOT gates in the chip; hence complements of signals have to be generated using the multiplexers in the logic block.

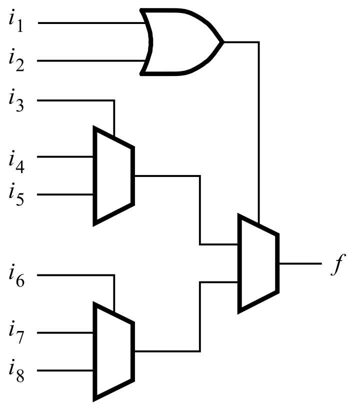


Figure P6.1 The Actel Act 1 logic block.

***6.18** Consider the VHDL code in Figure P6.2. What type of circuit does the code represent? Comment on whether or not the style of code used is a good choice for the circuit that it represents.

6.19 Write VHDL code that represents the function in problem 6.1, using one selected signal assignment.

6.20 Write VHDL code that represents the function in problem 6.2, using one selected signal assignment.

6.21 Using a selected signal assignment, write VHDL code for a 4-to-2 binary encoder.

6.22 Using a conditional signal assignment, write VHDL code for an 8-to-3 binary encoder.

6.23 Derive the circuit for an 8-to-3 priority encoder.

6.24 Using a conditional signal assignment, write VHDL code for an 8-to-3 priority encoder.

6.25 Repeat problem 6.24, using an if-then-else statement.

6.26 Create a VHDL entity named *if2to4* that represents a 2-to-4 binary decoder using an if-then-else statement. Create a second entity named *h3to8* that represents the 3-to-8 binary decoder in Figure 6.17, using two instances of the *if2to4* entity.

6.27 Create a VHDL entity named *h6to64* that represents a 6-to-64 binary decoder. Use the treelike structure in Figure 6.18, in which the 6-to-64 decoder is built using five instances of the *h3to8* decoder created in problem 6.26.

6.28 Write VHDL code for a BCD-to-7-segment code converter, using a selected signal assignment.

***6.29** Derive minimal sum-of-products expressions for the outputs a, b, and c of the 7-segment display in Figure 6.25.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( w                : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En               : IN    STD_LOGIC ;
           y0, y1, y2, y3   : OUT   STD_LOGIC ) ;
END problem ;

ARCHITECTURE Behavior OF problem IS
BEGIN
    PROCESS (w, En)
    BEGIN
        y0 <= '0' ; y1 <= '0' ; y2 <= '0' ; y3 <= '0' ;
        IF En = '1' THEN
            IF w = "00" THEN y0 <= '1' ;
            ELSIF w = "01" THEN y1 <= '1' ;
            ELSIF w = "10" THEN y2 <= '1' ;
            ELSE y3 <= '1' ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure P6.2 Code for problem 6.18.

6.30 Derive minimal sum-of-products expressions for the outputs d, e, f, and g of the 7-segment display in Figure 6.25.

6.31 Design a shifter circuit, similar to the one in Figure 6.56, which can shift a four-bit input vector, $W = w_3w_2w_1w_0$, one bit-position to the right when the control signal *Right* is equal to 1, and one bit-position to the left when the control signal *Left* is equal to 1. When *Right* = *Left* = 0, the output of the circuit should be the same as the input vector. Assume that the condition *Right* = *Left* = 1 will never occur.

6.32 Figure 6.21 shows a block diagram of a ROM. A circuit that implements a small ROM, with four rows and four columns, is depicted in Figure P6.3. Each X in the figure represents a switch that determines whether the ROM produces a 1 or 0 when that location is read.
(a) Show how a switch (X) can be realized using a single NMOS transistor.
(b) Draw the complete 4×4 ROM circuit, using your switches from part (*a*). The ROM should be programmed to store the bits 0101 in row 0 (the top row), 1010 in row 1, 1100 in row 2, and 0011 in row 3 (the bottom row).
(c) Show how each (X) can be implemented as a programmable switch (as opposed to providing either a 1 or 0 permanently), using an EEPROM cell as shown in Figure 3.64. Briefly describe how the storage cell is used.

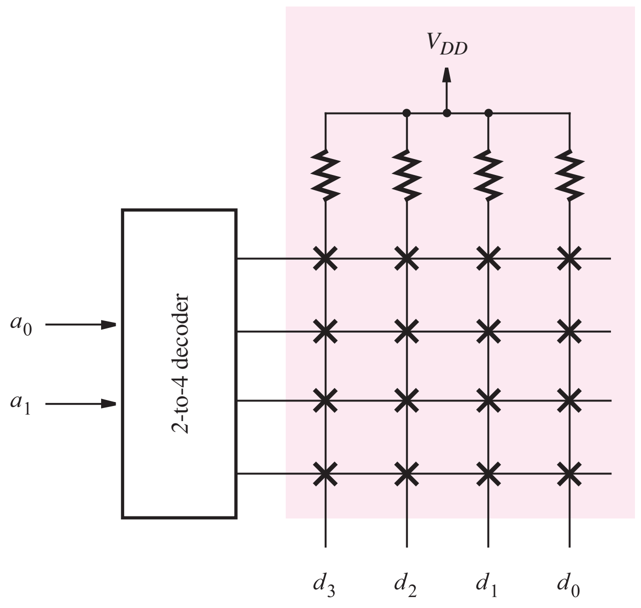


Figure P6.3 A 4 × 4 ROM circuit.

6.33 Show the complete circuit for a ROM using the storage cells designed in Part (*a*) of problem 6.33 that realizes the logic functions

$$d_3 = a_0 \oplus a_1$$
$$d_2 = \overline{a_0 \oplus a_1}$$
$$d_1 = a_0 a_1$$
$$d_0 = a_0 + a_1$$

REFERENCES

1. C. E. Shannon, "Symbolic Analysis of Relay and Switching Circuits," *Transactions AIEE* 57 (1938), pp. 713–723.
2. Actel Corporation, "MX FPGA Data Sheet," *http://www.actel.com.*
3. QuickLogic Corporation, "pASIC 3 FPGA Data Sheet," *http://www.quicklogic.com.*

4. R. Landers, S. Mahant-Shetti, and C. Lemonds, "A Multiplexer-Based Architecture for High-Density, Low Power Gate Arrays," *IEEE Journal of Solid-State Circuits* 30, no. 4 (April 1995).
5. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2nd ed. (McGraw-Hill: New York, 1998).
6. J. Bhasker, *A VHDL Primer*, 3rd ed. (Prentice-Hall: Englewood Cliffs, NJ, 1998).
7. D. L. Perry, *VHDL*, 3rd ed. (McGraw-Hill: New York, 1998).
8. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
9. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, 1997).
10. D. J. Smith, *HDL Chip Design* (Doone Publications: Madison, AL, 1996).

chapter

7 FLIP-FLOPS, REGISTERS, COUNTERS, AND A SIMPLE PROCESSOR

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Logic circuits that can store information
- Flip-flops, which store a single bit
- Registers, which store multiple bits
- Shift registers, which shift the contents of the register
- Counters of various types
- VHDL constructs used to implement storage elements
- Design of small subsystems
- Timing considerations

In previous chapters we considered combinational circuits where the value of each output depends solely on the values of signals applied to the inputs. There exists another class of logic circuits in which the values of the outputs depend not only on the present values of the inputs but also on the past behavior of the circuit. Such circuits include storage elements that store the values of logic signals. The contents of the storage elements are said to represent the *state* of the circuit. When the circuit's inputs change values, the new input values either leave the circuit in the same state or cause it to change into a new state. Over time the circuit changes through a sequence of states as a result of changes in the inputs. Circuits that behave in this way are referred to as *sequential circuits*.

In this chapter we will introduce circuits that can be used as storage elements. But first, we will motivate the need for such circuits by means of a simple example. Suppose that we wish to control an alarm system, as shown in Figure 7.1. The alarm mechanism responds to the control input $On/\overline{Off}$. It is turned on when $On/\overline{Off} = 1$, and it is off when $On/\overline{Off} = 0$. The desired operation is that the alarm turns on when the sensor generates a positive voltage signal, *Set*, in response to some undesirable event. Once the alarm is triggered, it must remain active even if the sensor output goes back to zero. The alarm is turned off manually by means of a *Reset* input. The circuit requires a memory element to remember that the alarm has to be active until the *Reset* signal arrives.

Figure 7.2 gives a rudimentary memory element, consisting of a loop that has two inverters. If we assume that $A = 0$, then $B = 1$. The circuit will maintain these values indefinitely. We say that the circuit is in the *state* defined by these values. If we assume that $A = 1$, then $B = 0$, and the circuit will remain in this second state indefinitely. Thus the circuit has two possible states. This circuit is not useful, because it lacks some practical means for changing its state.

A more useful circuit is shown in Figure 7.3. It includes a mechanism for changing the state of the circuit in Figure 7.2, using two transmission gates of the type discussed in section 3.9. One transmission gate, $TG1$, is used to connect the *Data* input terminal to point

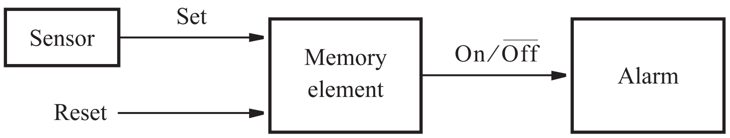


Figure 7.1 Control of an alarm system.

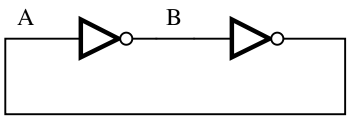


Figure 7.2 A simple memory element.

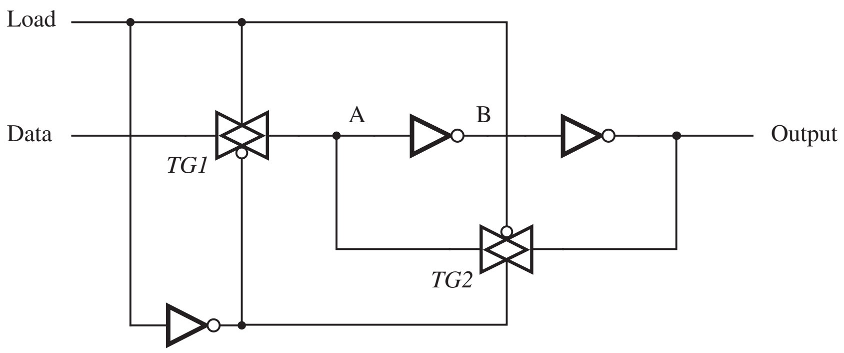


Figure 7.3 A controlled memory element.

A in the circuit. The second, *TG*2, is used as a switch in the *feedback loop* that maintains the state of the circuit. The transmission gates are controlled by the *Load* signal. If $Load = 1$, then *TG*1 is on and the point *A* will have the same value as the *Data* input. Since the value presently stored at *Output* may not be the same value as *Data*, the feedback loop is broken by having *TG*2 turned off when $Load = 1$. When *Load* changes to zero, then *TG*1 turns off and *TG*2 turns on. The feedback path is closed and the memory element will retain its state as long as $Load = 0$. This memory element cannot be applied directly to the system in Figure 7.1, but it is useful for many other applications, as we will see later.

7.1 Basic Latch

Instead of using the transmission gates, we can construct a similar circuit using ordinary logic gates. Figure 7.4 presents a memory element built with NOR gates. Its inputs, *Set* and *Reset*, provide the means for changing the state, Q, of the circuit. A more usual way of drawing this circuit is given in Figure 7.5*a*, where the two NOR gates are said to be connected in cross-coupled style. The circuit is referred to as a *basic latch*. Its behavior is described by the table in Figure 7.5*b*. When both inputs, *R* and *S*, are equal to 0 the latch maintains its existing state. This state may be either $Q_a = 0$ and $Q_b = 1$, or $Q_a = 1$ and $Q_b = 0$, which is indicated in the table by stating that the Q_a and Q_b outputs have values

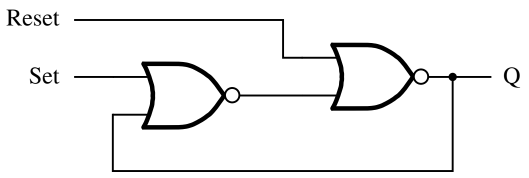
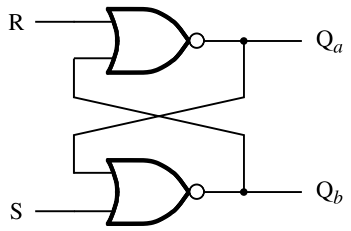


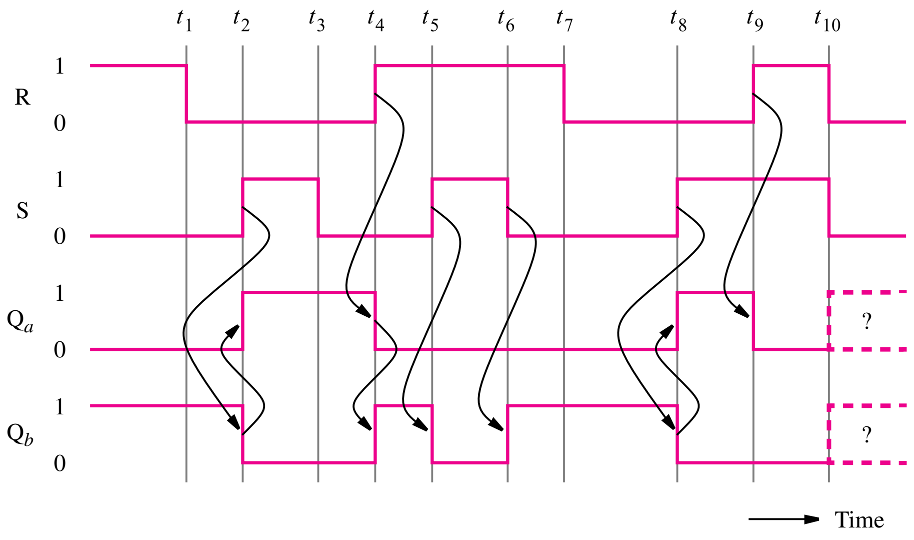
Figure 7.4 A memory element with NOR gates.



(a) Circuit

S	R	Q_a	Q_b	
0	0	0/1	1/0	(no change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Characteristic table



(c) Timing diagram

Figure 7.5 A basic latch built with NOR gates.

0/1 and 1/0, respectively. Observe that Q_a and Q_b are complements of each other in this case. When $R = 0$ and $S = 1$, the latch is *set* into a state where $Q_a = 1$ and $Q_b = 0$. When $R = 1$ and $S = 0$, the latch is *reset* into a state where $Q_a = 0$ and $Q_b = 1$. The fourth possibility is to have $R = S = 1$. In this case both Q_a and Q_b will be 0. The table in Figure 7.5*b* resembles a truth table. However, since it does not represent a combinational circuit in which the values of the outputs are determined solely by the current values of the inputs, it is often called a *characteristic table* rather than a truth table.

Figure 7.5*c* gives a timing diagram for the latch, assuming that the propagation delay through the NOR gates is negligible. Of course, in a real circuit the changes in the waveforms would be delayed according to the propagation delays of the gates. We assume that initially $Q_a = 0$ and $Q_b = 1$. The state of the latch remains unchanged until time t_2, when S becomes equal to 1, causing Q_b to change to 0, which in turn causes Q_a to change to 1.

The causality relationship is indicated by the arrows in the diagram. When S goes to 0 at t_3, there is no change in the state because both S and R are then equal to 0. At t_4 we have $R = 1$, which causes Q_a to go to 0, which in turn causes Q_b to go to 1. At t_5 both S and R are equal to 1, which forces both Q_a and Q_b to be equal to 0. As soon as S returns to 0, at t_6, Q_b becomes equal to 1 again. At t_8 we have $S = 1$ and $R = 0$, which causes $Q_b = 0$ and $Q_a = 1$. An interesting situation occurs at t_{10}. From t_9 to t_{10} we have $Q_a = Q_b = 0$ because $R = S = 1$. Now if both R and S change to 0 at t_{10}, both Q_a and Q_b will go to 1. But having both Q_a and Q_b equal to 1 will immediately force $Q_a = Q_b = 0$. There will be an oscillation between $Q_a = Q_b = 0$ and $Q_a = Q_b = 1$. If the delays through the two NOR gates are exactly the same, the oscillation will continue indefinitely. In a real circuit there will invariably be some difference in the delays through these gates, and the latch will eventually settle into one of its two stable states, but we don't know which state it will be. This uncertainty is indicated in the waveforms by dashed lines.

The oscillations discussed above illustrate that even though the basic latch is a simple circuit, careful analysis has to be done to fully appreciate its behavior. In general, any circuit that contains one or more feedback paths, such that the state of the circuit depends on the propagation delays through logic gates, has to be designed carefully. We discuss timing issues in detail in Chapter 9.

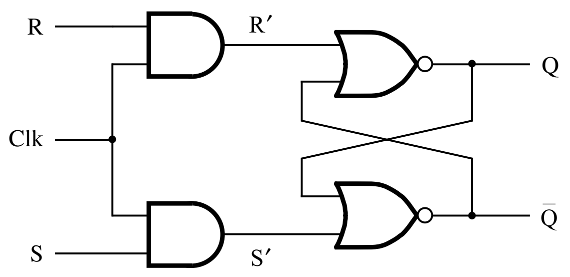
The latch in Figure 7.5*a* can perform the functions needed for the memory element in Figure 7.1, by connecting the *Set* signal to the S input and *Reset* to the R input. The Q_a output provides the desired $On/\overline{Off}$ signal. To initialize the operation of the alarm system, the latch is reset. Thus the alarm is off. When the sensor generates the logic value 1, the latch is set and Q_a becomes equal to 1. This turns on the alarm mechanism. If the sensor output returns to 0, the latch retains its state where $Q_a = 1$; hence the alarm remains turned on. The only way to turn off the alarm is by resetting the latch, which is accomplished by making the *Reset* input equal to 1.

7.2 Gated SR Latch

In section 7.1 we saw that the basic SR latch can serve as a useful memory element. It remembers its state when both the S and R inputs are 0. It changes its state in response to changes in the signals on these inputs. The state changes occur at the time when the changes in the signals occur. If we cannot control the time of such changes, then we don't know when the latch may change its state.

In the alarm system of Figure 7.1, it may be desirable to be able to enable or disable the entire system by means of a control input, *Enable*. Thus when enabled, the system would function as described above. In the disabled mode, changing the *Set* input from 0 to 1 would not cause the alarm to turn on. The latch in Figure 7.5*a* cannot provide the desired operation. But the latch circuit can be modified to respond to the input signals S and R only when $Enable = 1$. Otherwise, it would maintain its state.

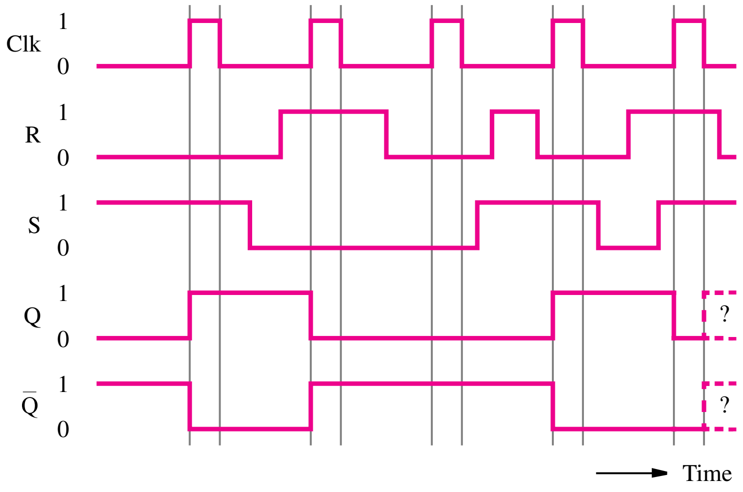
The modified circuit is depicted in Figure 7.6*a*. It includes two AND gates that provide the desired control. When the control signal *Clk* is equal to 0, the S' and R' inputs to the latch will be 0, regardless of the values of signals S and R. Hence the latch will maintain its



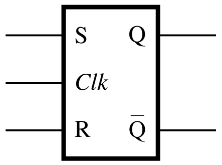
(a) Circuit

Clk	S	R	$Q(t+1)$
0	x	x	$Q(t)$ (no change)
1	0	0	$Q(t)$ (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table



(c) Timing diagram



(d) Graphical symbol

Figure 7.6 Gated SR latch.

existing state as long as $Clk = 0$. When Clk changes to 1, the S' and R' signals will be the same as the S and R signals, respectively. Therefore, in this mode the latch will behave as we described in section 7.1. Note that we have used the name *Clk* for the control signal that allows the latch to be set or reset, rather than call it the *Enable* signal. The reason is that such circuits are often used in digital systems where it is desirable to allow the changes in the states of memory elements to occur only at well-defined time intervals, as if they were controlled by a clock. The control signal that defines these time intervals is usually called the *clock* signal. The name *Clk* is meant to reflect this nature of the signal.

Circuits of this type, which use a control signal, are called *gated latches*. Because our circuit exhibits set and reset capability, it is called a *gated SR latch*. Figure 7.6*b* describes its behavior. It defines the state of the Q output at time $t+1$, namely, Q($t+1$), as a function of the inputs S, R, and *Clk*. When $Clk = 0$, the latch will remain in the state it is in at time t, that is, Q(t), regardless of the values of inputs S and R. This is indicated by specifying S = x and R = x, where x means that the signal value can be either 0 or 1. (Recall that we already used this notation in Chapter 4.) When $Clk = 1$, the circuit behaves as the basic latch in Figure 7.5. It is set by $S = 1$ and reset by $R = 1$. The last row of the table, where $S = R = 1$, shows that the state Q($t + 1$) is undefined because we don't know whether it will be 0 or 1. This corresponds to the situation described in section 7.1 in conjunction with the timing diagram in Figure 7.5 at time t_{10}. At this time both S and R inputs go from 1 to 0, which causes the oscillatory behavior that we discussed. If $S = R = 1$, this situation will occur as soon as *Clk* goes from 1 to 0. To ensure a meaningful operation of the gated SR latch, it is essential to avoid the possibility of having both the S and R inputs equal to 1 when *Clk* changes from 1 to 0.

A timing diagram for the gated SR latch is given in Figure 7.6*c*. It shows *Clk* as a periodic signal that is equal to 1 at regular time intervals to suggest that this is how the clock signal usually appears in a real system. The diagram presents the effect of several combinations of signal values. Observe that we have labeled one output as Q and the other as its complement $\overline{\text{Q}}$, rather than Q_a and Q_b as in Figure 7.5. Since the undefined mode, where $S = R = 1$, must be avoided in practice, the normal operation of the latch will have the outputs as complements of each other. Moreover, we will often say that the latch is *set* when Q = 1, and it is *reset* when Q = 0. A graphical symbol for the gated SR latch is given in Figure 7.6*d*.

7.2.1 Gated SR Latch with NAND Gates

So far we have implemented the basic latch with cross-coupled NOR gates. We can also construct the latch with NAND gates. Using this approach, we can implement the gated SR latch as depicted in Figure 7.7. The behavior of this circuit is described by the table in Figure 7.6*b*. Note that in this circuit, the clock is gated by NAND gates, rather than by

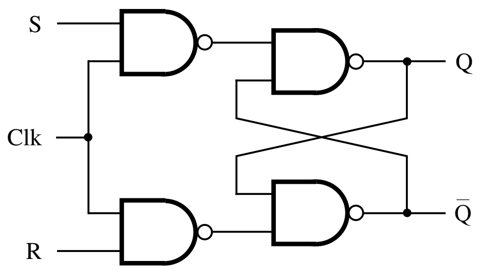


Figure 7.7 Gated SR latch with NAND gates.

AND gates. Note also that the S and R inputs are reversed in comparison with the circuit in Figure 7.6*a*. The circuit with NAND gates requires fewer transistors than the circuit with AND gates. We will use the circuit in Figure 7.7, in preference to the circuit in Figure 7.6*a*.

7.3 GATED D LATCH

In section 7.2 we presented the gated SR latch and showed how it can be used as the memory element in the alarm system of Figure 7.1. This latch is useful for many other applications. In this section we describe another gated latch that is even more useful in practice. It has a single data input, called D, and it stores the value on this input, under the control of a clock signal. It is called a *gated D latch*.

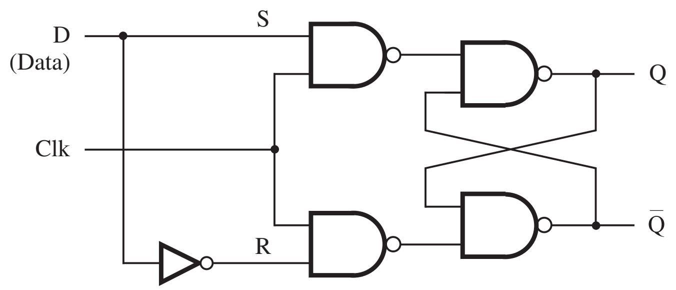
To motivate the need for a gated D latch, consider the adder/subtractor unit discussed in Chapter 5 (Figure 5.13). When we described how that circuit is used to add numbers, we did not discuss what is likely to happen with the sum bits that are produced by the adder. Adder/subtractor units are often used as part of a computer. The result of an addition or subtraction operation is often used as an operand in a subsequent operation. Therefore, it is necessary to be able to remember the values of the sum bits generated by the adder until they are needed again. We might think of using the basic latches to remember these bits, one bit per latch. In this context, instead of saying that a latch remembers the value of a bit, it is more illuminating to say that the latch *stores* the value of the bit or simply "stores the bit." We should think of the latch as a storage element.

But can we obtain the desired operation using the basic latches? We can certainly reset all latches before the addition operation begins. Then we would expect that by connecting a sum bit to the S input of a latch, the latch would be set to 1 if the sum bit has the value 1; otherwise, the latch would remain in the 0 state. This would work fine if all sum bits are 0 at the start of the addition operation and, after some propagation delay through the adder, some of these bits become equal to 1 to give the desired sum. Unfortunately, the propagation delays that exist in the adder circuit cause a big problem in this arrangement. Suppose that we use a ripple-carry adder. When the X and Y inputs are applied to the adder, the sum outputs may alternate between 0 and 1 a number of times as the carries ripple through the circuit. This situation was illustrated in the timing diagram in Figure 5.21. The problem is that if we connect a sum bit to the S input of a latch, then if the sum bit is temporarily a 1 and then settles to 0 in the final result, the latch will remain set to 1 erroneously.

The problem caused by the alternating values of the sum bits in the adder could be solved by using the gated SR latches, instead of the basic latches. Then we could arrange that the clock signal is 0 during the time needed by the adder to produce a correct sum. After allowing for the maximum propagation delay in the adder circuit, the clock should go to 1 to store the values of the sum bits in the gated latches. As soon as the values have been stored, the clock can return to 0, which ensures that the stored values will be retained until the next time the clock goes to 1. To achieve the desired operation, we would also have to reset all latches to 0 prior to loading the sum-bit values into these latches. This is an awkward way of dealing with the problem, and it is preferable to use the gated D latches instead.

Figure 7.8*a* shows the circuit for a gated D latch. It is based on the gated SR latch, but instead of using the *S* and *R* inputs separately, it has just one data input, *D*. For convenience we have labeled the points in the circuit that are equivalent to the *S* and *R* inputs. If $D = 1$, then $S = 1$ and $R = 0$, which forces the latch into the state Q = 1. If $D = 0$, then $S = 0$ and $R = 1$, which causes Q = 0. Of course, the changes in state occur only when $Clk = 1$.

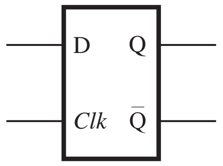
It is important to observe that in this circuit it is impossible to have the troublesome situation where $S = R = 1$. In the gated D latch, the output Q merely tracks the value of the input *D* while $Clk = 1$. As soon as *Clk* goes to 0, the state of the latch is frozen until the next time the clock signal goes to 1. Therefore, the gated D latch stores the value of the *D*



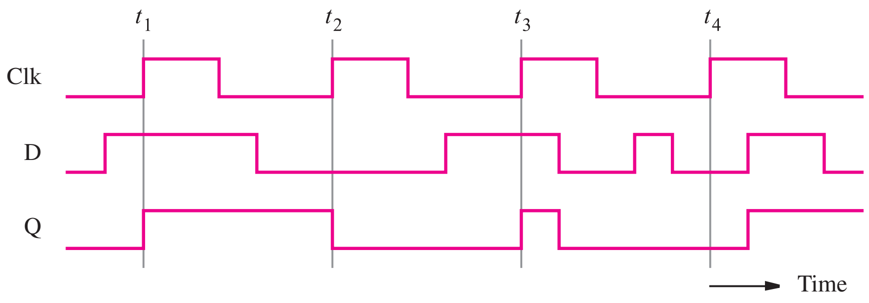
(a) Circuit

Clk	D	Q(*t* + 1)
0	x	Q(*t*)
1	0	0
1	1	1

(b) Characteristic table



(c) Graphical symbol



(d) Timing diagram

Figure 7.8 Gated D latch.

input seen at the time the clock changes from 1 to 0. Figure 7.8 also gives the characteristic table, the graphical symbol, and the timing diagram for the gated D latch.

The timing diagram illustrates what happens if the D signal changes while $Clk = 1$. During the third clock pulse, starting at t_3, the output Q changes to 1 because $D = 1$. But midway through the pulse D goes to 0, which causes Q to go to 0. This value of Q is stored when Clk changes to 0. Now no further change in the state of the latch occurs until the next clock pulse, at t_4. The key point to observe is that as long as the clock has the value 1, the Q output follows the D input. But when the clock has the value 0, the Q output cannot change. In Chapter 3 we saw that the logic values are implemented as low and high voltage levels. Since the output of the gated D latch is controlled by the level of the clock input, the latch is said to be *level sensitive*. The circuits in Figures 7.6 through 7.8 are level sensitive. We will show in section 7.4 that it is possible to design storage elements for which the output changes only at the point in time when the clock changes from one value to the other. Such circuits are said to be *edge triggered*.

At this point we should reconsider the circuit in Figure 7.3. Careful examination of that circuit shows that it behaves in exactly the same way as the circuit in Figure 7.8*a*. The *Data* and *Load* inputs correspond to the D and Clk inputs, respectively. The *Output*, which has the same signal value as point A, corresponds to the Q output. Point B corresponds to $\overline{\text{Q}}$. Therefore, the circuit in Figure 7.3 is also a gated D latch. An advantage of this circuit is that it can be implemented using fewer transistors than the circuit in Figure 7.8*a*.

7.3.1 Effects of Propagation Delays

In the previous discussion we ignored the effects of propagation delays. In practical circuits it is essential to take these delays into account. Consider the gated D latch in Figure 7.8*a*. It stores the value of the D input that is present at the time the clock signal changes from 1 to 0. It operates properly if the D signal is stable (that is, not changing) at the time Clk goes from 1 to 0. But it may lead to unpredictable results if the D signal also changes at this time. Therefore, the designer of a logic circuit that generates the D signal must ensure that this signal is stable when the critical change in the clock signal takes place.

Figure 7.9 illustrates the critical timing region. The minimum time that the D signal must be stable prior to the negative edge of the Clk signal is called the *setup time*, t_{su}, of the

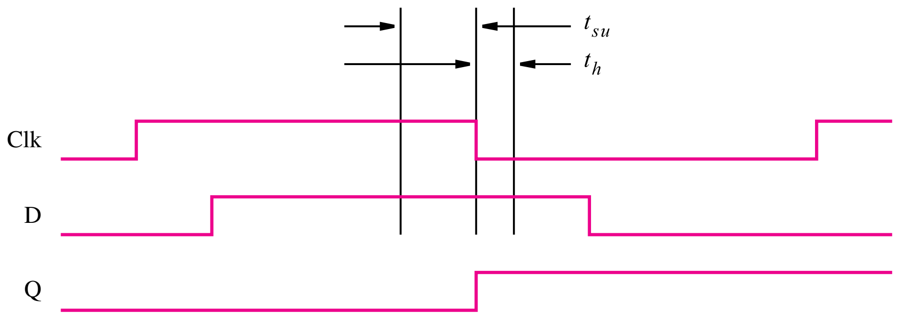


Figure 7.9 Setup and hold times.

latch. The minimum time that the *D* signal must remain stable after the negative edge of the *Clk* signal is called the *hold time*, t_h, of the latch. The values of t_{su} and t_h depend on the technology used. Manufacturers of integrated circuit chips provide this information on the data sheets that describe their chips. Typical values for a modern CMOS technology may be $t_{su} = 0.3$ ns and $t_h = 0.2$ ns. We will give examples of how setup and hold times affect the speed of operation of circuits in section 7.13. The behavior of storage elements when setup or hold times are violated is discussed in section 10.3.3.

7.4 Master-Slave and Edge-Triggered D Flip-Flops

In the level-sensitive latches, the state of the latch keeps changing according to the values of input signals during the period when the clock signal is active (equal to 1 in our examples). As we will see in sections 7.8 and 7.9, there is also a need for storage elements that can change their states no more than once during one clock cycle. We will discuss two types of circuits that exhibit such behavior.

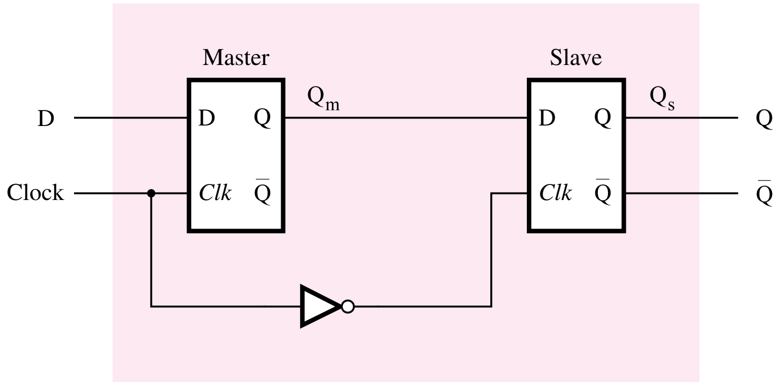
7.4.1 Master-Slave D Flip-Flop

Consider the circuit given in Figure 7.10*a*, which consists of two gated D latches. The first, called *master*, changes its state while *Clock* = 1. The second, called *slave*, changes its state while *Clock* = 0. The operation of the circuit is such that when the clock is high, the master tracks the value of the *D* input signal and the slave does not change. Thus the value of Q_m follows any changes in *D*, and the value of Q_s remains constant. When the clock signal changes to 0, the master stage stops following the changes in the *D* input. At the same time, the slave stage responds to the value of the signal Q_m and changes state accordingly. Since Q_m does not change while *Clock* = 0, the slave stage can undergo at most one change of state during a clock cycle. From the external observer's point of view, namely, the circuit connected to the output of the slave stage, the master-slave circuit changes its state at the negative-going edge of the clock. The *negative edge* is the edge where the clock signal changes from 1 to 0. Regardless of the number of changes in the *D* input to the master stage during one clock cycle, the observer of the Q_s signal will see only the change that corresponds to the *D* input at the negative edge of the clock.

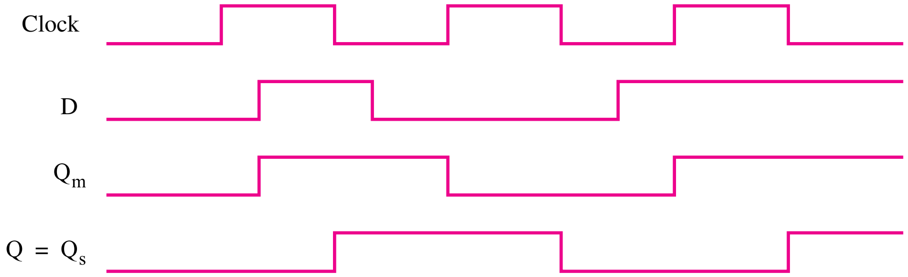
The circuit in Figure 7.10 is called a *master-slave D flip-flop*. The term *flip-flop* denotes a storage element that changes its output state at the edge of a controlling clock signal. The timing diagram for this flip-flop is shown in Figure 7.10*b*. A graphical symbol is given in Figure 7.10*c*. In the symbol we use the $>$ mark to denote that the flip-flop responds to the "active edge" of the clock. We place a bubble on the clock input to indicate that the active edge for this particular circuit is the negative edge.

7.4.2 Edge-Triggered D Flip-Flop

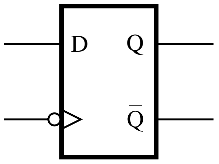
The output of the master-slave D flip-flop in Figure 7.10*a* responds on the negative edge of the clock signal. The circuit can be changed to respond to the positive clock edge by connecting the slave stage directly to the clock and the master stage to the complement of



(a) Circuit



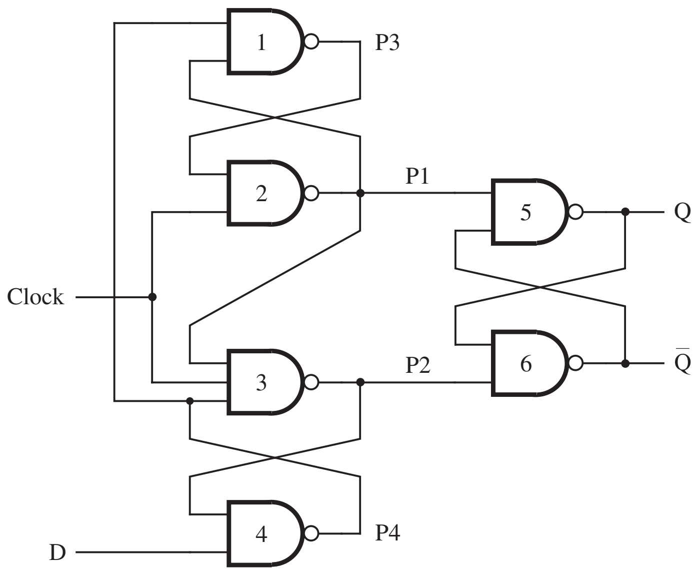
(b) Timing diagram



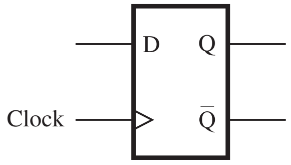
(c) Graphical symbol

Figure 7.10 Master-slave D flip-flop.

the clock. A different circuit that accomplishes the same task is presented in Figure 7.11*a*. It requires only six NAND gates and, hence, fewer transistors. The operation of the circuit is as follows. When $Clock = 0$, the outputs of gates 2 and 3 are high. Thus $P1 = P2 = 1$, which maintains the output latch, comprising gates 5 and 6, in its present state. At the same time, the signal $P3$ is equal to D, and $P4$ is equal to its complement $\overline{D}$. When *Clock* changes



(a) Circuit



(b) Graphical symbol

Figure 7.11 A positive-edge-triggered D flip-flop.

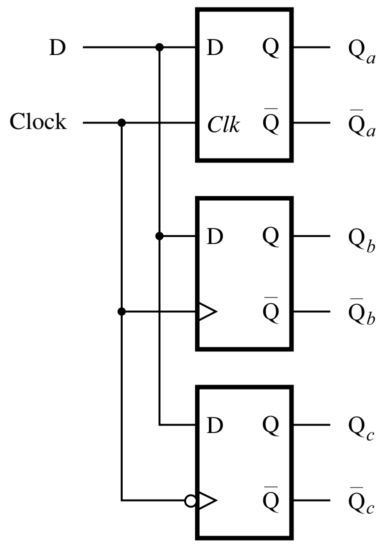
to 1, the following changes take place. The values of $P3$ and $P4$ are transmitted through gates 2 and 3 to cause $P1 = \overline{D}$ and $P2 = D$, which sets $\mathrm{Q} = D$ and $\overline{\mathrm{Q}} = \overline{D}$. To operate reliably, $P3$ and $P4$ must be stable when *Clock* changes from 0 to 1. Hence the setup time of the flip-flop is equal to the delay from the D input through gates 4 and 1 to $P3$. The hold time is given by the delay through gate 3 because once $P2$ is stable, the changes in D no longer matter.

For proper operation it is necessary to show that, after *Clock* changes to 1, any further changes in D will not affect the output latch as long as $Clock = 1$. We have to consider two cases. Suppose first that $D = 0$ at the positive edge of the clock. Then $P2 = 0$, which will keep the output of gate 4 equal to 1 as long as $Clock = 1$, regardless of the value of the D input. The second case is if $D = 1$ at the positive edge of the clock. Then $P1 = 0$, which forces the outputs of gates 1 and 3 to be equal to 1, regardless of the D input. Therefore, the flip-flop ignores changes in the D input while $Clock = 1$.

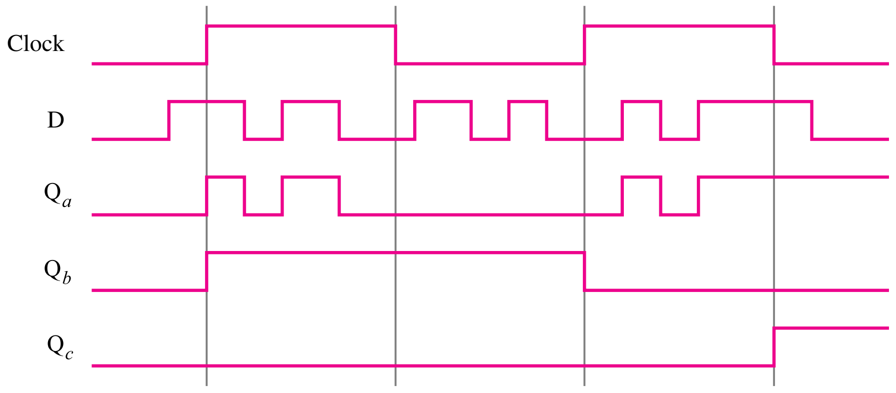
Figure 7.11*b* gives a graphical symbol for this flip-flop. The clock input indicates that the positive edge of the clock is the active edge. A similar circuit, constructed with NOR gates, can be used as a negative-edge-triggered flip-flop.

Level-Sensitive versus Edge-Triggered Storage Elements

Figure 7.12 shows three different types of storage elements that are driven by the same data and clock inputs. The first element is a gated D latch, which is level sensitive. The second one is a positive-edge-triggered D flip-flop, and the third one is a negative-edge-triggered D flip-flop. To accentuate the differences between these storage elements, the



(a) Circuit



(b) Timing diagram

Figure 7.12 Comparison of level-sensitive and edge-triggered D storage elements.

D input changes its values more than once during each half of the clock cycle. Observe that the gated D latch follows the *D* input as long as the clock is high. The positive-edge-triggered flip-flop responds only to the value of *D* when the clock changes from 0 to 1. The negative-edge-triggered flip-flop responds only to the value of *D* when the clock changes from 1 to 0.

7.4.3 D Flip-Flops with Clear and Preset

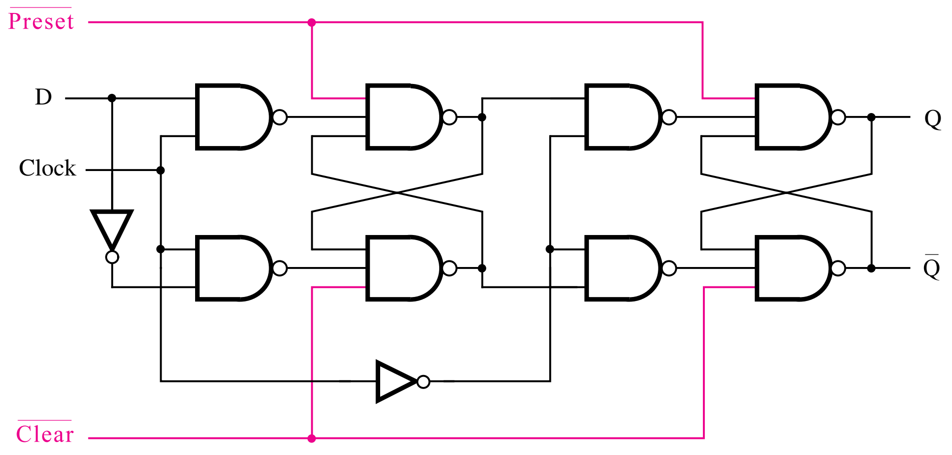
Flip-flops are often used for implementation of circuits that can have many possible states, where the response of the circuit depends not only on the present values of the circuit's inputs but also on the particular state that the circuit is in at that time. We will discuss a general form of such circuits in Chapter 8. A simple example is a counter circuit that counts the number of occurrences of some event, perhaps passage of time. We will discuss counters in detail in section 7.9. A counter comprises a number of flip-flops, whose outputs are interpreted as a number. The counter circuit has to be able to increment or decrement the number. It is also important to be able to force the counter into a known initial state (count). Obviously, it must be possible to clear the count to zero, which means that all flip-flops must have Q = 0. It is equally useful to be able to preset each flip-flop to Q = 1, to insert some specific count as the initial value in the counter. These features can be incorporated into the circuits of Figures 7.10 and 7.11 as follows.

Figure 7.13*a* shows an implementation of the circuit in Figure 7.10*a* using NAND gates. The master stage is just the gated D latch of Figure 7.8*a*. Instead of using another latch of the same type for the slave stage, we can use the slightly simpler gated SR latch of Figure 7.7. This eliminates one NOT gate from the circuit.

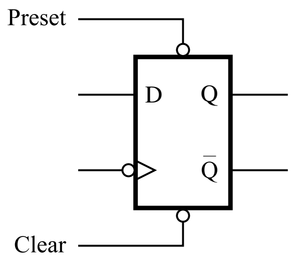
A simple way of providing the clear and preset capability is to add an extra input to each NAND gate in the cross-coupled latches, as indicated in blue. Placing a 0 on the *Clear* input will force the flip-flop into the state Q = 0. If *Clear* = 1, then this input will have no effect on the NAND gates. Similarly, *Preset* = 0 forces the flip-flop into the state Q = 1, while *Preset* = 1 has no effect. To denote that the *Clear* and *Preset* inputs are active when their value is 0, we placed an overbar on the names in the figure. We should note that the circuit that uses this flip-flop should not try to force both *Clear* and *Preset* to 0 at the same time. A graphical symbol for this flip-flop is shown in Figure 7.13*b*.

A similar modification can be done on the edge-triggered flip-flop of Figure 7.11*a*, as indicated in Figure 7.14*a*. Again, both *Clear* and *Preset* inputs are active low. They do not disturb the flip-flop when they are equal to 1.

In the circuits in Figures 7.13*a* and 7.14*a*, the effect of a low signal on either the *Clear* or *Preset* input is immediate. For example, if *Clear* = 0 then the flip-flop goes into the state Q = 0 immediately, regardless of the value of the clock signal. In such a circuit, where the *Clear* signal is used to clear a flip-flop without regard to the clock signal, we say that the flip-flop has an *asynchronous clear*. In practice, it is often preferable to clear the flip-flops on the active edge of the clock. Such *synchronous clear* can be accomplished as shown in Figure 7.14*c*. The flip-flop operates normally when the *Clear* input is equal to 1. But if *Clear* goes to 0, then on the next positive edge of the clock the flip-flop will be cleared to 0. We will examine the clearing of flip-flops in more detail in section 7.10.



(a) Circuit



(b) Graphical symbol

Figure 7.13 Master-slave D flip-flop with *Clear* and *Preset*.

7.4.4 Flip-Flop Timing Parameters

In section 7.3.1 we discussed timing issues related to latch circuits. In practice such issues are equally important for circuits with flip-flops. Figure 7.15*a* shows a positive-edge triggered flip-flop with asynchronous clear, and part *b* of the figure illustrates some important timing parameters for this flip-flop. Data is loaded into the D input of the flip-flop on a positive clock edge, and this logic value must be stable during the setup time, t_{su}, before the clock edge occurs. The data must remain stable during the hold time, t_h, after the edge. If the setup or hold requirements are not adhered to in a circuit that uses this flip-flop, then it may enter an unstable condition known as *metastability*; we discuss this concept in section 10.3.

As indicated in Figure 7.15, a clock-to-Q propagation delay, t_{cQ}, is incurred before the value of Q changes after a positive clock edge. In general, the delay may not be

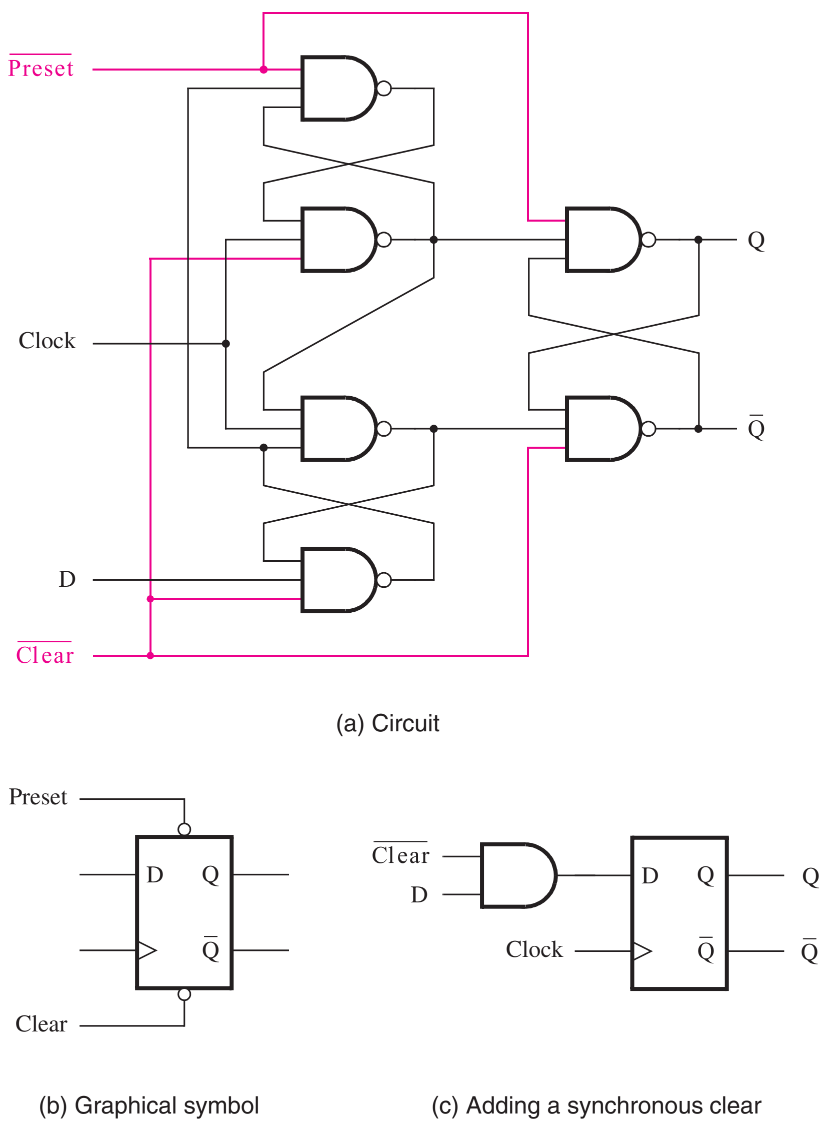
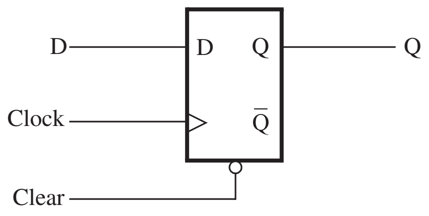
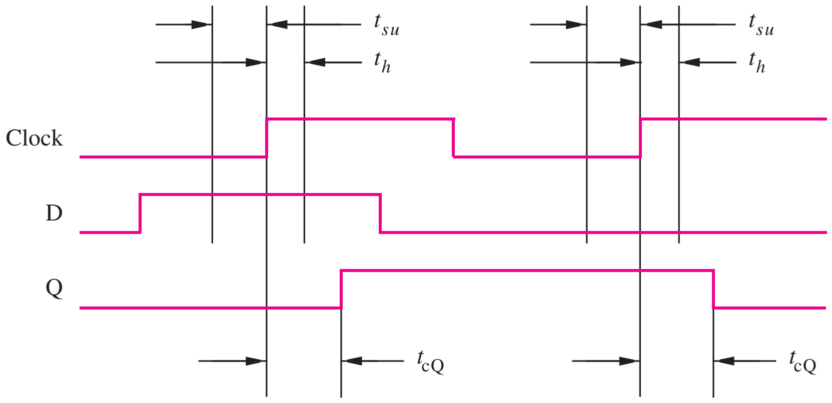


Figure 7.14 Positive-edge-triggered D flip-flop with *Clear* and *Preset*.

exactly the same for the cases when Q changes from 1 to 0 or 0 to 1, but we assume for simplicity that these delays are equal. For the flip-flops in a commercial chip, two values are usually specified for t_{cQ}, representing the maximum and minimum delays that may occur in practice. Specifying a range of values when estimating the delays in a chip is a common practice due to many sources of variation in delay that are caused by the chip manufacturing



(a) D flip-flop with asynchronous clear



(b) Timing diagram

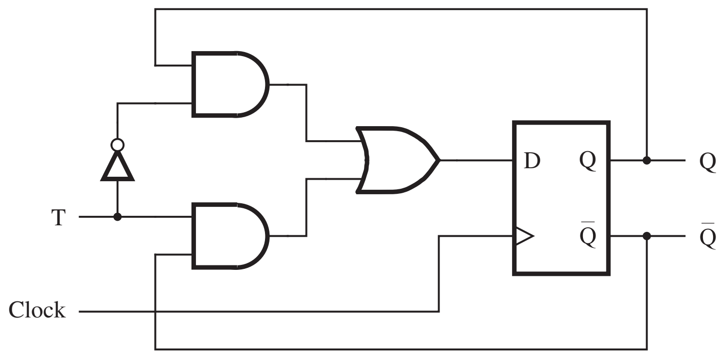
Figure 7.15 Flip-flop timing parameters.

process. In section 7.15 we provide some examples that illustrate the effects of flip-flop timing parameters on the operation of circuits.

7.5 T Flip-Flop

The D flip-flop is a versatile storage element that can be used for many purposes. By including some simple logic circuitry to drive its input, the D flip-flop may appear to be a different type of storage element. An interesting modification is presented in Figure 7.16*a*. This circuit uses a positive-edge-triggered D flip-flop. The *feedback* connections make the input signal D equal to either the value of Q or $\overline{\text{Q}}$ under the control of the signal that is labeled T. On each positive edge of the clock, the flip-flop may change its state Q(t). If $T = 0$, then $D = \text{Q}$ and the state will remain the same, that is, $\text{Q}(t+1) = \text{Q}(t)$. But if $T = 1$, then $D = \overline{\text{Q}}$ and the new state will be $\text{Q}(t+1) = \overline{\text{Q}}(t)$. Therefore, the overall operation of the circuit is that it retains its present state if $T = 0$, and it reverses its present state if $T = 1$.

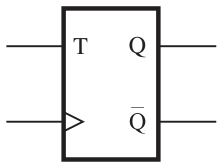
The operation of the circuit is specified in the form of a characteristic table in Figure 7.16*b*. Any circuit that implements this table is called a *T flip-flop*. The name T flip-flop



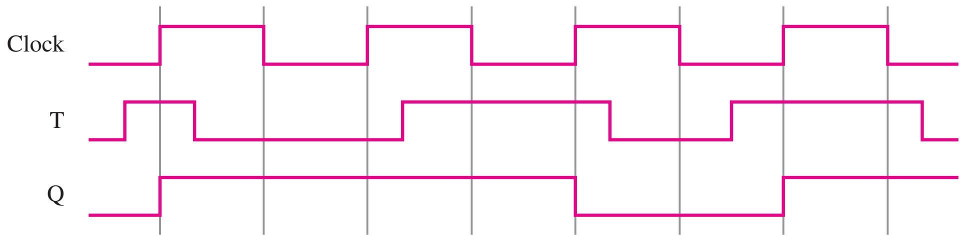
(a) Circuit

T	$Q(t+1)$
0	$Q(t)$
1	$\overline{Q}(t)$

(b) Characteristic table



(c) Graphical symbol



(d) Timing diagram

Figure 7.16 T flip-flop.

derives from the behavior of the circuit, which "toggles" its state when $T = 1$. The toggle feature makes the T flip-flop a useful element for building counter circuits, as we will see in section 7.9.

7.5.1 Configurable Flip-Flops

For some circuits one type of flip-flop may lead to a more efficient implementation than a different type of flip-flop. In general purpose chips like PLDs, the flip-flops that are provided are sometimes *configurable*, which means that a flip-flop circuit can be configured to be either D, T, or some other type. For example, in some PLDs the flip-flops can be configured as either D or T types (see problems 7.6 and 7.8).

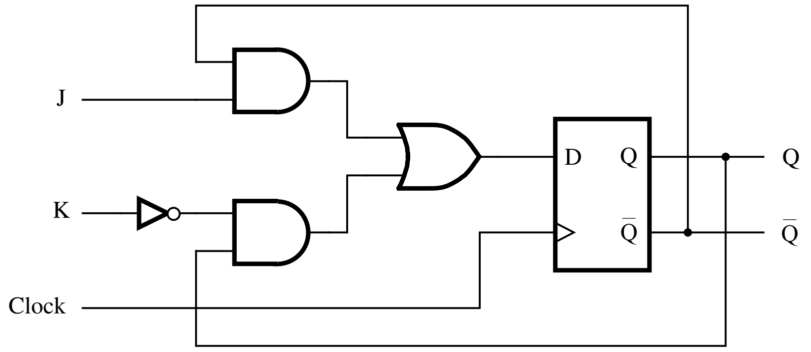
7.6 JK Flip-Flop

Another interesting circuit can be derived from Figure 7.16*a*. Instead of using a single control input, T, we can use two inputs, J and K, as indicated in Figure 7.17*a*. For this circuit the input D is defined as

$$D = J\overline{Q} + \overline{K}Q$$

A corresponding characteristic table is given in Figure 7.17*b*. The circuit is called a *JK flip-flop*. It combines the behaviors of SR and T flip-flops in a useful way. It behaves as the SR flip-flop, where $J = S$ and $K = R$, for all input values except $J = K = 1$. For the latter case, which has to be avoided in the SR flip-flop, the JK flip-flop toggles its state like the T flip-flop.

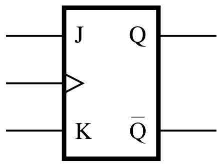
The JK flip-flop is a versatile circuit. It can be used for straight storage purposes, just like the D and SR flip-flops. But it can also serve as a T flip-flop by connecting the J and K inputs together.



(a) Circuit

J	K	Q($t+1$)
0	0	Q(t)
0	1	0
1	0	1
1	1	$\overline{Q}(t)$

(b) Characteristic table



(c) Graphical symbol

Figure 7.17 JK flip-flop.

7.7 Summary of Terminology

We have used the terminology that is quite common. But the reader should be aware that different interpretations of the terms *latch* and *flip-flop* can be found in the literature. Our terminology can be summarized as follows:

Basic latch is a feedback connection of two NOR gates or two NAND gates, which can store one bit of information. It can be set to 1 using the S input and reset to 0 using the R input.

Gated latch is a basic latch that includes input gating and a control input signal. The latch retains its existing state when the control input is equal to 0. Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock. We considered two types of gated latches:

- **Gated SR latch** uses the S and R inputs to set the latch to 1 or reset it to 0, respectively.
- **Gated D latch** uses the D input to force the latch into a state that has the same logic value as the D input.

A **flip-flop** is a storage element based on the gated latch principle, which can have its output state changed only on the edge of the controlling clock signal. We considered two types:

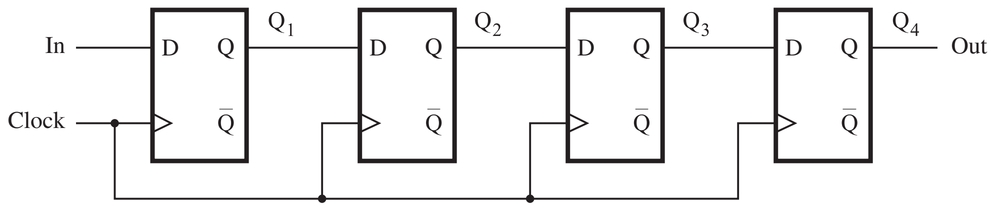
- **Edge-triggered flip-flop** is affected only by the input values present when the active edge of the clock occurs.
- **Master-slave flip-flop** is built with two gated latches. The master stage is active during half of the clock cycle, and the slave stage is active during the other half. The output value of the flip-flop changes on the edge of the clock that activates the transfer into the slave stage.

7.8 Registers

A flip-flop stores one bit of information. When a set of n flip-flops is used to store n bits of information, such as an n-bit number, we refer to these flip-flops as a *register*. A common clock is used for each flip-flop in a register, and each flip-flop operates as described in the previous sections. The term register is merely a convenience for referring to n-bit structures consisting of flip-flops.

7.8.1 Shift Register

In section 5.6 we explained that a given number is multiplied by 2 if its bits are shifted one bit position to the left and a 0 is inserted as the new least-significant bit. Similarly, the number is divided by 2 if the bits are shifted one bit-position to the right. A register that provides the ability to shift its contents is called a *shift register*.



(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

(b) A sample sequence

Figure 7.18 A simple shift register.

Figure 7.18*a* shows a four-bit shift register that is used to shift its contents one bit-position to the right. The data bits are loaded into the shift register in a serial fashion using the *In* input. The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock. An illustration of the transfer is given in Figure 7.18*b*, which shows what happens when the signal values at *In* during eight consecutive clock cycles are 1, 0, 1, 1, 1, 0, 0, and 0, assuming that the initial state of all flip-flops is 0.

To implement a shift register, it is necessary to use either edge-triggered or master-slave flip-flops. The level-sensitive gated latches are not suitable, because a change in the value of *In* would propagate through more than one latch during the time when the clock is equal to 1.

7.8.2 Parallel-Access Shift Register

In computer systems it is often necessary to transfer *n*-bit data items. This may be done by transmitting all bits at once using *n* separate wires, in which case we say that the transfer is performed in *parallel*. But it is also possible to transfer all bits using a single wire, by

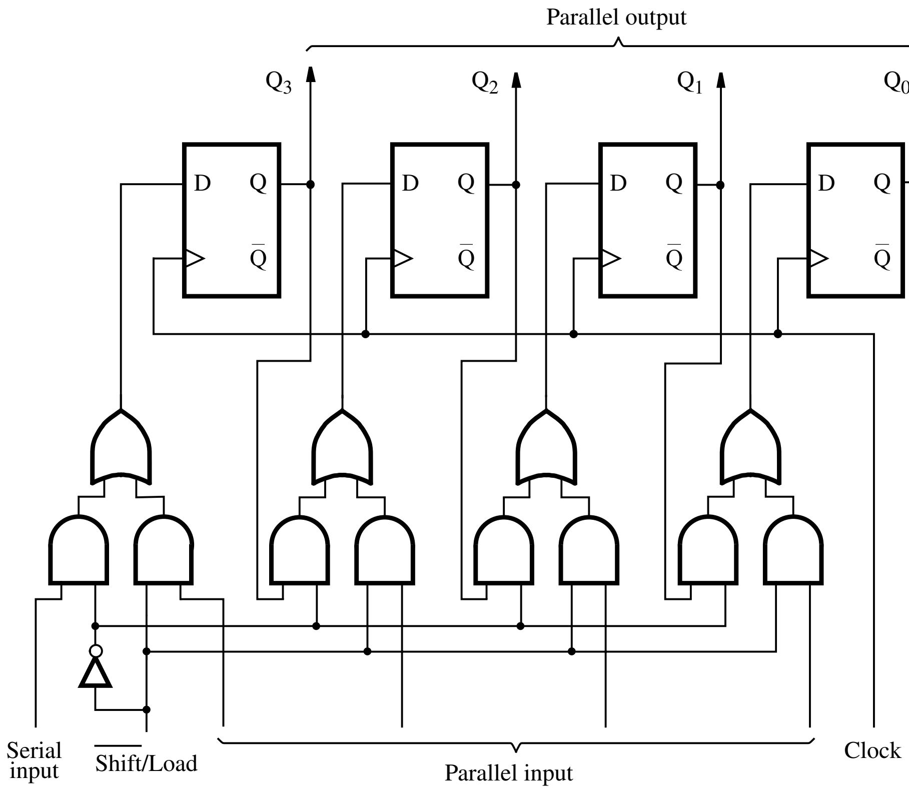


Figure 7.19 Parallel-access shift register.

performing the transfer one bit at a time, in n consecutive clock cycles. We refer to this scheme as *serial* transfer. To transfer an n-bit data item serially, we can use a shift register that can be loaded with all n bits in parallel (in one clock cycle). Then during the next n clock cycles, the contents of the register can be shifted out for serial transfer. The reverse operation is also needed. If bits are received serially, then after n clock cycles the contents of the register can be accessed in parallel as an n-bit item.

Figure 7.19 shows a four-bit shift register that allows the parallel access. Instead of using the normal shift register connection, the D input of each flip-flop is connected to two different sources. One source is the preceding flip-flop, which is needed for the shift-register operation. The other source is the external input that corresponds to the bit that is to be loaded into the flip-flop as a part of the parallel-load operation. The control signal $\overline{Shift}/Load$ is used to select the mode of operation. If $\overline{Shift}/Load = 0$, then the circuit operates as a shift register. If $\overline{Shift}/Load = 1$, then the parallel input data are loaded into the register. In both cases the action takes place on the positive edge of the clock.

In Figure 7.19 we have chosen to label the flip-flops outputs as $Q_3, \ldots, Q_0$ because shift registers are often used to hold binary numbers. The contents of the register can be accessed in parallel by observing the outputs of all flip-flops. The flip-flops can also be accessed serially, by observing the values of Q_0 during consecutive clock cycles while the contents are being shifted. A circuit in which data can be loaded in series and then accessed in parallel is called a series-to-parallel converter. Similarly, the opposite type of circuit is a parallel-to-series converter. The circuit in Figure 7.19 can perform both of these functions.

7.9 Counters

In Chapter 5 we dealt with circuits that perform arithmetic operations. We showed how adder/subtractor circuits can be designed, either using a simple cascaded (ripple-carry) structure that is inexpensive but slow or using a more complex carry-lookahead structure that is both more expensive and faster. In this section we examine special types of addition and subtraction operations, which are used for the purpose of counting. In particular, we want to design circuits that can increment or decrement a count by 1. Counter circuits are used in digital systems for many purposes. They may count the number of occurrences of certain events, generate timing intervals for control of various tasks in a system, keep track of time elapsed between specific events, and so on.

Counters can be implemented using the adder/subtractor circuits discussed in Chapter 5 and the registers discussed in section 7.8. However, since we only need to change the contents of a counter by 1, it is not necessary to use such elaborate circuits. Instead, we can use much simpler circuits that have a significantly lower cost. We will show how the counter circuits can be designed using T and D flip-flops.

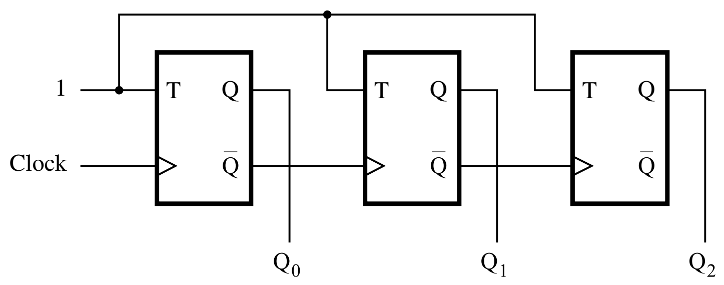
7.9.1 Asynchronous Counters

The simplest counter circuits can be built using T flip-flops because the toggle feature is naturally suited for the implementation of the counting operation.

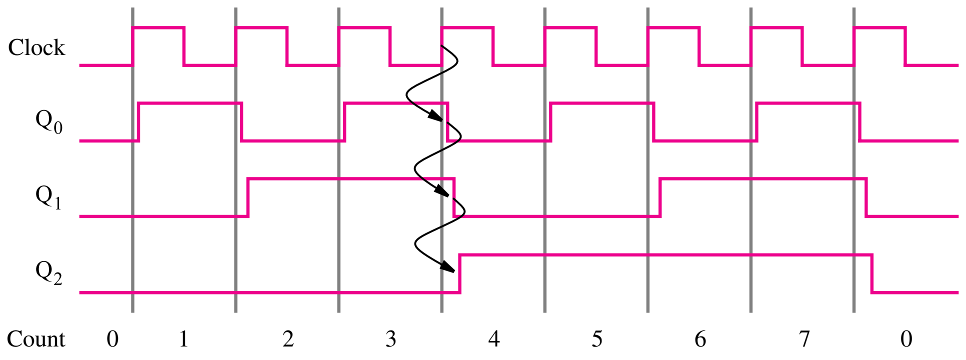
Up-Counter with T Flip-Flops

Figure 7.20*a* gives a three-bit counter capable of counting from 0 to 7. The clock inputs of the three flip-flops are connected in cascade. The *T* input of each flip-flop is connected to a constant 1, which means that the state of the flip-flop will be reversed (toggled) at each positive edge of its clock. We are assuming that the purpose of this circuit is to count the number of pulses that occur on the primary input called *Clock*. Thus the clock input of the first flip-flop is connected to the *Clock* line. The other two flip-flops have their clock inputs driven by the $\overline{Q}$ output of the preceding flip-flop. Therefore, they toggle their state whenever the preceding flip-flop changes its state from $Q = 1$ to $Q = 0$, which results in a positive edge of the $\overline{Q}$ signal.

Figure 7.20*b* shows a timing diagram for the counter. The value of Q_0 toggles once each clock cycle. The change takes place shortly after the positive edge of the *Clock* signal. The delay is caused by the propagation delay through the flip-flop. Since the second flip-flop is clocked by $\overline{Q}_0$, the value of Q_1 changes shortly after the negative edge of the Q_0 signal. Similarly, the value of Q_2 changes shortly after the negative edge of the Q_1 signal. If we look at the values $Q_2Q_1Q_0$ as the count, then the timing diagram indicates that the counting sequence is 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, and so on. This circuit is a modulo-8 counter. Because it counts in the upward direction, we call it an *up-counter*.



(a) Circuit



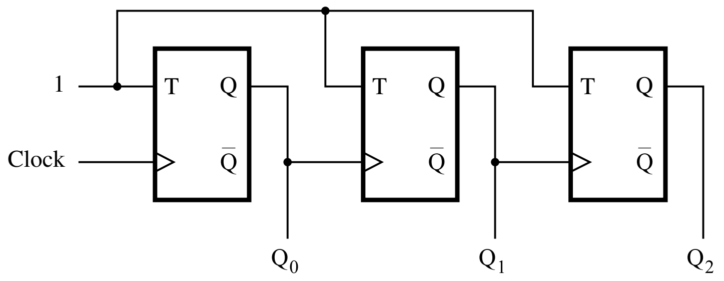
(b) Timing diagram

Figure 7.20 A three-bit up-counter.

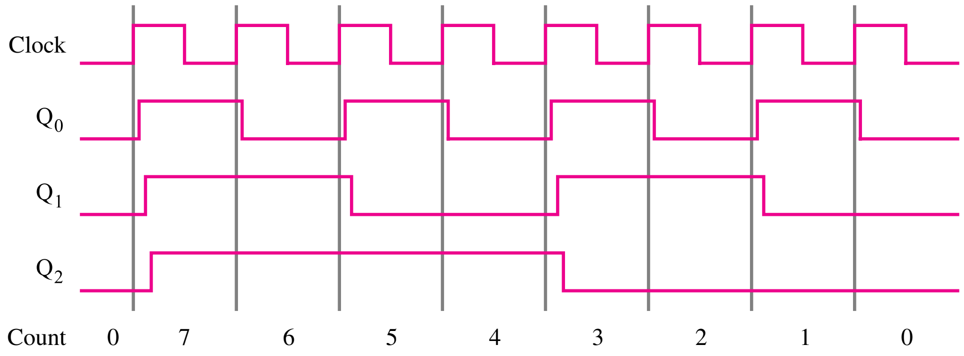
The counter in Figure 7.20*a* has three *stages*, each comprising a single flip-flop. Only the first stage responds directly to the *Clock* signal; we say that this stage is *synchronized* to the clock. The other two stages respond after an additional delay. For example, when *Count* = 3, the next clock pulse will cause the *Count* to go to 4. As indicated by the arrows in the timing diagram in Figure 7.20*b*, this change requires the toggling of the states of all three flip-flops. The change in Q_0 is observed only after a propagation delay from the positive edge of *Clock*. The Q_1 and Q_2 flip-flops have not yet changed; hence for a brief time the count is $Q_2Q_1Q_0 = 010$. The change in Q_1 appears after a second propagation delay, at which point the count is 000. Finally, the change in Q_2 occurs after a third delay, at which point the stable state of the circuit is reached and the count is 100. This behavior is similar to the rippling of carries in the ripple-carry adder circuit of Figure 5.6. The circuit in Figure 7.20*a* is an *asynchronous counter*, or a *ripple counter*.

Down-Counter with T Flip-Flops

A slight modification of the circuit in Figure 7.20*a* is presented in Figure 7.21*a*. The only difference is that in Figure 7.21*a* the clock inputs of the second and third flip-flops are driven by the Q outputs of the preceding stages, rather than by the $\overline{Q}$ outputs. The timing diagram, given in Figure 7.21*b*, shows that this circuit counts in the sequence 0, 7, 6, 5, 4,



(a) Circuit



(b) Timing diagram

Figure 7.21 A three-bit down-counter.

3, 2, 1, 0, 7, and so on. Because it counts in the downward direction, we say that it is a *down-counter*.

It is possible to combine the functionality of the circuits in Figures 7.20*a* and 7.21*a* to form a counter that can count either up or down. Such a counter is called an *up/down-counter*. We leave the derivation of this counter as an exercise for the reader (problem 7.16).

7.9.2 Synchronous Counters

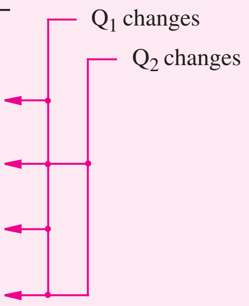
The asynchronous counters in Figures 7.20*a* and 7.21*a* are simple, but not very fast. If a counter with a larger number of bits is constructed in this manner, then the delays caused by the cascaded clocking scheme may become too long to meet the desired performance requirements. We can build a faster counter by clocking all flip-flops at the same time, using the approach described below.

Synchronous Counter with T Flip-Flops

Table 7.1 shows the contents of a three-bit up-counter for eight consecutive clock cycles, assuming that the count is initially 0. Observing the pattern of bits in each row of

Table 7.1 Derivation of the synchronous up-counter.

Clock cycle	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

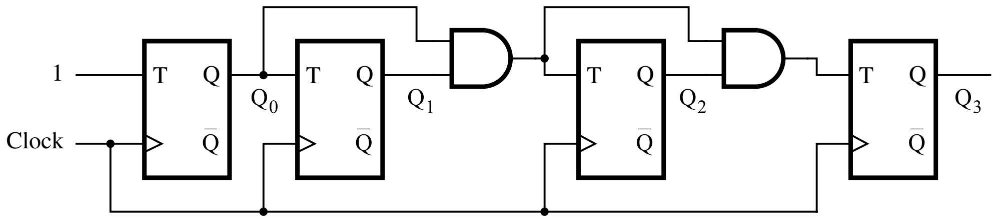


the table, it is apparent that bit Q_0 changes on each clock cycle. Bit Q_1 changes only when $Q_0 = 1$. Bit Q_2 changes only when both Q_1 and Q_0 are equal to 1. In general, for an n-bit up-counter, a given flip-flop changes its state only when all the preceding flip-flops are in the state $Q = 1$. Therefore, if we use T flip-flops to realize the counter, then the T inputs are defined as

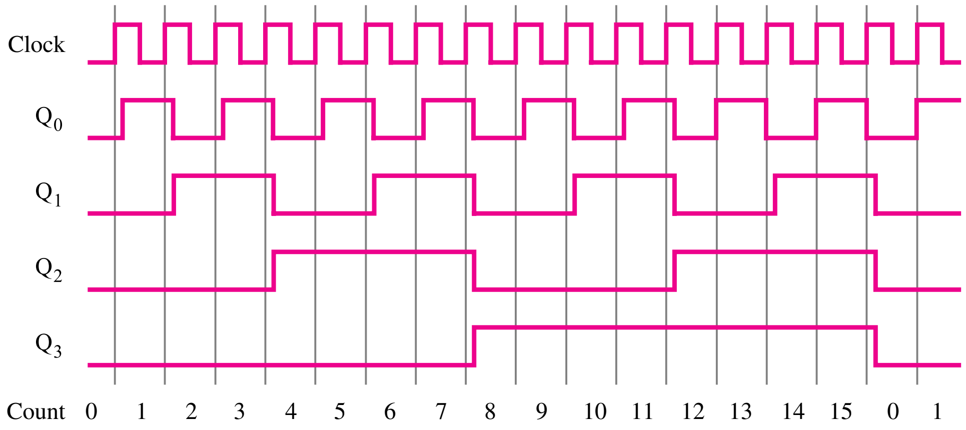
$$\begin{aligned}
T_0 &= 1 \\
T_1 &= Q_0 \\
T_2 &= Q_0Q_1 \\
T_3 &= Q_0Q_1Q_2 \\
&\cdot \\
&\cdot \\
&\cdot \\
T_n &= Q_0Q_1 \cdots Q_{n-1}
\end{aligned}$$

An example of a four-bit counter based on these expressions is given in Figure 7.22*a*. Instead of using AND gates of increased size for each stage, which may lead to fan-in problems, we use a factored arrangement, as shown in the figure. This arrangement does not slow down the response of the counter, because all flip-flops change their states after a propagation delay from the positive edge of the clock. Note that a change in the value of Q_0 may have to propagate through several AND gates to reach the flip-flops in the higher stages of the counter, which requires a certain amount of time. This time must not exceed the clock period. Actually, it must be less than the clock period minus the setup time for the flip-flops.

Figure 7.22*b* gives a timing diagram. It shows that the circuit behaves as a modulo-16 up-counter. Because all changes take place with the same delay after the active edge of the *Clock* signal, the circuit is called a *synchronous counter*.



(a) Circuit



(b) Timing diagram

Figure 7.22 A four-bit synchronous up-counter.

Enable and Clear Capability

The counters in Figures 7.20 through 7.22 change their contents in response to each clock pulse. Often it is desirable to be able to inhibit counting, so that the count remains in its present state. This may be accomplished by including an *Enable* control signal, as indicated in Figure 7.23. The circuit is the counter of Figure 7.22, where the *Enable* signal controls directly the T input of the first flip-flop. Connecting the *Enable* also to the AND-gate chain means that if $Enable = 0$, then all T inputs will be equal to 0. If $Enable = 1$, then the counter operates as explained previously.

In many applications it is necessary to start with the count equal to zero. This is easily achieved if the flip-flops can be cleared, as explained in section 7.4.3. The clear inputs on all flip-flops can be tied together and driven by a *Clear* control input.

Synchronous Counter with D Flip-Flops

While the toggle feature makes T flip-flops a natural choice for the implementation of counters, it is also possible to build counters using other types of flip-flops. The JK

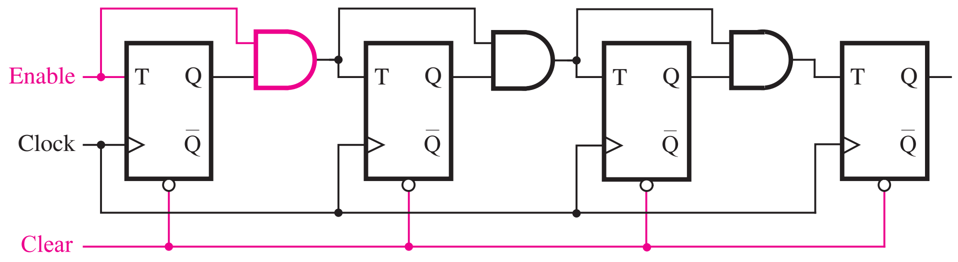


Figure 7.23 Inclusion of Enable and Clear capability.

flip-flops can be used in exactly the same way as the T flip-flops because if the J and K inputs are tied together, a JK flip-flop becomes a T flip-flop. We will now consider using D flip-flops for this purpose.

It is not obvious how D flip-flops can be used to implement a counter. We will present a formal method for deriving such circuits in Chapter 8. Here we will present a circuit structure that meets the requirements but will leave the derivation for Chapter 8. Figure 7.24 gives a four-bit up-counter that counts in the sequence 0, 1, 2, ..., 14, 15, 0, 1, and so on. The count is indicated by the flip-flop outputs $Q_3Q_2Q_1Q_0$. If we assume that $Enable = 1$, then the D inputs of the flip-flops are defined by the expressions

$$\begin{aligned} D_0 &= \overline{Q}_0 = 1 \oplus Q_0 \\ D_1 &= Q_1 \oplus Q_0 \\ D_2 &= Q_2 \oplus Q_1Q_0 \\ D_3 &= Q_3 \oplus Q_2Q_1Q_0 \end{aligned}$$

For a larger counter the ith stage is defined by

$$D_i = Q_i \oplus Q_{i-1}Q_{i-2} \cdots Q_1Q_0$$

We will show how to derive these equations in Chapter 8.

We have included the *Enable* control signal so that the counter counts the clock pulses only if $Enable = 1$. In effect, the above equations are modified to implement the circuit in the figure as follows

$$\begin{aligned} D_0 &= Q_0 \oplus Enable \\ D_1 &= Q_1 \oplus Q_0 \cdot Enable \\ D_2 &= Q_2 \oplus Q_1 \cdot Q_0 \cdot Enable \\ D_3 &= Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot Enable \end{aligned}$$

The operation of the counter is based on our observation for Table 7.1 that the state of the flip-flop in stage i changes only if all preceding flip-flops are in the state $Q = 1$. This makes the output of the AND gate that feeds stage i equal to 1, which causes the output of the XOR gate connected to D_i to be equal to $\overline{Q}_i$. Otherwise, the output of the XOR gate provides $D_i = Q_i$, and the flip-flop remains in the same state. This resembles the carry propagation in a carry-lookahead adder circuit (see section 5.4); hence the AND-gate chain

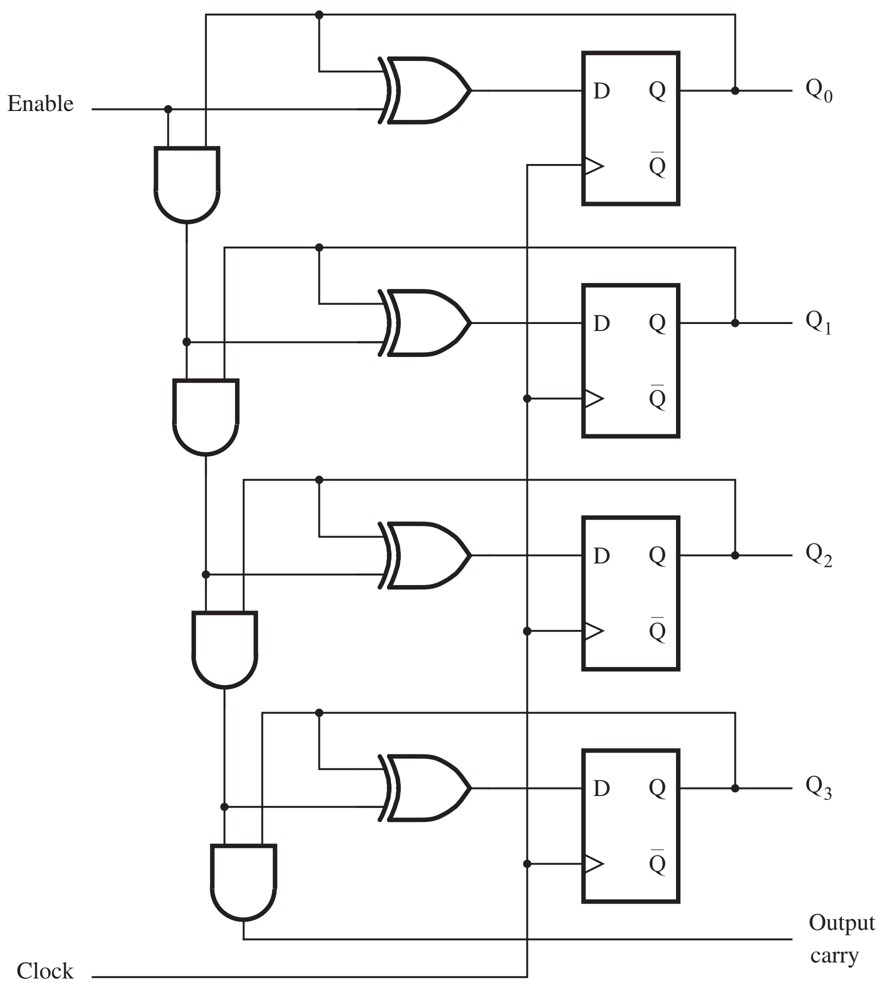


Figure 7.24 A four-bit counter with D flip-flops.

can be thought of as the *carry chain*. Even though the circuit is only a four-bit counter, we have included an extra AND gate that produces the "output carry." This signal makes it easy to concatenate two such four-bit counters to create an eight-bit counter.

Finally, the reader should note that the counter in Figure 7.24 is essentially the same as the circuit in Figure 7.23. We showed in Figure 7.16*a* that a T flip-flop can be formed from a D flip-flop by providing the extra gating that gives

$$\begin{aligned} D &= \mathrm{Q}\overline{T} + \overline{\mathrm{Q}}T \\ &= \mathrm{Q} \oplus T \end{aligned}$$

Thus in each stage in Figure 7.24, the D flip-flop and the associated XOR gate implement the functionality of a T flip-flop.

7.9.3 Counters with Parallel Load

Often it is necessary to start counting with the initial count being equal to 0. This state can be achieved by using the capability to clear the flip-flops as indicated in Figure 7.23. But sometimes it is desirable to start with a different count. To allow this mode of operation, a counter circuit must have some inputs through which the initial count can be loaded. Using the *Clear* and *Preset* inputs for this purpose is a possibility, but a better approach is discussed below.

The circuit of Figure 7.24 can be modified to provide the parallel-load capability as shown in Figure 7.25. A two-input multiplexer is inserted before each D input. One input to the multiplexer is used to provide the normal counting operation. The other input is a data bit that can be loaded directly into the flip-flop. A control input, *Load*, is used to choose the mode of operation. The circuit counts when $Load = 0$. A new initial value, $D_3D_2D_1D_0$, is loaded into the counter when $Load = 1$.

7.10 Reset Synchronization

We have already mentioned that it is important to be able to clear, or *reset*, the contents of a counter prior to commencing a counting operation. This can be done using the clear capability of the individual flip-flops. But we may also be interested in resetting the count to 0 during the normal counting process. An n-bit up-counter functions naturally as a modulo-2^n counter. Suppose that we wish to have a counter that counts modulo some base that is not a power of 2. For example, we may want to design a modulo-6 counter, for which the counting sequence is 0, 1, 2, 3, 4, 5, 0, 1, and so on.

The most straightforward approach is to recognize when the count reaches 5 and then reset the counter. An AND gate can be used to detect the occurrence of the count of 5. Actually, it is sufficient to ascertain that $Q_2 = Q_0 = 1$, which is true only for 5 in our desired counting sequence. A circuit based on this approach is given in Figure 7.26*a*. It uses a three-bit synchronous counter of the type depicted in Figure 7.25. The parallel-load feature of the counter is used to reset its contents when the count reaches 5. The resetting action takes place at the positive clock edge after the count has reached 5. It involves loading $D_2D_1D_0 = 000$ into the flip-flops. As seen in the timing diagram in Figure 7.26*b*, the desired counting sequence is achieved, with each value of the count being established for one full clock cycle. Because the counter is reset on the active edge of the clock, we say that this type of counter has a *synchronous reset*.

Consider now the possibility of using the clear feature of individual flip-flops, rather than the parallel-load approach. The circuit in Figure 7.27*a* illustrates one possibility. It uses the counter structure of Figure 7.22*a*. Since the clear inputs are active when low, a

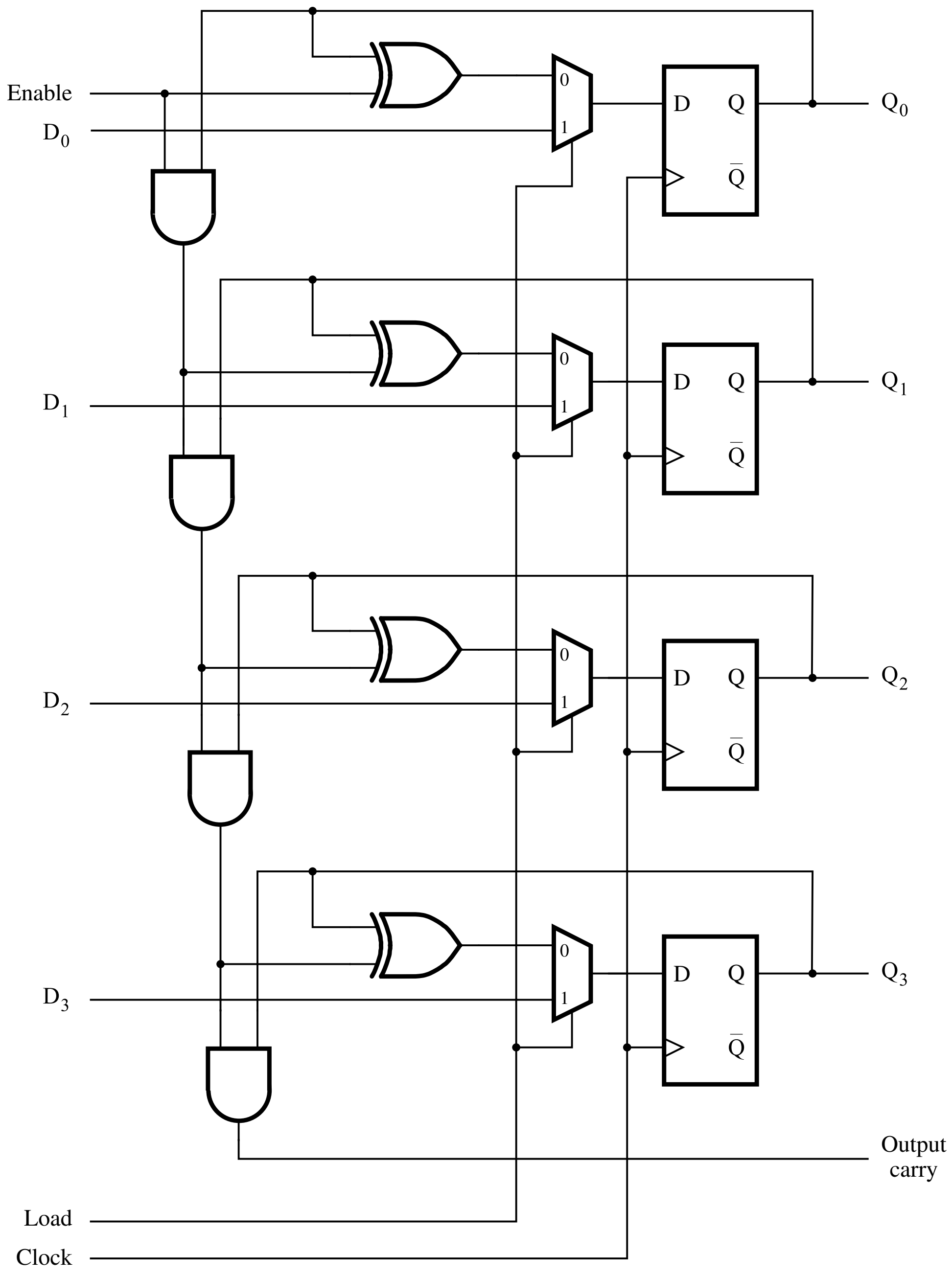
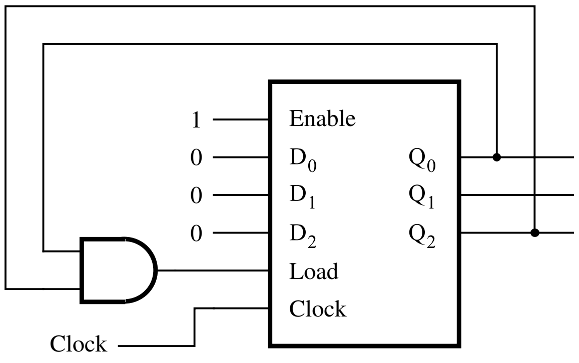
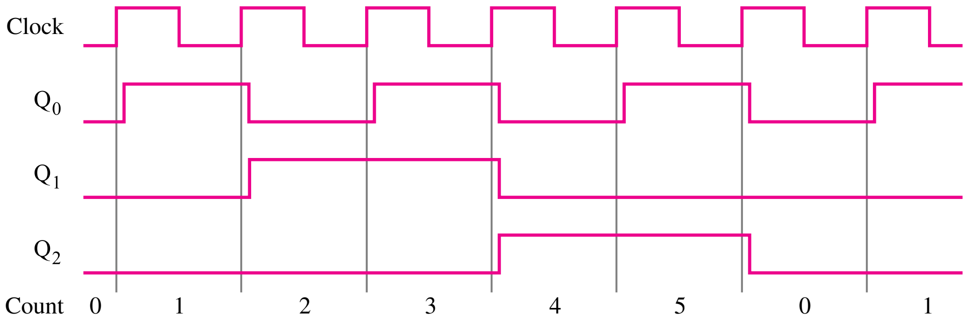


Figure 7.25 A counter with parallel-load capability.



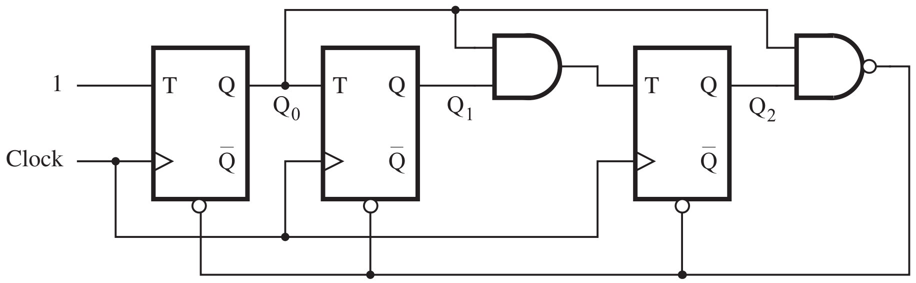
(a) Circuit



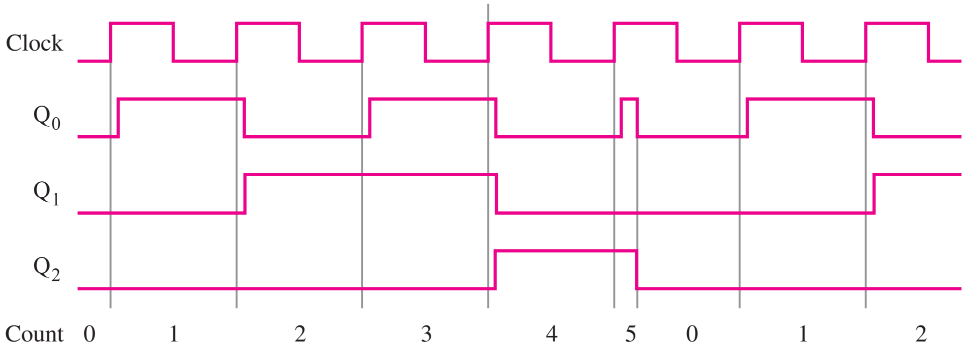
(b) Timing diagram

Figure 7.26 A modulo-6 counter with synchronous reset.

NAND gate is used to detect the occurrence of the count of 5 and cause the clearing of all three flip-flops. Conceptually, this seems to work fine, but closer examination reveals a potential problem. The timing diagram for this circuit is given in Figure 7.27*b*. It shows a difficulty that arises when the count is equal to 5. As soon as the count reaches this value, the NAND gate triggers the resetting action. The flip-flops are cleared to 0 a short time after the NAND gate has detected the count of 5. This time depends on the gate delays in the circuit, but not on the clock. Therefore, signal values $Q_2Q_1Q_0 = 101$ are maintained for a time that is much less than a clock cycle. Depending on a particular application of such a counter, this may be adequate, but it may also be completely unacceptable. For example, if the counter is used in a digital system where all operations in the system are synchronized by the same clock, then this narrow pulse denoting *Count* = 5 would not be seen by the



(a) Circuit



(b) Timing diagram

Figure 7.27 A modulo-6 counter with asynchronous reset.

rest of the system. To solve this problem, we could try to use a modulo-7 counter instead, assuming that the system would ignore the short pulse that denotes the count of 6. This is not a good way of designing circuits, because undesirable pulses often cause unforeseen difficulties in practice. The approach employed in Figure 7.27*a* is said to use *asynchronous reset*.

The timing diagrams in Figures 7.26*b* and 7.27*b* suggest that synchronous reset is a better choice than asynchronous reset. The same observation is true if the natural counting sequence has to be broken by loading some value other than zero. The new value of the count can be established cleanly using the parallel-load feature. The alternative of using the clear and preset capability of individual flip-flops to set their states to reflect the desired count has the same problems as discussed in conjunction with the asynchronous reset.

7.11 Other Types of Counters

In this section we discuss three other types of counters that can be found in practical applications. The first uses the decimal counting sequence, and the other two generate sequences of codes that do not represent binary numbers.

7.11.1 BCD Counter

Binary-coded-decimal (BCD) counters can be designed using the approach explained in section 7.10. A two-digit BCD counter is presented in Figure 7.28. It consists of two modulo-10 counters, one for each BCD digit, which we implemented using the parallel-load four-bit counter of Figure 7.25. Note that in a modulo-10 counter it is necessary to reset the four flip-flops after the count of 9 has been obtained. Thus the *Load* input to each

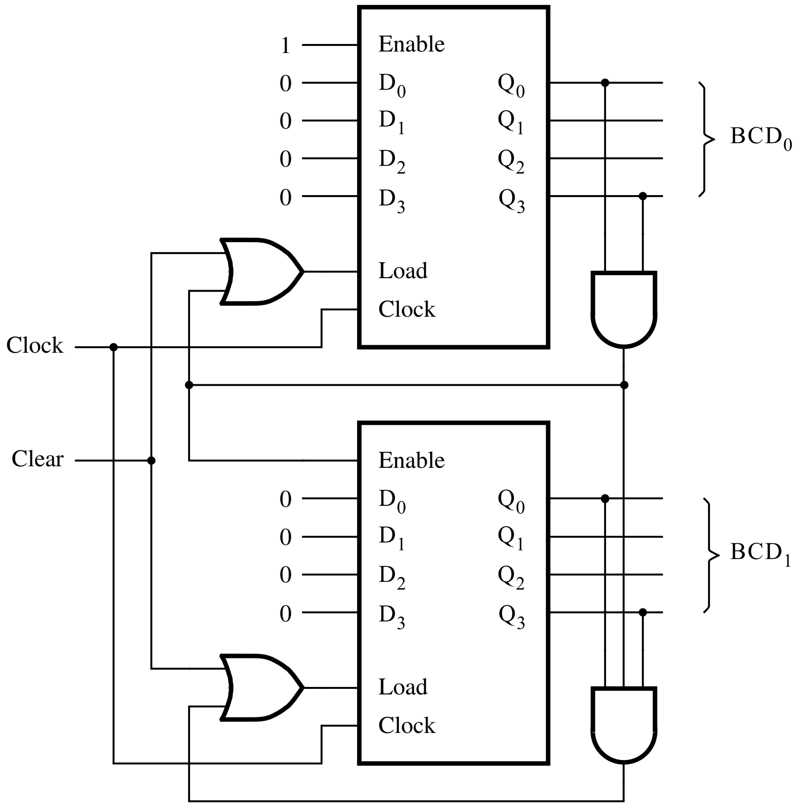


Figure 7.28 A two-digit BCD counter.

stage is equal to 1 when $Q_3 = Q_0 = 1$, which causes 0s to be loaded into the flip-flops at the next positive edge of the clock signal. Whenever the count in stage 0, BCD_0, reaches 9 it is necessary to enable the second stage so that it will be incremented when the next clock pulse arrives. This is accomplished by keeping the *Enable* signal for BCD_1 low at all times except when $BCD_0 = 9$.

In practice, it has to be possible to clear the contents of the counter by activating some control signal. Two OR gates are included in the circuit for this purpose. The control input *Clear* can be used to load 0s into the counter. Observe that in this case *Clear* is active when high. VHDL code for a two-digit BCD counter is given in Figure 7.77.

In any digital system there is usually one or more clock signals used to drive all synchronous circuitry. In the preceding counter, as well as in all counters presented in the previous figures, we have assumed that the objective is to count the number of clock pulses. Of course, these counters can be used to count the number of pulses in any signal that may be used in place of the clock signal.

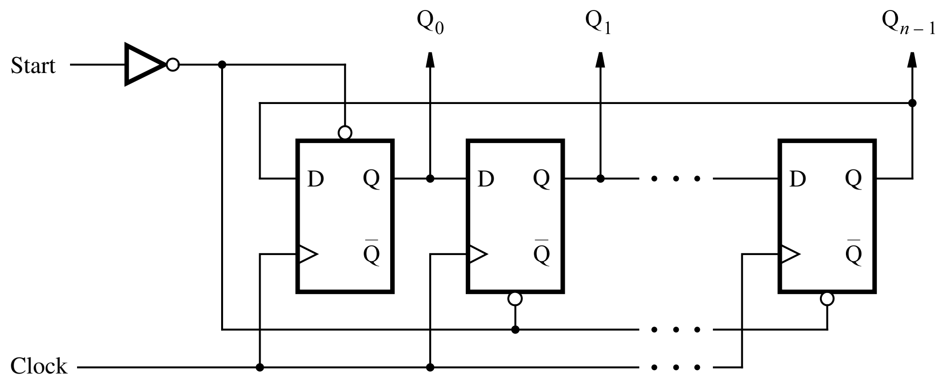
7.11.2 Ring Counter

In the preceding counters the count is indicated by the state of the flip-flops in the counter. In all cases the count is a binary number. Using such counters, if an action is to be taken as a result of a particular count, then it is necessary to detect the occurrence of this count. This may be done using AND gates, as illustrated in Figures 7.26 through 7.28.

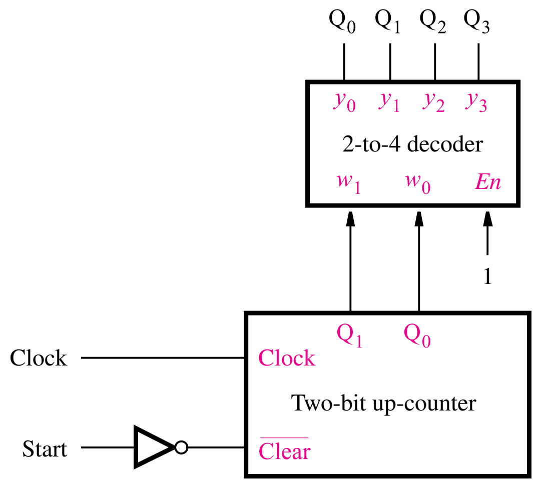
It is possible to devise a counterlike circuit in which each flip-flop reaches the state $Q_i = 1$ for exactly one count, while for all other counts $Q_i = 0$. Then Q_i indicates directly an occurrence of the corresponding count. Actually, since this does not represent binary numbers, it is better to say that the outputs of the flips-flops represent a code. Such a circuit can be constructed from a simple shift register, as indicated in Figure 7.29*a*. The Q output of the last stage in the shift register is fed back as the input to the first stage, which creates a ringlike structure. If a single 1 is injected into the ring, this 1 will be shifted through the ring at successive clock cycles. For example, in a four-bit structure, the possible codes $Q_0Q_1Q_2Q_3$ will be 1000, 0100, 0010, and 0001. As we said in section 6.2, such encoding, where there is a single 1 and the rest of the code variables are 0, is called a *one-hot code*.

The circuit in Figure 7.29*a* is referred to as a *ring counter*. Its operation has to be initialized by injecting a 1 into the first stage. This is achieved by using the *Start* control signal, which presets the left-most flip-flop to 1 and clears the others to 0. We assume that all changes in the value of the *Start* signal occur shortly after an active clock edge so that the flip-flop timing parameters are not violated.

The circuit in Figure 7.29*a* can be used to build a ring counter with any number of bits, n. For the specific case of $n = 4$, part (*b*) of the figure shows how a ring counter can be constructed using a two-bit up-counter and a decoder. When *Start* is set to 1, the counter is reset to 00. After *Start* changes back to 0, the counter increments its value in the normal way. The 2-to-4 decoder, described in section 6.2, changes the counter output into a one-hot code. For the count values 00, 01, 10, 11, 00, and so on, the decoder produces $Q_0Q_1Q_2Q_3 = 1000, 0100, 0010, 0001, 1000$, and so on. This circuit structure can be used for larger ring counters, as long as the number of bits is a power of two. We will give an example of a larger circuit that uses the ring counter in Figure 7.29*b* as a subcircuit in section 7.14.



(a) An *n*-bit ring counter



(b) A four-bit ring counter

Figure 7.29 Ring counter.

7.11.3 Johnson Counter

An interesting variation of the ring counter is obtained if, instead of the Q output, we take the $\overline{Q}$ output of the last stage and feed it back to the first stage, as shown in Figure 7.30. This circuit is known as a *Johnson counter*. An *n*-bit counter of this type generates a counting sequence of length 2*n*. For example, a four-bit counter produces the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, and so on. Note that in this sequence, only a single bit has a different value for two consecutive codes.

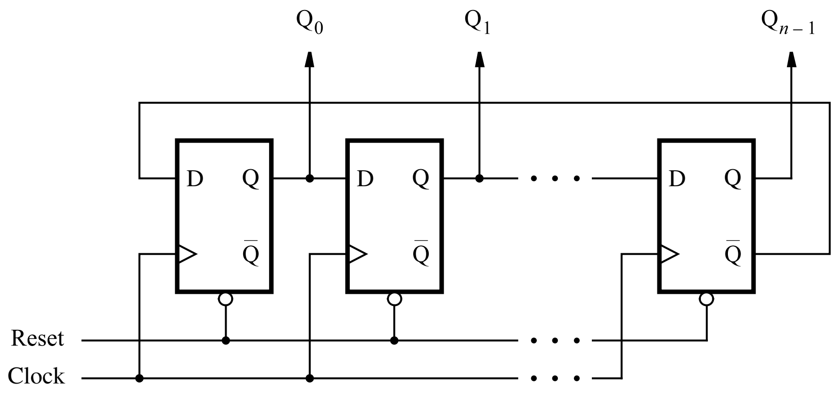


Figure 7.30 Johnson counter.

To initialize the operation of the Johnson counter, it is necessary to reset all flip-flops, as shown in the figure. Observe that neither the Johnson nor the ring counter will generate the desired counting sequence if not initialized properly.

7.11.4 Remarks on Counter Design

The sequential circuits presented in this chapter, namely, registers and counters, have a regular structure that allows the circuits to be designed using an intuitive approach. In Chapter 8 we will present a more formal approach to design of sequential circuits and show how the circuits presented in this chapter can be derived using this approach.

7.12 Using Storage Elements with CAD Tools

This section shows how circuits with storage elements can be designed using either schematic capture or VHDL code.

7.12.1 Including Storage Elements in Schematics

One way to create a circuit is to draw a schematic that builds latches and flip-flops from logic gates. Because these storage elements are used in many applications, most CAD systems provide them as prebuilt modules. Figure 7.31 shows a schematic created with a schematic capture tool, which includes three types of flip-flops that are imported from a library provided as part of the CAD system. The top element is a gated D latch, the middle element is a positive-edge-triggered D flip-flop, and the bottom one is a positive-edge-triggered T flip-flop. The D and T flip-flops have asynchronous, active-low clear and

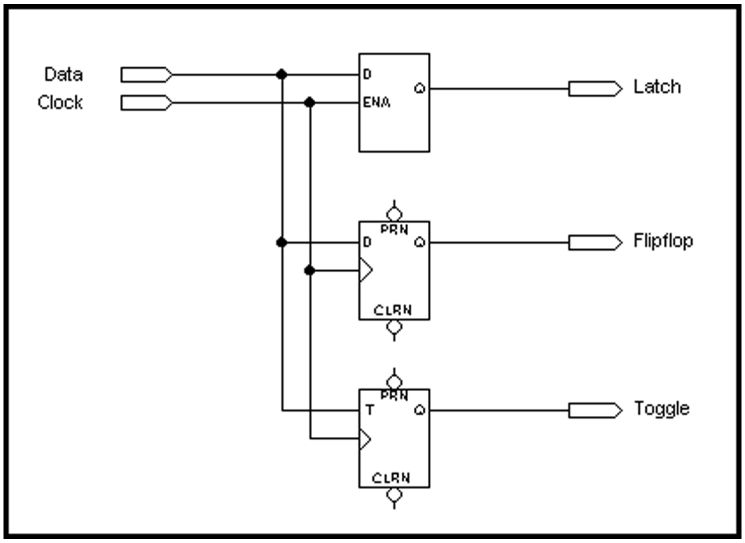


Figure 7.31 Three types of storage elements in a schematic.

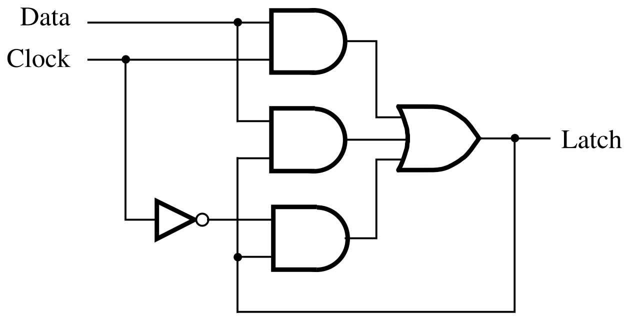


Figure 7.32 Gated D latch generated by CAD tools.

preset inputs. If these inputs are not connected in a schematic, then the CAD tool makes them inactive by assigning the default value of 1 to them.

When the gated D latch is synthesized for implementation in a chip, the CAD tool may not generate the cross-coupled NOR or NAND gates shown in section 7.2. In some chips, such as a CPLD, the AND-OR circuit depicted in Figure 7.32 may be preferable. This circuit is functionally equivalent to the cross-coupled version in section 7.2. The sum-of-products circuit is used because it is more suitable for implementation in a CPLD macrocell. One aspect of this circuit should be mentioned. From the functional point of view, it appears that the circuit can be simplified by removing the AND gate with the inputs *Data* and *Latch*.

Without this gate, the top AND gate sets the value stored in the latch when the clock is 1, and the bottom AND gate maintains the stored value when the clock is 0. But without this gate, the circuit has a timing problem known as a *static hazard.* A detailed explanation of hazards will be given in section 9.6.

The circuit in Figure 7.31 can be implemented in a CPLD as shown in Figure 7.33. The D and T flip-flops are realized using the flip-flops on the chip that are configurable as

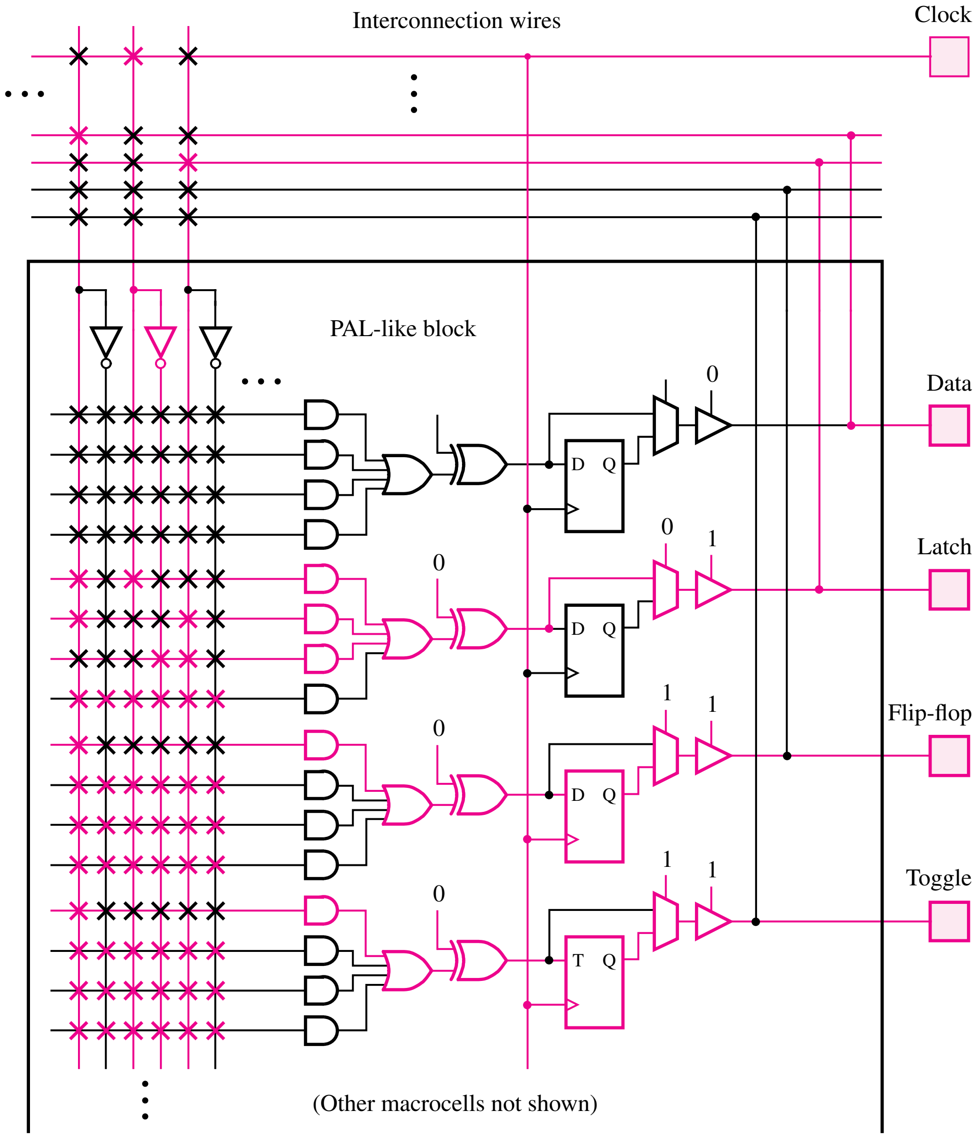


Figure 7.33 Implementation of the schematic in Figure 7.31 in a CPLD.

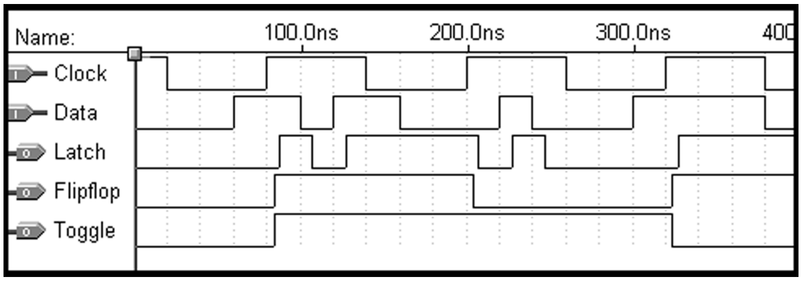


Figure 7.34 Timing simulation for the storage elements in Figure 7.31.

either D or T types. The figure depicts in blue the gates and wires needed to implement the circuit in Figure 7.31.

The results of a timing simulation for the implementation in Figure 7.33 are given in Figure 7.34. The *Latch* signal, which is the output of the gated D latch, implemented as indicated in Figure 7.32, follows the *Data* input whenever the *Clock* signal is 1. Because of propagation delays in the chip, the *Latch* signal is delayed in time with respect to the *Data* signal. Since the *Flipflop* signal is the output of the D flip-flop, it changes only after a positive clock edge. Similarly, the output of the T flip-flop, called *Toggle* in the figure, toggles when $Data = 1$ and a positive clock edge occurs. The timing diagram illustrates the delay from when the positive clock edge occurs at the input pin of the chip until a change in the flip-flop output appears at the output pin of the chip. This time is called the *clock-to-output time*, t_{co}.

7.12.2 Using VHDL Constructs for Storage Elements

In section 6.6 we described a number of VHDL assignment statements. The IF and CASE statements were introduced as two types of sequential assignment statements. In this section we show how these statements can be used to describe storage elements.

Figure 6.43, which is repeated in Figure 7.35, gives an example of VHDL code that has implied memory. Because the code does not specify what value the *AeqB* signal should have when the condition for the IF statement is not satisfied, the semantics specify that in this case *AeqB* should retain its current value. The implied memory is the key concept used for describing sequential circuit elements, which we will illustrate using several examples.

Example 7.1

CODE FOR A GATED D LATCH The code in Figure 7.36 defines an entity named *latch*, which has the inputs *D* and *Clk* and the output Q. The process uses an if-then-else statement to define the value of the Q output. When $Clk = 1$, Q takes the value of *D*. For the case when *Clk* is not 1, the code does not specify what value Q should have. Hence Q will retain its current value in this case, and the code describes a gated D latch. The process sensitivity list includes both *Clk* and *D* because these signals can cause a change in the value of the Q output.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
   PORT ( A, B   : IN    STD_LOGIC ;
          AeqB : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
   PROCESS ( A, B )
   BEGIN
      IF A = B THEN
         AeqB <= '1' ;
      END IF ;
   END PROCESS ;
END Behavior ;
```

Figure 7.35 The code from Figure 6.43, illustrating implied memory.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
   PORT ( D, Clk  : IN    STD_LOGIC ;
          Q        : OUT  STD_LOGIC) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
   PROCESS ( D, Clk )
   BEGIN
      IF Clk = '1' THEN
         Q <= D ;
      END IF ;
   END PROCESS ;
END Behavior ;
```

Figure 7.36 Code for a gated D latch.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock   : IN    STD_LOGIC ;
                Q     : OUT  STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.37 Code for a D flip-flop.

Example 7.2

CODE FOR A D FLIP-FLOP Figure 7.37 defines an entity named *flipflop*, which is a positive-edge-triggered D flip-flop. The code is identical to Figure 7.36 with two exceptions. First, the process sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output. Second, the if-then-else statement uses a different condition from the one used in the latch. The syntax Clock'EVENT uses a VHDL construct called an *attribute*. An attribute refers to a property of an object, such as a signal. In this case the 'EVENT attribute refers to any change in the *Clock* signal. Combining the Clock'EVENT condition with the condition $Clock = 1$ means that "the value of the $Clock$ signal has just changed, and the value is now equal to 1." Hence the condition refers to a positive clock edge. Because the Q output changes only as a result of a positive clock edge, the code describes a positive-edge-triggered D flip-flop.

Example 7.3

ALTERNATIVE CODE FOR A D FLIP-FLOP The process in Figure 7.38 uses a different syntax from that in Figure 7.37 to describe a D flip-flop. It uses the statement WAIT UNTIL Clock'EVENT AND Clock = '1'. This statement has the same effect as the IF statement in Figure 7.37. A process that uses a WAIT UNTIL statement is a special case because the sensitivity list is omitted. The WAIT UNTIL construct implies that the sensitivity list includes only the clock signal. In our use of VHDL, which is for synthesis of circuits, a process can use a WAIT UNTIL statement only if this is the first statement in the process.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
    PORT ( D, Clock  : IN    STD_LOGIC ;
           Q         : OUT   STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        Q <= D ;
    END PROCESS ;
END Behavior ;
```

Figure 7.38 Equivalent code to Figure 7.37, using a WAIT UNTIL statement.

Actually, the attribute 'EVENT is redundant in the WAIT UNTIL statement. We can write simply

```
WAIT UNTIL Clock = '1';
```

which also implies that the action occurs when the *Clock* signal becomes equal to 1, namely, at the edge when the signal changes from 0 to 1. However, some CAD synthesis tools require the inclusion of the 'EVENT attribute, which is the reason why we use this style in the book.

In general, whenever it is desired to include in VHDL code flip-flops that are clocked by the positive clock edge, the condition Clock'EVENT AND Clock '1' is used. When this condition appears in an IF statement, any signals that are assigned values inside the IF statement are implemented as the outputs of flip-flops. When the condition is used in a WAIT UNTIL statement, any signal that is assigned a value in the entire process is implemented as the output of a flip-flop.

The differences in using the IF and WAIT UNTIL statements are discussed in more detail in Appendix A, section A.10.3.

Example 7.4 **ASYNCHRONOUS CLEAR** Figure 7.39 gives a process that is similar to the one in Figure 7.37. It describes a D flip-flop with an asynchronous active-low reset (clear) input. When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0.

Example 7.5 **SYNCHRONOUS CLEAR** Figure 7.40 shows how a D flip-flop with a synchronous reset input can be described. In this case the reset signal is acted upon only when a positive clock edge arrives. The code generates the circuit in Figure 7.14*c*, which has an AND gate connected to the flip-flop's D input.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN    STD_LOGIC ;
           Q                 : OUT   STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.39 D flip-flop with asynchronous reset.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN    STD_LOGIC ;
           Q                 : OUT   STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.40 D flip-flop with synchronous reset.

Figure A.33*a* in Appendix A shows how the same circuit is specified by using an IF statement instead of WAIT UNTIL.

7.13 USING REGISTERS AND COUNTERS WITH CAD TOOLS

In this section we show how registers and counters can be included in circuits designed with the aid of CAD tools. Examples are given using both schematic capture and VHDL code.

7.13.1 INCLUDING REGISTERS AND COUNTERS IN SCHEMATICS

In section 5.5.1 we explained that a CAD system usually includes libraries of prebuilt subcircuits. We introduced the library of parameterized modules (LPM) and used the adder/subtractor module, *lpm_add_sub*, as an example. The LPM includes modules that constitute flip-flops, registers, counters, and many other useful circuits. Figure 7.41 shows a symbol that represents the *lpm_ff* module. This module is a register with one or more positive-edge-triggered flip-flops that can be of either D or T type. The module has parameters that allow the number of flip-flops and flip-flop type to be chosen. In this case we chose to have four D flip-flops. The tutorial in Appendix C explains how the configuration of LPM modules is done.

The D inputs to the four flip-flops, called *data* on the graphical symbol, are connected to the four-bit input signal *Data*[3..0]. The module's asynchronous active-high reset (clear) input, *aclr*, is shown in the schematic. The flip-flop outputs, *q*, are attached to the output symbol labeled Q[3..0].

In section 7.3 we said that a useful application of D flip-flops is to hold the results of an arithmetic computation, such as the output from an adder circuit. An example is given in Figure 7.42, which uses two LPM modules, *lpm_add_sub* and *lpm_ff*. The *lpm_add_sub* module was described in section 5.5.1. Its parameters, which are not shown in Figure 7.42,

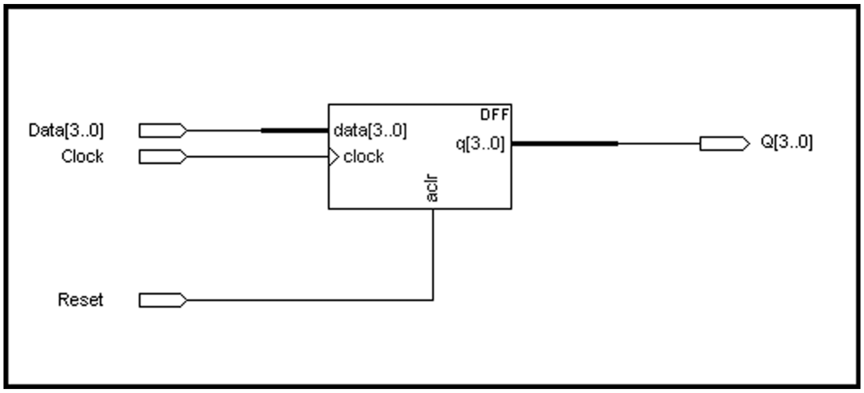


Figure 7.41 The *lpm_ff* parameterized flip-flop module.

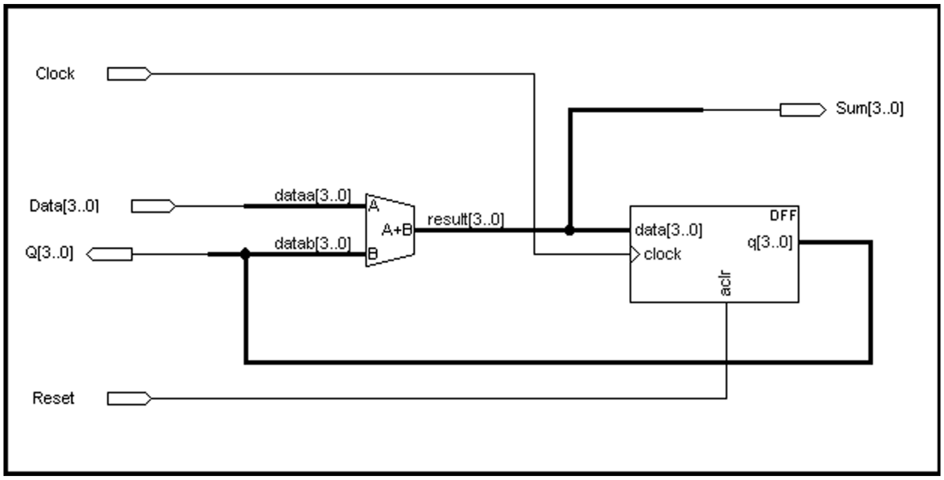


Figure 7.42 An adder with registered feedback.

are set to configure the module as a four-bit adder circuit. The adder's four-bit data input *dataa* is driven by the *Data*[3..0] input signal. The sum bits, *result*, are connected to the *data* inputs of the *lpm_ff*, which is configured as a four-bit D register with asynchronous clear. The register generates the output of the circuit, Q[3..0], which appears on the left side of the schematic. This signal is fed back to the *datab* input of the adder. The sum bits from the adder are also provided as an output of the circuit, *Sum*[3..0], for ease of reference in the discussion that follows. If the register is first cleared to 0000, then the circuit can be used to add the binary numbers on the *Data*[3..0] input to a sum that is being accumulated in the register, if a new number is applied to the input during each clock cycle. A circuit that performs this function is referred to as an *accumulator* circuit.

We synthesized a circuit from the schematic and implemented the four-bit adder using the carry-lookahead structure. A timing simulation for the circuit appears in Figure 7.43. After resetting the circuit, the *Data* input is set to 0001. The adder produces the sum $0000 + 0001 = 0001$, which is then clocked into the register at the 60 ns point in time. After the t_{co} delay, Q[3..0] becomes 0001, and this causes the adder to produce the new sum $0001 + 0001 = 0010$. The time needed to generate the new sum is determined by the speed of the adder circuit, which produces the sum after 12.5 ns in this case. The new sum does not appear at the Q output until after the next positive clock edge, at 100 ns. The adder then produces 0011 as the next sum. When *Sum* changes from 0010 to 0011, some oscillations appear in the timing diagram, caused by the propagation of carry signals through the adder circuit. These oscillations are not seen at the Q output, because *Sum* is stable by the time the next positive clock edge occurs. Moving forward to the 180 ns point in time, $Sum = 0100$, and this value is clocked into the register. The adder produces the new sum 0101. Then at 200 ns *Data* is changed to 0010, which causes the sum to change to $0100 + 0010 = 0110$. At the next positive clock edge, Q is set to 0110; the value $Sum = 0101$ that was present temporarily in the circuit is not observed at the Q output. The circuit continues to add 0010 to the Q output at each successive positive clock edge.

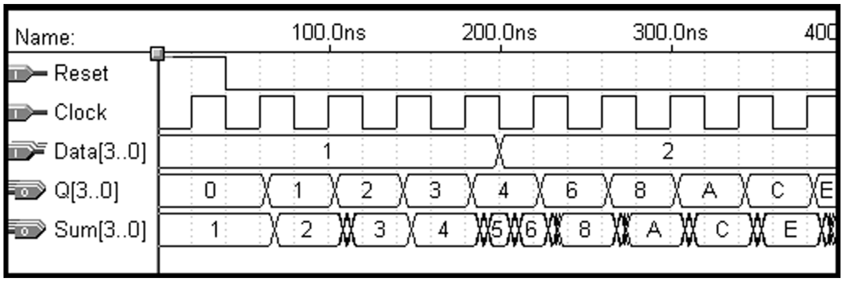


Figure 7.43 Timing simulation of the circuit from Figure 7.42.

Having simulated the behavior of the circuit, we should consider whether or not we can conclude with some certainty that the circuit works properly. Ideally, it is prudent to test all possible combinations of a circuit's inputs before declaring that it works as desired. However, in practice such testing is often not feasible because of the number of input combinations that exist. For the circuit in Figure 7.42, we could verify that a correct sum is produced by the adder, and we could also check that each of the four flip-flops in the register properly stores either 0 or 1. We will discuss issues associated with the testing of circuits in Chapter 11.

For the circuit in Figure 7.42 to work properly, the following timing constraints must be met. When the register is clocked by a positive clock edge, a change of signal value at the register's output must propagate through the feedback path to the *datab* input of the adder. The adder then produces a new sum, which must propagate to the *data* input of the register. For the chip used to implement the circuit, the total delay incurred is 14 ns. The delay can be broken down as follows: It takes 2 ns from when the register is clocked until a change in its output reaches the *datab* input of the adder. The adder produces a new sum in 8 ns, and it takes 4 ns for the sum to propagate to the register's *data* input. In Figure 7.43 the clock period is 40 ns. Hence after the new sum arrives at the *data* input of the register, there remain $40 - 14 = 26$ ns until the next positive clock edge occurs. The *data* input must be stable for the amount of the setup time, $t_{su} = 3$ ns, before the clock edge. Hence we have $26 - 3 = 23$ ns to spare. The clock period can be decreased by as much as 23 ns, and the circuit will still work. But if the clock period is less than $40 - 23 = 17$ ns, then the circuit will not function properly. Of course, if a different chip were used to implement the circuit, then different timing results would be produced. CAD systems provide tools that can automatically determine the minimum allowable clock period for which a circuit will work correctly. The tutorial in Appendix C shows how this is done using the tools that accompany the book.

7.13.2 Registers and Counters in VHDL Code

The predefined subcircuits in the LPM library can be instantiated in VHDL code. Figure 7.44 instantiates the *lpm_shiftreg* module, which is an *n*-bit shift register. The module's

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY lpm ;
USE lpm.lpm_components.all ;

ENTITY shift IS
    PORT ( Clock          : IN    STD_LOGIC ;
           Reset          : IN    STD_LOGIC ;
           Shiftin, Load  : IN    STD_LOGIC ;
           R              : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Q              : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END shift ;

ARCHITECTURE Structure OF shift IS
BEGIN
    instance: lpm_shiftreg
        GENERIC MAP (LPM_WIDTH => 4, LPM_DIRECTION => "RIGHT")
        PORT MAP (data => R, clock => Clock, aclr => Reset,
            load => Load, shiftin => Shiftin, q => Q ) ;
END Structure ;
```

Figure 7.44 Instantiation of the *lpm_shiftreg* module.

parameters are set using the GENERIC MAP construct, as shown. The GENERIC MAP construct is similar to the PORT MAP construct that is used to assign signal names to the ports of a subcircuit. GENERIC MAP is used to assign values to the parameters of the subcircuit. The number of flip-flops in the shift register is set to 4 using the parameter LPM_WIDTH => 4. The module can be configured to shift either left or right. The parameter LPM_DIRECTION => RIGHT sets the shift direction to be from the left to the right. The code uses the module's asynchronous active-high clear input, *aclr*, and the active-high parallel-load input, *load*, which allows the shift register to be loaded with the parallel data on the module's *data* input. When shifting takes place, the value on the *shiftin* input is shifted into the left-most flip-flop and the bit shifted out appears on the right-most bit of the *q* parallel output. The code uses the named association, described in section 5.5.2, to connect the input and output signals of the *shift* entity to the ports of the module. For example, the *R* input signal is connected to the module's *data* port. When translated into a circuit, the *lpm_shiftreg* has the structure shown in Figure 7.19.

Predefined modules also exist for various types of counters, which are commonly needed in logic circuits. An example is the *lpm_counter* module, which is a variable-width counter with parallel-load inputs.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg8 IS
    PORT ( D              : IN   STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg8 ;

ARCHITECTURE Behavior OF reg8 IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= "00000000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.45 Code for an eight-bit register with asynchronous clear.

7.13.3 Using VHDL Sequential Statements for Registers and Counters

Rather than instantiating predefined subcircuits for registers, shift registers, counters, and the like, the circuits can be described in VHDL using sequential statements. Figure 7.39 gives code for a D flip-flop. A straightforward way to describe an *n*-bit register is to write hierarchical code that includes *n* instances of the D flip-flop subcircuit. A simpler approach is shown in Figure 7.45. It uses the same code as in Figure 7.39 except that the D input and Q output are defined as multibit signals. The code represents an eight-bit register with asynchronous clear.

Example 7.6 **AN *N*-BIT REGISTER** Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register entity for which the number of flip-flops can be easily changed. Figure 7.46 shows how the code in Figure 7.45 can be extended to include a parameter that sets the number of flip-flops. The parameter is an integer, N, which is defined using the VHDL construct called GENERIC. The value of N is set to 16 using the := assignment operator. By changing this parameter, the code can represent a register of any size. If the register is declared as a component, then it can be used as a subcircuit in other code. That code can either use the default value of the GENERIC parameter or else specify a different parameter using the GENERIC MAP construct. An example showing how GENERIC MAP is used is shown in Figure 7.44.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 16 ) ;
    PORT ( D              : IN    STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
           Resetn, Clock  : IN    STD_LOGIC ;
           Q              : OUT   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.46 Code for an n-bit register with asynchronous clear.

The D and Q signals in Figure 7.46 are defined in terms of N. The statement that resets all the bits of Q to 0 uses the odd-looking syntax Q <= (OTHERS => '0'). For the default value of $N = 16$, this statement is equivalent to the statement Q <= "0000000000000000". The (OTHERS => '0') syntax results in a '0' digit being assigned to each bit of Q, regardless of how many bits Q has. It allows the code to be used for any value of N, rather than only for $N = 16$.

Example 7.7

A FOUR-BIT SHIFT REGISTER Assume that we wish to write VHDL code that represents the four-bit shift register in Figure 7.19. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the *D input*. Figure 7.47 defines the entity named *muxdff*, which represents this subcircuit. The two data inputs are named D_0 and D_1, and they are selected using the *Sel* input. The process statement specifies that on the positive clock edge if $Sel = 0$, then Q is assigned the value of D_0; otherwise, Q is assigned the value of D_1.

Figure 7.48 defines the four-bit shift register. The statement labeled *Stage*3 instantiates the left-most flip-flop, which has the output Q_3, and the statement labeled *Stage*0 instantiates the right-most flip-flop, Q_0. When $L = 1$, it is loaded in parallel from the R input, and when $L = 0$, shifting takes place in the left to right direction. Serial data is shifted into the most-significant bit, Q_3, from the w input.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY muxdff IS
    PORT ( D0, D1, Sel, Clock  : IN    STD_LOGIC ;
           Q                   : OUT   STD_LOGIC ) ;
END muxdff ;

ARCHITECTURE Behavior OF muxdff IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Sel = '0' THEN
            Q <= D0 ;
        ELSE
            Q <= D1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.47 Code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
    PORT ( R            : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           L, w, Clock  : IN      STD_LOGIC ;
           Q            : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END shift4 ;

ARCHITECTURE Structure OF shift4 IS
    COMPONENT muxdff
        PORT ( D0, D1, Sel, Clock  : IN    STD_LOGIC ;
               Q                   : OUT   STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    Stage3: muxdff PORT MAP ( w, R(3), L, Clock, Q(3) ) ;
    Stage2: muxdff PORT MAP ( Q(3), R(2), L, Clock, Q(2) ) ;
    Stage1: muxdff PORT MAP ( Q(2), R(1), L, Clock, Q(1) ) ;
    Stage0: muxdff PORT MAP ( Q(1), R(0), L, Clock, Q(0) ) ;
END Structure ;
```

Figure 7.48 Hierarchical code for a four-bit shift register.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( R        : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5              Clock    : IN      STD_LOGIC ;
6              L, w     : IN      STD_LOGIC ;
7              Q        : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
8   END shift4 ;

9   ARCHITECTURE Behavior OF shift4 IS
10  BEGIN
11      PROCESS
12      BEGIN
13          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
14          IF L = '1' THEN
15              Q <= R ;
16          ELSE
17              Q(0) <= Q(1) ;
18              Q(1) <= Q(2);
19              Q(2) <= Q(3) ;
20              Q(3) <= w ;
21          END IF ;
22      END PROCESS ;
23  END Behavior ;
```

Figure 7.49 Alternative code for a shift register.

Example 7.8

ALTERNATIVE CODE FOR A FOUR-BIT SHIFT REGISTER A different style of code for the four-bit shift register is given in Figure 7.49. The lines of code are numbered for ease of reference. Instead of using subcircuits, the shift register is described using sequential statements. Due to the WAIT UNTIL statement in line 13, any signal that is assigned a value inside the process has to be implemented as the output of a flip-flop. Lines 14 and 15 specify the parallel loading of the shift register when $L = 1$. The ELSE clause in lines 16 to 20 specifies the shifting operation. Line 17 shifts the value of Q_1 into the flip-flop with the output Q_0. Lines 18 and 19 shift the values of Q_2 and Q_3 into the flip-flops with the outputs Q_1 and Q_2, respectively. Finally, line 20 shifts the value of w into the left-most flip-flop, which has the output Q_3. Note that the process semantics, described in section 6.6.6, stipulate that the four assignments in lines 17 to 20 are scheduled to occur only after all of the statements in the process have been evaluated. Hence all four flip-flops change their values at the same time, as required in the shift register. The code generates the same shift-register circuit as the code in Figure 7.48.

It is instructive to consider the effect of reversing the ordering of lines 17 through 20 in Figure 7.49, as indicated in Figure 7.50. In this case the first shift operation specified

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( R        : IN        STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5              Clock    : IN        STD_LOGIC ;
6              L, w     : IN        STD_LOGIC ;
7              Q        : BUFFER    STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
8   END shift4 ;

9   ARCHITECTURE Behavior OF shift4 IS
10  BEGIN
11      PROCESS
12      BEGIN
13          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
14          IF L = '1' THEN
15              Q <= R ;
16          ELSE
17              Q(3) <= w ;
18              Q(2) <= Q(3) ;
19              Q(1) <= Q(2);
20              Q(0) <= Q(1) ;
21          END IF ;
22      END PROCESS ;
23  END Behavior ;
```

Figure 7.50 Code that reverses the ordering of statements in Figure 7.49.

in the code, in line 17, shifts the value of *w* into the left-most flip-flop with the output Q_3. Due to the semantics of the process statement, the assignment to Q_3 does not take effect until all of the subsequent statements inside the process are evaluated. Hence line 18 shifts the present value of Q_3, before it is changed as a result of line 17, into the flip-flop with the output Q_2. Similarly, lines 19 and 20 shift the present values of Q_2 and Q_1 into the flip-flops with the outputs Q_1 and Q_0, respectively. The code produces the same circuit as it did with the ordering of the statements in Figure 7.49.

Example 7.9 ***N*-BIT SHIFT REGISTER** Figure 7.51 shows code that can be used to represent shift registers of any size. The GENERIC parameter N, which has the default value 8 in the figure, sets the number of flip-flops. The code is identical to that in Figure 7.49 with two exceptions. First, R and Q are defined in terms of N. Second, the ELSE clause that describes the shifting operation is generalized to work for any number of flip-flops.

Lines 18 to 20 specify the shifting operation for the right-most $N - 1$ flip-flops, which have the outputs Q_{N-2} to Q_0. The construct used is called a FOR LOOP. It is similar to the

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shiftn IS
4       GENERIC ( N : INTEGER := 8 ) ;
5       PORT ( R        : IN      STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
6              Clock    : IN      STD_LOGIC ;
7              L, w     : IN      STD_LOGIC ;
8              Q        : BUFFER  STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
9   END shiftn ;

10  ARCHITECTURE Behavior OF shiftn IS
11  BEGIN
12      PROCESS
13      BEGIN
14          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
15          IF L = '1' THEN
16              Q <= R ;
17          ELSE
18              Genbits: FOR i IN 0 TO N-2 LOOP
19                  Q(i) <= Q(i + 1) ;
20              END LOOP ;
21              Q(N-1) <= w ;
22          END IF ;
23      END PROCESS ;
24  END Behavior ;
```

Figure 7.51 Code for an n-bit left-to-right shift register.

FOR GENERATE statement, introduced in section 6.6.4, which is used to generate a set of concurrent statements. The FOR LOOP is used to generate a set of sequential statements. The first loop iteration shifts the present value of Q_1 into the flip-flop with the output Q_0. The next loop iteration shifts Q_2 into the flip-flop with the output Q_1, and so on, with the final iteration shifting Q_{N-1} into the flip-flop with the output Q_{N-2}. Line 21 completes the shift operation by shifting the value of the serial input w into the left-most flip-flop with the output Q_{N-1}.

Example 7.10

UP-COUNTER Figure 7.52 shows the code for a four-bit up-counter that has a reset input, *Resetn*, and an enable input, *E*. In the architecture body the flip-flops in the counter are represented by the signal named *Count*. The process statement specifies an asynchronous reset of *Count* if *Resetn* = 0. The ELSIF clause specifies that on the positive clock edge,

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clock, Resetn, E  : IN    STD_LOGIC ;
           Q                 : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            ELSE
                Count <= Count ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

Figure 7.52 Code for a four-bit up-counter.

if $E = 1$, the count is incremented. If $E = 0$, the code explicitly assigns *Count* <= *Count*. This statement is not required to correctly describe the counter, because of the implied memory semantics, but it may be included for clarity. The Q outputs are assigned the value of *Count* at the end of the code. The code produces the circuit shown in Figure 7.23 if the VHDL compiler opts to use T flip-flops, and it generates the circuit in Figure 7.24 (with the reset input added) if the compiler chooses D flip-flops.

Example 7.11 **USING INTEGER SIGNALS IN A COUNTER** Counters are often defined in VHDL using the INTEGER type, which was introduced in section 5.5.4. The code in Figure 7.53 defines an up-counter that has a parallel-load input in addition to a reset input. The parallel data, *R*, as well as the counter's output, Q, are defined using the INTEGER type. Since they

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY upcount IS
    PORT ( R                  : IN      INTEGER RANGE 0 TO 15 ;
           Clock, Resetn, L   : IN      STD_LOGIC ;
           Q                  : BUFFER  INTEGER RANGE 0 TO 15 ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Q <= 0 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF L = '1' THEN
                Q <= R ;
            ELSE
                Q <= Q + 1 ;
            END IF;
        END IF;
    END PROCESS;
END Behavior;
```

Figure 7.53 A four-bit counter with parallel load, using INTEGER signals.

have the range from 0 to 15, both of these signals represent four-bit quantities. In Figure 7.52 the signal *Count* is defined to represent the flip-flops in the counter. This signal is not needed if the Q outputs have the BUFFER mode, as shown in Figure 7.53. The if-then-else statement at the beginning of the process includes the same asynchronous reset as in Figure 7.53. The ELSIF clause specifies that on the positive clock edge, if $L = 1$, the flip-flops in the counter are loaded in parallel from the R inputs. If $L = 0$, the count is incremented.

Example 7.12

DOWN-COUNTER Figure 7.54 shows the code for a down-counter named *downcnt*. To make it easy to change the starting count, it is defined as a GENERIC parameter named *modulus*. On the positive clock edge, if $L = 1$, the counter is loaded with the value $modulus-1$, and if $L = 0$, the count is decremented. The counter also includes an enable

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY downcnt IS
    GENERIC ( modulus : INTEGER := 8 ) ;
    PORT ( Clock, L, E  : IN    STD_LOGIC ;
                Q        : OUT  INTEGER RANGE 0 TO modulus−1 ) ;
END downcnt ;

ARCHITECTURE Behavior OF downcnt IS
    SIGNAL Count : INTEGER RANGE 0 TO modulus−1 ;
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
        IF L = '1' THEN
            Count <= modulus−1 ;
        ELSE
            IF E = '1' THEN
                Count <= Count−1 ;
            END IF ;
        END IF ;
    END PROCESS;
    Q <= Count ;
END Behavior ;
```

Figure 7.54 Code for a down-counter.

input, E. Setting $E = 1$ allows the count to be decremented when an active clock edge occurs.

7.14 Design Examples

This section presents two examples of digital systems that make use of some of the building blocks described in this chapter and in Chapter 6.

7.14.1 Bus Structure

Digital systems often contain a set of registers used to store data. Figure 7.55 gives an example of a system that has k n-bit registers, $R1$ to Rk. Each register is connected to a common set of n wires, which are used to transfer data into and out of the registers. This

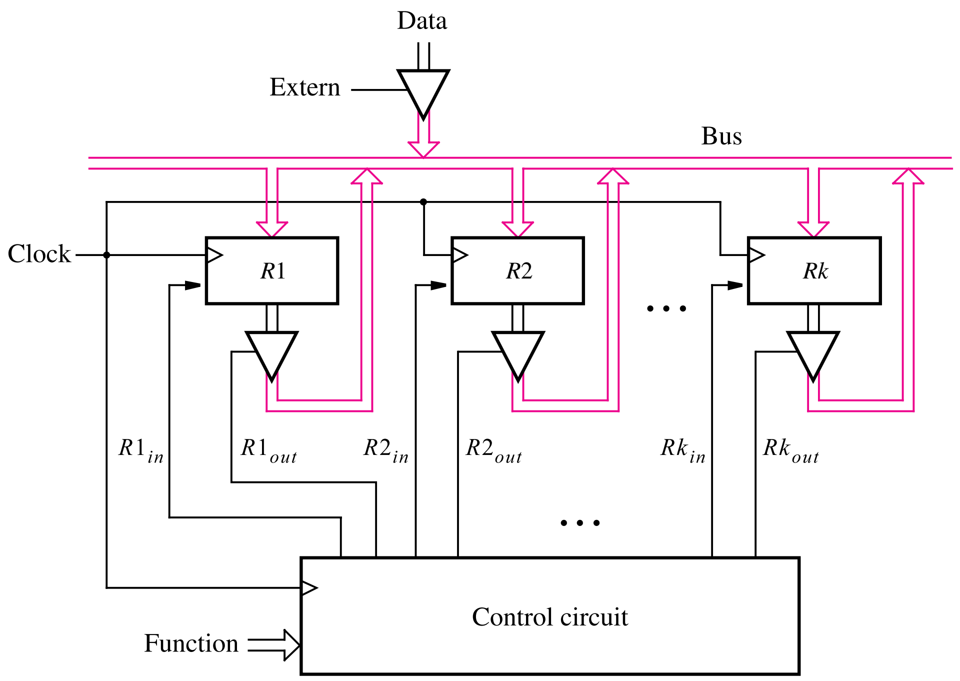


Figure 7.55 A digital system with k registers.

common set of wires is usually called a *bus*. In addition to registers, in a real system other types of circuit blocks would be connected to the bus. The figure shows how n bits of data can be placed on the bus from another circuit block, using the control input *Extern*. The data stored in any of the registers can be transferred via the bus to a different register or to another circuit block that is connected to the bus.

It is essential to ensure that only one circuit block attempts to place data onto the bus wires at any given time. In Figure 7.55 each register is connected to the bus through an n-bit tri-state buffer. A control circuit is used to ensure that only one of the tri-state buffer enable inputs, $R1_{out}, \ldots, Rk_{out}$, is asserted at a given time. The control circuit also produces the signals $R1_{in}, \ldots, Rk_{in}$, which control when data is loaded into each register. In general, the control circuit could perform a number of functions, such as transferring the data stored in one register into another register and the like. Figure 7.55 shows an input signal named *Function* that instructs the control circuit to perform a particular task. The control circuit is synchronized by a clock input, which is the same clock signal that controls the k registers.

Figure 7.56 provides a more detailed view of how the registers from Figure 7.55 can be connected to a bus. To keep the picture simple, 2 two-bit registers are shown, but the same scheme can be used for larger registers. For register $R1$, two tri-state buffers enabled by $R1_{out}$ are used to connect each flip-flop output to a wire in the bus. The D input on each flip-flop is connected to a 2-to-1 multiplexer, whose select input is controlled by $R1_{in}$.

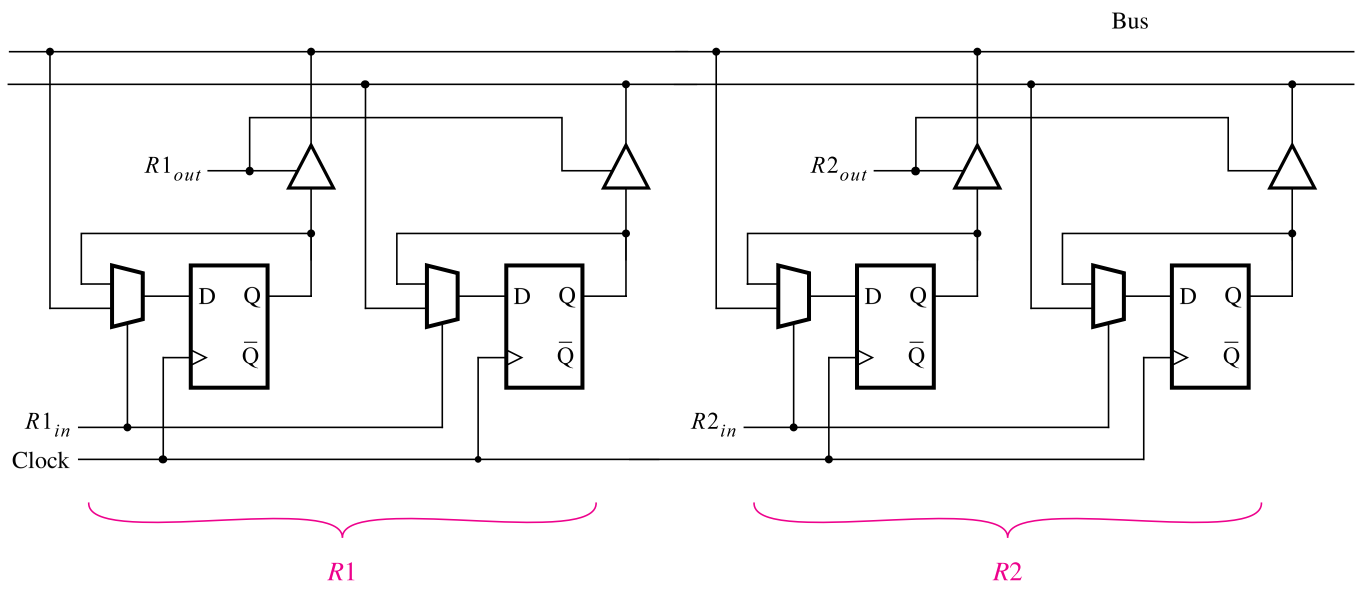


Figure 7.56 Details for connecting registers to a bus.

If $R1_{in} = 0$, the flip-flops are loaded from their Q outputs; hence the stored data does not change. But if $R1_{in} = 1$, data is loaded into the flip-flops from the bus. Instead of using multiplexers on the flip-flop inputs, one could attempt to connect the D inputs on the flip-flops directly to the bus. Then it is necessary to control the clock inputs on all flip-flops to ensure that they are clocked only when new data should be loaded into the register. This approach is not good because it may happen that different flip-flops will be clocked at slightly different times, leading to a problem known as *clock skew*. A detailed discussion of the issues related to the clocking of flip-flops is provided in section 10.3.

The system in Figure 7.55 can be used in many different ways, depending on the design of the control circuit and on how many registers and other circuit blocks are connected to the bus. As a simple example, consider a system that has three registers, $R1$, $R2$, and $R3$. Each register is connected to the bus as indicated in Figure 7.56. We will design a control circuit that performs a single function—it swaps the contents of registers $R1$ and $R2$, using $R3$ for temporary storage.

The required swapping is done in three steps, each needing one clock cycle. In the first step the contents of $R2$ are transferred into $R3$. Then the contents of $R1$ are transferred into $R2$. Finally, the contents of $R3$, which are the original contents of $R2$, are transferred into $R1$. Note that we say that the contents of one register, R_i, are "transferred" into another register, R_j. This jargon is commonly used to indicate that the new contents of R_j will be a copy of the contents of R_i. The contents of R_i are not changed as a result of the transfer. Therefore, it would be more precise to say that the contents of R_i are "copied" into R_j.

Using a Shift Register for Control

There are many ways to design a suitable control circuit for the swap operation. One possibility is to use the left-to-right shift register shown in Figure 7.57. Assume that the reset input is used to clear the flip-flops to 0. Hence the control signals $R1_{in}$, $R1_{out}$, and so on are not asserted, because the shift register outputs have the value 0. The serial input w normally has the value 0. We assume that changes in the value of w are synchronized to occur shortly after the active clock edge. This assumption is reasonable because w would normally be generated as the output of some circuit that is controlled by the same clock signal. When the desired swap should be performed, w is set to 1 for one clock cycle, and

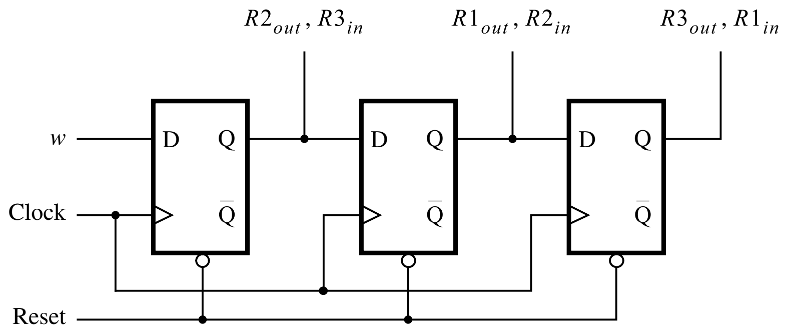


Figure 7.57 A shift-register control circuit.

then w returns to 0. After the next active clock edge, the output of the left-most flip-flop becomes equal to 1, which asserts both $R2_{out}$ and $R3_{in}$. The contents of register $R2$ are placed onto the bus wires and are loaded into register $R3$ on the next active clock edge. This clock edge also shifts the contents of the shift register, resulting in $R1_{out} = R2_{in} = 1$. Note that since w is now 0, the first flip-flop is cleared, causing $R2_{out} = R3_{in} = 0$. The contents of $R1$ are now on the bus and are loaded into $R2$ on the next clock edge. After this clock edge the shift register contains 001 and thus asserts $R3_{out}$ and $R1_{in}$. The contents of $R3$ are now on the bus and are loaded into $R1$ on the next clock edge.

Using the control circuit in Figure 7.57, when w changes to 1 the swap operation does not begin until after the next active clock edge. We can modify the control circuit so that it starts the swap operation in the same clock cycle in which w changes to 1. One possible approach is illustrated in Figure 7.58. The reset signal is used to set the shift-register contents to 100, by presetting the left-most flip-flop to 1 and clearing the other two flip-flops. As long as $w = 0$, the output control signals are not asserted. When w changes to 1, the signals $R2_{out}$ and $R3_{in}$ are immediately asserted and the contents of $R2$ are placed onto the bus. The next active clock edge loads this data into $R3$ and also shifts the shift register contents to 010. Since the signal $R1_{out}$ is now asserted, the contents of $R1$ appear on the bus. The next clock edge loads this data into $R2$ and changes the shift register contents to 001. The contents of $R3$ are now on the bus; this data is loaded into $R1$ at the next clock edge, which also changes the shift register contents to 100. We assume that w had the value 1 for only one clock cycle; hence the output control signals are not asserted at this point. It may not be obvious to the reader how to design a circuit such as the one in Figure 7.58, because we have presented the design in an ad hoc fashion. In section 8.3 we will show how this circuit can be designed using a more formal approach.

The circuit in Figure 7.58 assumes that a preset input is available on the left-most flip-flop. If the flip-flop has only a clear input, then we can use the equivalent circuit shown in Figure 7.59. In this circuit we use the $\overline{Q}$ output of the left-most flip-flop and also complement the input to this flip-flop by using a NOR gate instead of an OR gate.

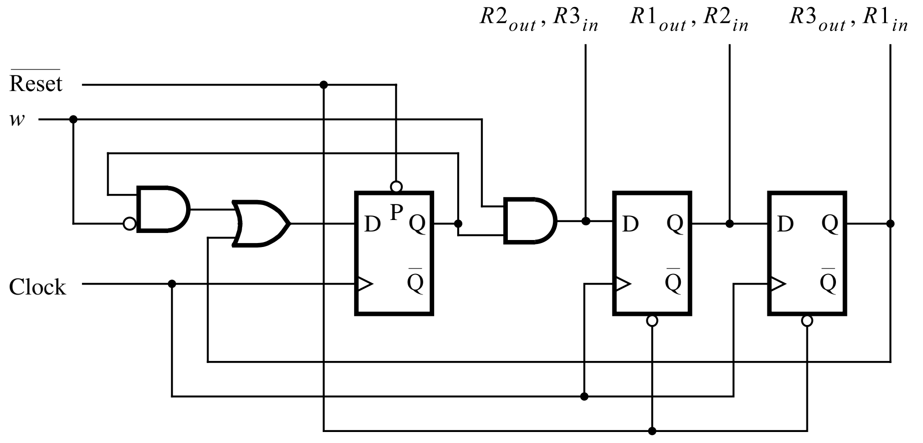


Figure 7.58 A modified control circuit.

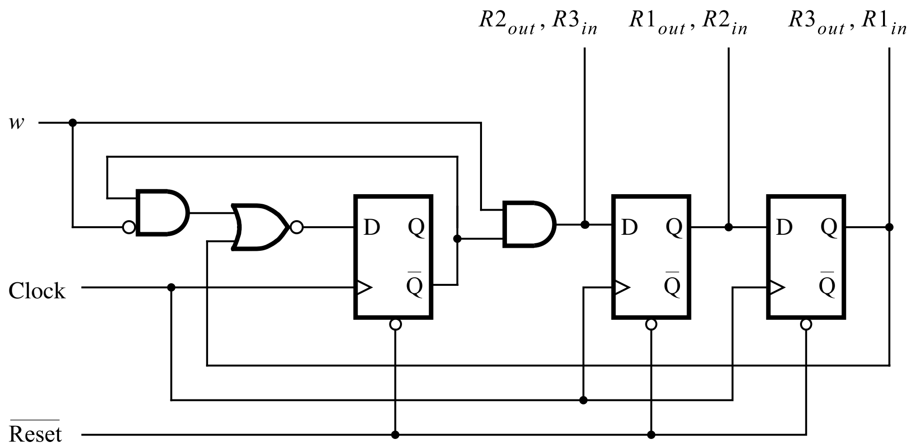


Figure 7.59 A modified version of the circuit in Figure 7.58.

Using a Multiplexer to Implement a Bus

In Figure 7.55 we used tri-state buffers to control access to the bus. An alternative approach is to use multiplexers, as depicted in Figure 7.60. The outputs of each register are connected to a multiplexer. This multiplexer's output is connected to the inputs of the registers, thus realizing the bus. The multiplexer select inputs determine which register's contents appear on the bus. Although the figure shows just one multiplexer symbol, we actually need one multiplexer for each bit in the registers. For example, assume that there are 4 eight-bit registers, $R1$ to $R4$, plus the externally-supplied eight-bit *Data*. To interconnect them, we need eight 5-to-1 multiplexers. In Figure 7.57 we used a shift

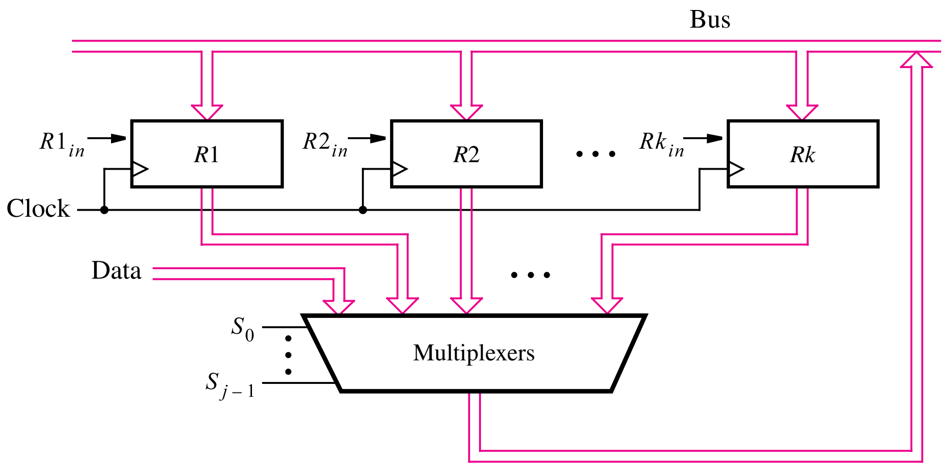


Figure 7.60 Using multiplexers to implement a bus.

register to implement the control circuit. A similar approach can be used with multiplexers. The signals that control when data is loaded into a register, like $R1_{in}$, can still be connected directly to the shift-register outputs. However, instead of using control signals like $R1_{out}$ to place the contents of a register onto the bus, we have to generate the select inputs for the multiplexers. One way to do so is to connect the shift-register outputs to an encoder circuit that produces the select inputs for the multiplexer. We discussed encoder circuits in section 6.3.

The tri-state buffer and multiplexer approaches for implementing a bus are both equally valid. However, some types of chips, such as most PLDs, do not contain a sufficient number of tri-state buffers to realize even moderately large buses. In such chips the multiplexer-based approach is the only practical alternative. In practice, circuits are designed with CAD tools. If the designer describes the circuit using tri-state buffers, but there are not enough such buffers in the target device, then the CAD tools automatically produce an equivalent circuit that uses multiplexers.

VHDL Code

This section presents VHDL code for our circuit example that swaps the contents of two registers. We first give the code for the style of circuit in Figure 7.55 that uses tri-state buffers to implement the bus and then give the code for the style of circuit in Figure 7.60 that uses multiplexers. The code is written in a hierarchical fashion, using subcircuits for the registers, tri-state buffers, and the shift register. Figure 7.61 gives the code for an *n*-bit register of the type in Figure 7.56. The number of bits in the register is set by

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( R            : IN   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
           Rin, Clock   : IN   STD_LOGIC ;
           Q            : OUT  STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Rin = '1' THEN
            Q <= R ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 7.61 Code for an *n*-bit register of the type in Figure 7.56.

the generic parameter N, which has the default value of 8. The process that describes the register specifies that if the input $Rin = 1$, then the flip-flops are loaded from the n-bit input R. Otherwise, the flip-flops retain their presently stored values. The circuit synthesized from this code has a 2-to-1 multiplexer controlled by *Rin* connected to the D input on each flip-flop, as depicted in Figure 7.56.

Figure 7.62 gives the code for a subcircuit that represents n tri-state buffers, each enabled by the input E. The number of buffers is set by the generic parameter N. The inputs to the buffers are the n-bit signal X, and the outputs are the n-bit signal F. The architecture uses the syntax (OTHERS => 'Z') to specify that the output of each buffer is set to the value Z if $E = 0$; otherwise, the output is set to $F = X$.

Figure 7.63 provides the code for a shift register that can be used to implement the control circuit in Figure 7.57. The number of flip-flops is set by the generic parameter K, which has the default value of 4. The shift register has an active-low asynchronous reset input. The shift operation is defined with a FOR LOOP in the style used in Example 7.9.

To use the entities in Figures 7.61 through 7.63 as subcircuits, we have to provide component declarations for each one. For convenience, we placed these declarations inside a single package, named *components*, which is shown in Figure 7.64. This package is used in the code given in Figure 7.65. It represents the digital system in Figure 7.55 with 3 eight-bit registers, $R1$, $R2$, and $R3$.

The circuit in Figure 7.55 includes tri-state buffers that are used to place n bits of externally supplied data on the bus. In the code in Figure 7.65, these buffers are instantiated in the statement labeled *tri_ext*. Each of the eight buffers is enabled by the input signal *Extern*, and the data inputs on the buffers are attached to the eight-bit signal *Data*. When *Extern* = 1, the value of *Data* is placed on the bus, which is represented by the signal *BusWires*. The *BusWires* port represents the circuit's output. This port has the mode INOUT, which is required because *BusWires* is connected to the outputs of tri-state buffers and these buffers are connected to the inputs of the registers.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY trin IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
           E  : IN   STD_LOGIC ;
           F  : OUT  STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
END trin ;

ARCHITECTURE Behavior OF trin IS
BEGIN
    F <= (OTHERS => 'Z') WHEN E = '0' ELSE X ;
END Behavior ;
```

Figure 7.62 Code for an n-bit tri-state buffer.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shiftr IS - - left-to-right shift register with async reset
     GENERIC ( K : INTEGER := 4 ) ;
     PORT ( Resetn, Clock, w  : IN       STD_LOGIC ;
                Q               : BUFFER STD_LOGIC_VECTOR(1 TO K) ) ;
END shiftr ;

ARCHITECTURE Behavior OF shiftr IS
BEGIN
     PROCESS ( Resetn, Clock )
     BEGIN
          IF Resetn = '0' THEN
               Q <= (OTHERS => '0') ;
          ELSIF Clock'EVENT AND Clock = '1' THEN
               Genbits: FOR i IN K DOWNTO 2 LOOP
                    Q(i) <= Q(i-1) ;
               END LOOP ;
               Q(1) <= w ;
          END IF ;
     END PROCESS ;
END Behavior ;
```

Figure 7.63 Code for the shift register in Figure 7.57.

We assume that a three-bit control signal named *RinExt* exists, which is used to allow the externally supplied data to be loaded from the bus into registers $R1$, $R2$, or $R3$. The *RinExt* input is not shown in Figure 7.55, to keep the figure simple, but it would be generated by the same external circuit block that produces *Extern* and *Data*. When $RinExt(1) = 1$, the data on the bus is loaded into register $R1$; when $RinExt(2) = 1$, the data is loaded into $R2$; and when $RinExt(3) = 1$, the data is loaded into $R3$.

In Figure 7.65 the three-bit shift register is instantiated in the statement labeled *control*. The outputs of the shift register are the three-bit signal Q. The next three statements connect Q to the control signals that determine when data is loaded into each register, which are represented by the three-bit signal *Rin*. The signals $Rin(1)$, $Rin(2)$, and $Rin(3)$ in the code correspond to the signals $R1_{in}$, $R2_{in}$, and $R3_{in}$ in Figure 7.55. As specified in Figure 7.57, the left-most shift-register output, Q(1), controls when data is loaded into register $R3$. Similarly, Q(2) controls register $R2$, and Q(3) controls $R1$. Each bit in *Rin* is ORed with the corresponding bit in *RinExt* so that externally supplied data can be stored in the registers as discussed above. The code also connects the shift-register outputs to the enable inputs, called *Rout*, on the tri-state buffers that connect the registers to the bus. Figure 7.57 shows that Q(1) is used to put the contents of $R2$ onto the bus; hence $Rout(2)$ is assigned the value

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS

   COMPONENT regn - - register
      GENERIC ( N : INTEGER := 8 ) ;
      PORT ( R          : IN    STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
             Rin, Clock : IN    STD_LOGIC ;
             Q          : OUT   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
   END COMPONENT ;

   COMPONENT shiftr - - left-to-right shift register with async reset
      GENERIC ( K : INTEGER := 4 ) ;
      PORT ( Resetn, Clock, w  : IN       STD_LOGIC ;
             Q                 : BUFFER   STD_LOGIC_VECTOR(1 TO K) ) ;
   END component ;

   COMPONENT trin - - tri-state buffers
      GENERIC ( N : INTEGER := 8 ) ;
      PORT ( X  : IN    STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
             E  : IN    STD_LOGIC ;
             F  : OUT   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
   END COMPONENT ;

END components ;
```

Figure 7.64 Package and component declarations.

of Q(1). Similarly, *Rout*(1) is assigned the value of Q(2), and *Rout*(3) is assigned the value of Q(3). The remaining statements in the code instantiate the registers and tri-state buffers in the system.

VHDL Code Using Multiplexers

Figure 7.66 shows how the code in Figure 7.65 can be modified to use multiplexers instead of tri-state buffers. Using the circuit structure shown in Figure 7.60, the bus is implemented using eight 4-to-1 multiplexers. Three of the data inputs on each 4-to-1 multiplexer are connected to one bit from registers $R1$, $R2$, and $R3$. The fourth data input is connected to one bit of the *Data* input signal to allow externally supplied data to be written into the registers. When the shift register's contents are 000, the multiplexers select *Data* to be placed on the bus. This data is loaded into the register selected by *RinExt*. It is loaded into $R1$ if $RinExt(1) = 1$, $R2$ if $RinExt(2) = 1$, and $R3$ if $RinExt(3) = 1$.

The *Rout* signal in Figure 7.65, which is used as the enable inputs on the tri-state buffers connected to the bus, is not needed for the multiplexer implementation. Instead, we have

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY swap IS
    PORT ( Data          : IN    STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Resetn, w     : IN    STD_LOGIC ;
           Clock, Extern : IN    STD_LOGIC ;
           RinExt        : IN    STD_LOGIC_VECTOR(1 TO 3) ;
           BusWires      : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END swap ;

ARCHITECTURE Behavior OF swap IS
    SIGNAL Rin, Rout, Q : STD_LOGIC_VECTOR(1 TO 3) ;
    SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
BEGIN
    control: shiftr GENERIC MAP ( K => 3 )
                PORT MAP ( Resetn, Clock, w, Q ) ;
    Rin(1) <= RinExt(1) OR Q(3) ;
    Rin(2) <= RinExt(2) OR Q(2) ;
    Rin(3) <= RinExt(3) OR Q(1) ;
    Rout(1) <= Q(2) ; Rout(2) <= Q(1) ; Rout(3) <= Q(3) ;

    tri_ext: trin PORT MAP ( Data, Extern, BusWires ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
    tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
    tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
END Behavior ;
```

Figure 7.65 A digital system like the one in Figure 7.55.

to provide the select inputs on the multiplexers. In the architecture body in Figure 7.66, the shift-register outputs are called Q. These signals are used to generate the *Rin* control signals for the registers in the same way as shown in Figure 7.65. We said in the discussion concerning Figure 7.60 that an encoder is needed between the shift-register outputs and the multiplexer select inputs. A suitable encoder is described in the selected signal assignment labeled *encoder*. It produces the multiplexer select inputs, which are named S. It sets $S = 00$ when the shift register contains 000, $S = 10$ when the shift register contains 100, and so on, as given in the code. The multiplexers are described by the selected signal assignment labeled *muxes*. This statement places the value of *Data* onto the bus (*BusWires*) if $S = 00$, the contents of register $R1$ if $S = 01$, and so on. Using this scheme, when the swap operation is not active, the multiplexers place the bits from the *Data* input on the bus.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;
ENTITY swapmux IS
     PORT ( Data          : IN        STD_LOGIC_VECTOR(7 DOWNTO 0) ;
               Resetn, w  : IN        STD_LOGIC ;
               Clock        : IN        STD_LOGIC ;
               RinExt      : IN        STD_LOGIC_VECTOR(1 TO 3) ;
               BusWires   : BUFFER  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END swapmux ;

ARCHITECTURE Behavior OF swapmux IS
     SIGNAL Rin, Q : STD_LOGIC_VECTOR(1 TO 3) ;
     SIGNAL S : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
     SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
BEGIN
     control: shiftr GENERIC MAP ( K => 3 )
                    PORT MAP ( Resetn, Clock, w, Q ) ;
     Rin(1) <= RinExt(1) OR Q(3) ;
     Rin(2) <= RinExt(2) OR Q(2) ;
     Rin(3) <= RinExt(3) OR Q(1) ;

     reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
     reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
     reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
     encoder:
     WITH Q SELECT
          S <= "00" WHEN "000",
                  "10" WHEN "100",
                  "01" WHEN "010",
                  "11" WHEN OTHERS;
     muxes: --eight 4-to-1 multiplexers
     WITH S SELECT
          BusWires <= Data WHEN "00",
                            R1 WHEN "01",
                            R2 WHEN "10",
                            R3 WHEN OTHERS ;
END Behavior ;
```

Figure 7.66 Using multiplexers to implement a bus.

In Figure 7.66 we use two selected signal assignments, one to describe an encoder and the other to describe the bus multiplexers. A simpler approach is to use a single selected signal assignment as shown in Figure 7.67. The statement labeled *muxes* specifies directly which signal should appear on *BusWires* for each pattern of the shift-register outputs. The circuit synthesized from this statement is similar to an 8-to-1 multiplexer with the three

```
ARCHITECTURE Behavior OF swapmux IS
    SIGNAL Rin, Q : STD_LOGIC_VECTOR(1 TO 3) ;
    SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
BEGIN
    control: shiftr GENERIC MAP ( K => 3 )
                PORT MAP ( Resetn, Clock, w, Q ) ;
    Rin(1) <= RinExt(1) OR Q(3) ;
    Rin(2) <= RinExt(2) OR Q(2) ;
    Rin(3) <= RinExt(3) OR Q(1) ;

    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;

    muxes:
    WITH Q SELECT
        BusWires <= Data WHEN "000",
                    R2 WHEN "100",
                    R1 WHEN "010",
                    R3 WHEN OTHERS ;
END Behavior ;
```

Figure 7.67 A simplified version of the architecture in Figure 7.66.

select inputs connected to the shift-register outputs. However, only half of the multiplexer circuit is actually generated by the synthesis tools because there are only four data inputs. The circuit generated from the code in Figure 7.67 is the same as the one generated from the code in Figure 7.66.

Figure 7.68 gives an example of a timing simulation for a circuit synthesized from the code in Figure 7.67. In the first half of the simulation, the circuit is reset, and the contents of registers $R1$ and $R2$ are initialized. The hex value 55 is loaded into $R1$, and the value AA is loaded into $R2$. The clock edge at 275 ns, marked by the vertical reference line in Figure 7.68, loads the value $w = 1$ into the shift register. The contents of $R2$ (AA) then appear on the bus and are loaded into $R3$ by the clock edge at 325 ns. Following this clock edge, the contents of the shift register are 010, and the data stored in $R1$ (55) is on the bus. The clock edge at 375 ns loads this data into $R2$ and changes the shift register to 001. The contents of $R3$ (AA) now appear on the bus and are loaded into $R1$ by the clock edge at 425 ns. The shift register is now in state 000, and the swap is completed.

7.14.2 Simple Processor

A second example of a digital system like the one in Figure 7.55 is shown in Figure 7.69. It has four n-bit registers, $R0, \ldots, R3$, that are connected to the bus using tri-state buffers.

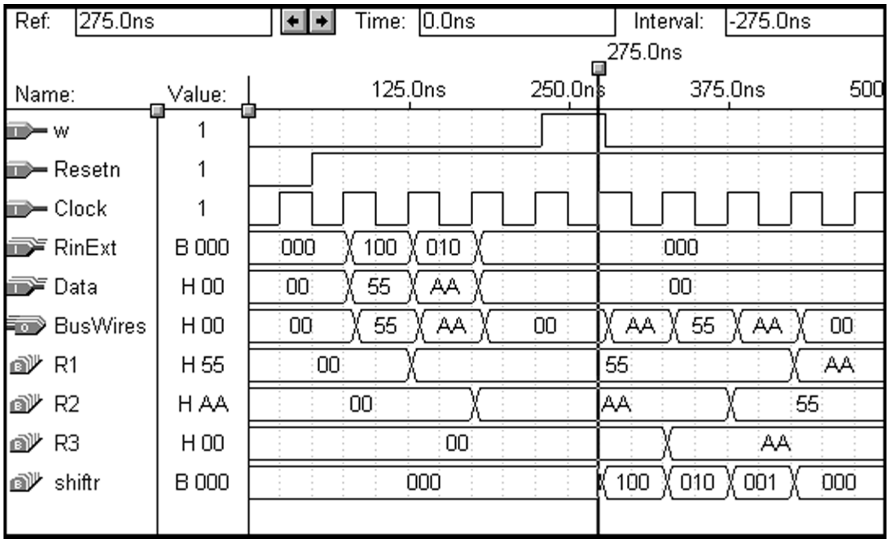


Figure 7.68 Timing simulation for the VHDL code in Figure 7.67.

External data can be loaded into the registers from the n-bit *Data* input, which is connected to the bus using tri-state buffers enabled by the *Extern* control signal. The system also includes an adder/subtractor module. One of its data inputs is provided by an n-bit register, A, that is attached to the bus, while the other data input, B, is directly connected to the bus. If the *AddSub* signal has the value 0, the module generates the sum $A + B$; if $AddSub = 1$, the module generates the difference $A - B$. To perform the subtraction, we assume that the adder/subtractor includes the required XOR gates to form the 2's complement of B, as discussed in section 5.3. The register G stores the output produced by the adder/subtractor. The A and G registers are controlled by the signals A_{in}, G_{in}, and G_{out}.

The system in Figure 7.69 can perform various functions, depending on the design of the control circuit. As an example, we will design a control circuit that can perform the four operations listed in Table 7.2. The left column in the table shows the name of an operation and its operands; the right column indicates the function performed in the operation. For the *Load* operation the meaning of $Rx \leftarrow Data$ is that the data on the external *Data* input is transferred across the bus into any register, Rx, where Rx can be $R0$ to $R3$. The *Move* operation copies the data stored in register Ry into register Rx. In the table the square brackets, as in $[Rx]$, refer to the *contents* of a register. Since only a single transfer across the bus is needed, both the *Load* and *Move* operations require only one step (clock cycle) to be completed. The *Add* and *Sub* operations require three steps, as follows: In the first step the contents of Rx are transferred across the bus into register A. Then in the next step, the contents of Ry are placed onto the bus. The adder/subtractor module performs the required function, and the results are stored in register G. Finally, in the third step the contents of G are transferred into Rx.

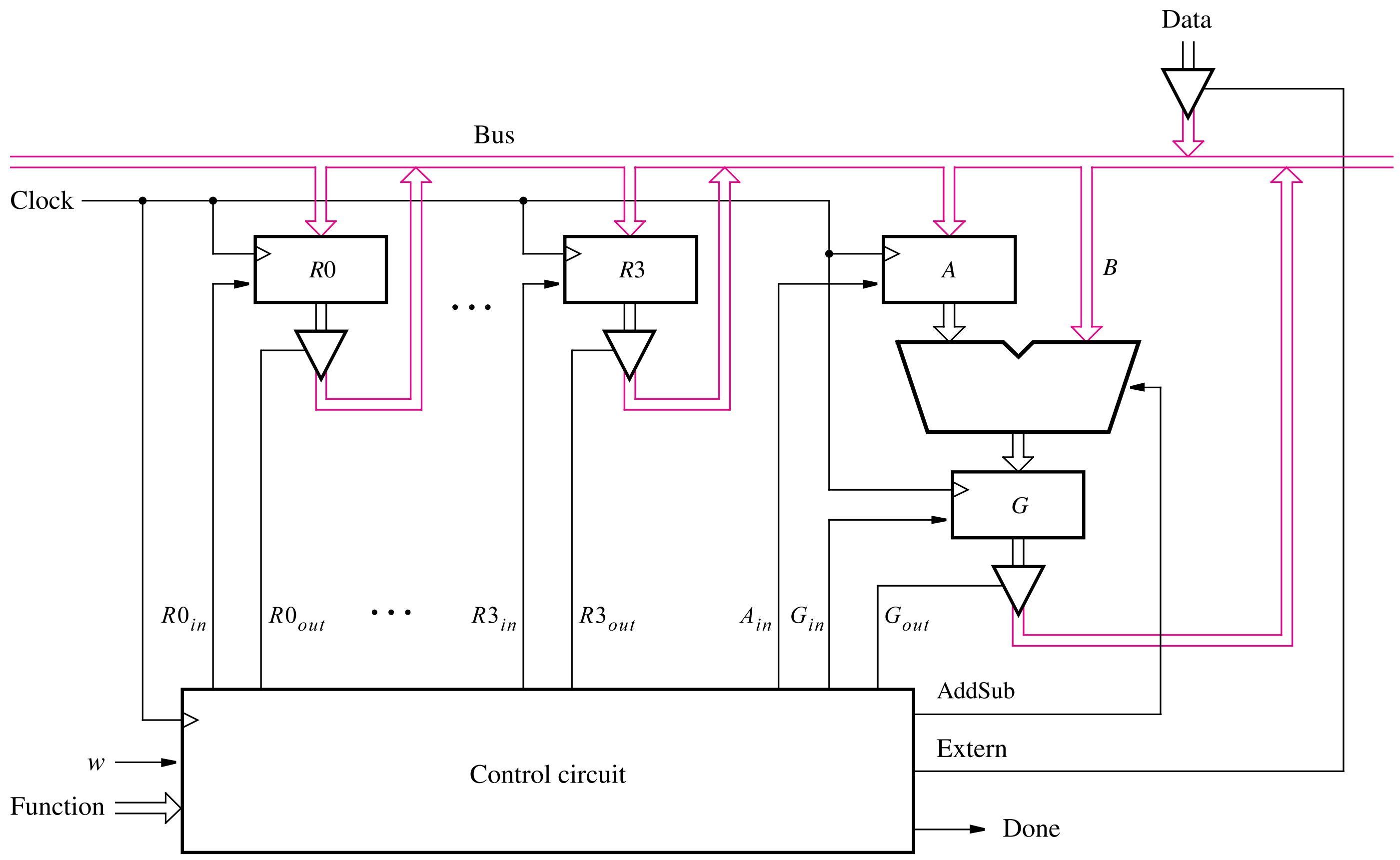


Figure 7.69 A digital system that implements a simple processor.

Table 7.2 Operations performed in the processor.

Operation	Function Performed
Load Rx, $Data$	$Rx \leftarrow Data$
Move Rx, Ry	$Rx \leftarrow [Ry]$
Add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
Sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

A digital system that performs the types of operations listed in Table 7.2 is usually called a *processor*. The specific operation to be performed at any given time is indicated using the control circuit input named *Function*. The operation is initiated by setting the w input to 1, and the control circuit asserts the *Done* output when the operation is completed.

In Figure 7.55 we used a shift register to implement the control circuit. It is possible to use a similar design for the system in Figure 7.69. To illustrate a different approach, we will base the design of the control circuit on a counter. This circuit has to generate the required control signals in each step of each operation. Since the longest operations (*Add* and *Sub*) need three steps (clock cycles), a two-bit counter can be used. Figure 7.70 shows a two-bit up-counter connected to a 2-to-4 decoder. Decoders are discussed in section 6.2. The decoder is enabled at all times by setting its enable (*En*) input permanently to the value 1. Each of the decoder outputs represents a step in an operation. When no operation is currently being performed, the count value is 00; hence the T_0 output of the decoder is asserted. In the first step of an operation, the count value is 01, and T_1 is asserted. During the second and third steps of the *Add* and *Sub* operations, T_2 and T_3 are asserted, respectively.

In each of steps T_0 to T_3, various control signal values have to be generated by the control circuit, depending on the operation being performed. Figure 7.71 shows that the operation is specified with six bits, which form the *Function* input. The two left-most bits, $F = f_1 f_0$, are used as a two-bit number that identifies the operation. To represent *Load*, *Move*, *Add*, and *Sub*, we use the codes $f_1 f_0$ = 00, 01, 10, and 11, respectively. The inputs $Rx_1 Rx_0$ are a binary number that identifies the Rx operand, while $Ry_1 Ry_0$ identifies the Ry operand. The *Function* inputs are stored in a six-bit Function Register when the FR_{in} signal is asserted.

Figure 7.71 also shows three 2-to-4 decoders that are used to decode the information encoded in the F, Rx, and Ry inputs. We will see shortly that these decoders are included as a convenience because their outputs provide simple-looking logic expressions for the various control signals.

The circuits in Figures 7.70 and 7.71 form a part of the control circuit. Using the input w and the signals $T_0, \ldots, T_3$, $I_0, \ldots, I_3$, $X_0, \ldots, X_3$, and $Y_0, \ldots, Y_3$, we will show how to derive the rest of the control circuit. It has to generate the outputs *Extern*, *Done*, A_{in}, G_{in}, G_{out}, *AddSub*, $R0_{in}, \ldots, R3_{in}$, and $R0_{out}, \ldots, R3_{out}$. The control circuit also has to generate the *Clear* and FR_{in} signals used in Figures 7.70 and 7.71.

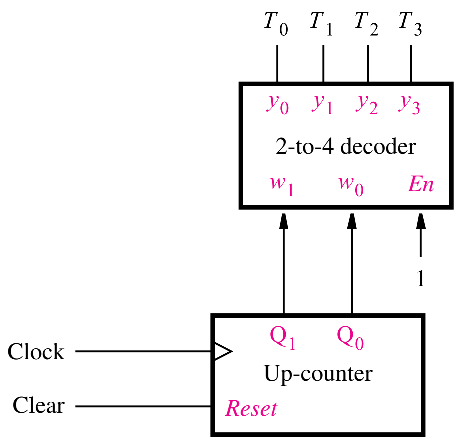


Figure 7.70 A part of the control circuit for the processor.

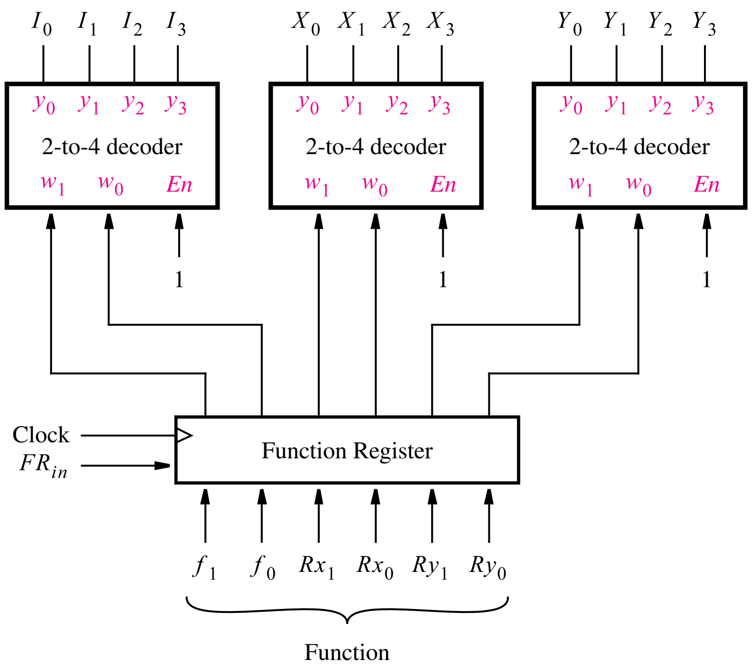


Figure 7.71 The function register and decoders.

Table 7.3 Control signals asserted in each operation/time step.

	T_1	T_2	T_3
(Load): I_0	$Extern$, $R_{in} = X$, $Done$		
(Move): I_1	$R_{in} = X$, $R_{out} = Y$, $Done$		
(Add): I_2	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, $AddSub = 0$	G_{out}, $R_{in} = X$, $Done$
(Sub): I_3	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, $AddSub = 1$	G_{out}, $R_{in} = X$, $Done$

Clear and FR_{in} are defined in the same way for all operations. *Clear* is used to ensure that the count value remains at 00 as long as $w = 0$ and no operation is being executed. Also, it is used to clear the count value to 00 at the end of each operation. Hence an appropriate logic expression is

$$Clear = \overline{w}\,T_0 + Done$$

The FR_{in} signal is used to load the values on the *Function* inputs into the Function Register when w changes to 1. Hence

$$FR_{in} = wT_0$$

The rest of the outputs from the control circuit depend on the specific step being performed in each operation. The values that have to be generated for each signal are shown in Table 7.3. Each row in the table corresponds to a specific operation, and each column represents one time step. The *Extern* signal is asserted only in the first step of the *Load* operation. Therefore, the logic expression that implements this signal is

$$Extern = I_0T_1$$

Done is asserted in the first step of *Load* and *Move*, as well as in the third step of *Add* and *Sub*. Hence

$$Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

The A_{in}, G_{in}, and G_{out} signals are asserted in the *Add* and *Sub* operations. A_{in} is asserted in step T_1, G_{in} is asserted in T_2, and G_{out} is asserted in T_3. The *AddSub* signal has to be set to 0 in the *Add* operation and to 1 in the *Sub* operation. This is achieved with the following logic expressions

$$A_{in} = (I_2 + I_3)T_1$$

$$G_{in} = (I_2 + I_3)T_2$$

$$G_{out} = (I_2 + I_3)T_3$$

$$AddSub = I_3$$

The values of $R0_{in}, \ldots, R3_{in}$ are determined using either the $X_0, \ldots, X_3$ signals or the $Y_0, \ldots, Y_3$ signals. In Table 7.3 these actions are indicated by writing either $R_{in} = X$ or $R_{in} = Y$. The meaning of $R_{in} = X$ is that $R0_{in} = X_0$, $R1_{in} = X_1$, and so on. Similarly, the values of $R0_{out}, \ldots, R3_{out}$ are specified using either $R_{out} = X$ or $R_{out} = Y$.

We will develop the expressions for $R0_{in}$ and $R0_{out}$ by examining Table 7.3 and then show how to derive the expressions for the other register control signals. The table shows that $R0_{in}$ is set to the value of X_0 in the first step of both the *Load* and *Move* operations and in the third step of both the *Add* and *Sub* operations, which leads to the expression

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

Similarly, $R0_{out}$ is set to the value of Y_0 in the first step of *Move*. It is set to X_0 in the first step of *Add* and *Sub* and to Y_0 in the second step of these operations, which gives

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

The expressions for $R1_{in}$ and $R1_{out}$ are the same as those for $R0_{in}$ and $R0_{out}$ except that X_1 and Y_1 are used in place of X_0 and Y_0. The expressions for $R2_{in}$, $R2_{out}$, $R3_{in}$, and $R3_{out}$ are derived in the same way.

The circuits shown in Figures 7.70 and 7.71, combined with the circuits represented by the above expressions, implement the control circuit in Figure 7.69.

Processors are extremely useful circuits that are widely used. We have presented only the most basic aspects of processor design. However, the techniques presented can be extended to design realistic processors, such as modern microprocessors. The interested reader can refer to books on computer organization for more details on processor design [1–2].

VHDL Code

In this section we give two different styles of VHDL code for describing the system in Figure 7.69. The first style uses tri-state buffers to represent the bus, and it gives the logic expressions shown above for the outputs of the control circuit. The second style of code uses multiplexers to represent the bus, and it uses CASE statements that correspond to Table 7.3 to describe the outputs of the control circuit.

VHDL code for an up-counter is shown in Figure 7.52. A modified version of this counter, named *upcount*, is shown in the code in Figure 7.72. It has a synchronous reset input, which is active high. In Figure 7.64 we defined the package named *components*, which provides component declarations for a number of subcircuits. In the VHDL code for the processor, we will use the *regn* and *trin* components listed in Figure 7.64, but not the *shiftr* component. We created a new package called *subccts* for use with the processor. The code is not shown here, but it includes component declarations for *regn* (Figure 7.61), *trin* (Figure 7.62), *upcount*, and *dec2to4* (Figure 6.30).

Complete code for the processor is given in Figure 7.73. In the architecture body, the statements labeled *counter* and *decT* instantiate the subcircuits in Figure 7.70. Note that we have assumed that the circuit has an active-high reset input, *Reset*, which is used to initialize the counter to 00. The statement Func <= F & Rx & Ry uses the concatenate operator to create the six-bit signal *Func*, which represents the inputs to the Function Register in Figure 7.71. The next statement instantiates the Function Register with the data inputs *Func* and

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clear, Clock  : IN      STD_LOGIC ;
               Q         : BUFFER  STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    upcount: PROCESS ( Clock )
    BEGIN
        IF (Clock'EVENT AND Clock = '1') THEN
            IF Clear = '1' THEN
                Q <= "00" ;
            ELSE
                Q <= Q + '1' ;
            END IF ;
        END IF;
    END PROCESS;
END Behavior ;
```

Figure 7.72 Code for a two-bit up-counter with synchronous reset.

the outputs *FuncReg*. The statements labeled *decI*, *decX*, and *decY* instantiate the decoders in Figure 7.71. Following these statements the previously derived logic expressions for the outputs of the control circuit are given. For $R0_{in}, \ldots, R3_{in}$ and $R0_{out}, \ldots, R3_{out}$, a GENERATE statement is used to produce the expressions.

At the end of the code, the tri-state buffers and registers in the processor are instantiated, and the adder/subtractor module is described using a selected signal assignment.

Using Multiplexers and CASE Statements

We showed in Figure 7.60 that a bus can be implemented using multiplexers, rather than tri-state buffers. VHDL code that describes the processor using this approach is shown in Figure 7.74. The same entity declaration given in Figure 7.73 can be used and is not shown in Figure 7.74. The code illustrates a different way of describing the control circuit in the processor. It does not give logic expressions for the signals *Extern*, *Done*, and so on, as we did in Figure 7.73. Instead, CASE statements are used to represent the information shown in Table 7.3. These statements are provided inside the process labeled *controlsignals*. Each control signal is first assigned the value 0, as a default. This is required because the CASE statements specify the values of the control signals only when they should be asserted, as we did in Table 7.3. As explained for Figure 7.35, when the value of a signal is not specified,

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
USE work.subccts.all ;

ENTITY proc IS
    PORT ( Data       : IN       STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Reset, w   : IN       STD_LOGIC ;
           Clock      : IN       STD_LOGIC ;
           F, Rx, Ry  : IN       STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           Done       : BUFFER   STD_LOGIC ;
           BusWires   : INOUT    STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END proc ;

ARCHITECTURE Behavior OF proc IS
    SIGNAL Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL Clear, High, AddSub : STD_LOGIC ;
    SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
    SIGNAL Count, Zero : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL T, I, X, Y : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Func, FuncReg : STD_LOGIC_VECTOR(1 TO 6) ;
BEGIN
    Zero <= "00" ; High <= '1' ;
    Clear <= Reset OR Done OR (NOT w AND T(0)) ;
    counter: upcount PORT MAP ( Clear, Clock, Count ) ;
    decT: dec2to4 PORT MAP ( Count, High, T );
    Func <= F & Rx & Ry ;
    FRin <= w AND T(0) ;
    functionreg: regn GENERIC MAP ( N => 6 )
                 PORT MAP ( Func, FRin, Clock, FuncReg ) ;
    decI: dec2to4 PORT MAP ( FuncReg(1 TO 2), High, I ) ;
    decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
    decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;
    Extern <= I(0) AND T(1) ;
    Done <= ((I(0) OR I(1)) AND T(1)) OR ((I(2) OR I(3)) AND T(3)) ;
    Ain <= (I(2) OR I(3)) AND T(1) ;
    Gin <= (I(2) OR I(3)) AND T(2) ;
    Gout <= (I(2) OR I(3)) AND T(3) ;
    AddSub <= I(3) ;

. . . continued in Part b.
```

Figure 7.73 Code for the processor (Part *a*).

```
    RegCntl:
    FOR k IN 0 TO 3 GENERATE
        Rin(k) <= ((I(0) OR I(1)) AND T(1) AND X(k)) OR
            ((I(2) OR I(3)) AND T(3) AND X(k)) ;
        Rout(k) <= (I(1) AND T(1) AND Y(k)) OR
            ((I(2) OR I(3)) AND ((T(1) AND X(k)) OR (T(2) AND Y(k)))) ;
    END GENERATE RegCntl ;
    tri_extern: trin PORT MAP ( Data, Extern, BusWires ) ;
    reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    tri0: trin PORT MAP ( R0, Rout(0), BusWires ) ;
    tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
    tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
    tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
    regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
    alu:
    WITH AddSub SELECT
        Sum <= A + BusWires WHEN '0',
               A − BusWires WHEN OTHERS ;
    regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
    triG: trin PORT MAP ( G, Gout, BusWires ) ;
END Behavior ;
```

Figure 7.73 Code for the processor (Part *b*).

the signal retains its current value. This implied memory results in a feedback connection in the synthesized circuit. We avoid this problem by providing the default value of 0 for each of the control signals involved in the CASE statements.

In Figure 7.73 the statements labeled *decT* and *decI* are used to decode the *Count* signal and the stored values of the *F* input, respectively. The *decT* decoder has the outputs $T_0, \ldots, T_3$, and *decI* produces $I_0, \ldots, I_3$. In Figure 7.74 these two decoders are not used, because they do not serve a useful purpose in this code. Instead, the signals T and I are defined as two-bit signals, which are used in the CASE statements. The code sets T to the value of *Count*, while I is set to the value of the two left-most bits in the Function Register, which correspond to the stored values of the input F.

There are two nested levels of CASE statements. The first one enumerates the possible values of T. For each WHEN clause in this CASE statement, which represents a column in Table 7.3, there is a nested CASE statement that enumerates the four values of I. As indicated by the comments in the code, the nested CASE statements correspond exactly to the information given in Table 7.3.

```
ARCHITECTURE Behavior OF proc IS
    SIGNAL X, Y, Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL Clear, High, AddSub : STD_LOGIC ;
    SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
    SIGNAL Count, Zero, T, I : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Func, FuncReg, Sel : STD_LOGIC_VECTOR(1 TO 6) ;
BEGIN
    Zero <= "00" ; High <= '1' ;
    Clear <= Reset OR Done OR (NOT w AND NOT T(1) AND NOT T(0)) ;
    counter: upcount PORT MAP ( Clear, Clock, Count ) ;
    T <= Count ;
    Func <= F & Rx & Ry ;
    FRin <= w AND NOT T(1) AND NOT T(0) ;
    functionreg: regn GENERIC MAP ( N => 6 )
                    PORT MAP ( Func, FRin, Clock, FuncReg ) ;
    I <= FuncReg(1 TO 2) ;
    decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
    decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;

    controlsignals: PROCESS ( T, I, X, Y )
    BEGIN
        Extern <= '0' ; Done <= '0' ; Ain <= '0' ; Gin <= '0' ;
        Gout <= '0' ; AddSub <= '0' ; Rin <= "0000" ; Rout <= "0000" ;
        CASE T IS    WHEN "00" => -- no signals asserted in time step T0
            WHEN "01" => -- define signals asserted in time step T1
                CASE I IS
                    WHEN "00" => -- Load
                        Extern <= '1' ; Rin <= X ; Done <= '1' ;
                    WHEN "01" => -- Move
                        Rout <= Y ; Rin <= X ; Done <= '1' ;
                    WHEN OTHERS => -- Add, Sub
                        Rout <= X ; Ain <= '1' ;
                END CASE ;

. . . continued in Part b
```

Figure 7.74 Alternative code for the processor (Part *a*).

At the end of Figure 7.74, the bus is described using a selected signal assignment. This statement represents multiplexers that place the appropriate data onto *BusWires*, depending on the values of R_{out}, G_{out}, and *Extern*.

The circuits synthesized from the code in Figures 7.73 and 7.74 are functionally equivalent. The style of code in Figure 7.74 has the advantage that it does not require the manual

```
                WHEN ”10” => - - define signals asserted in time step T2
                    CASE I IS
                        WHEN ”10” => - - Add
                            Rout <= Y ; Gin <= ’1’ ;
                        WHEN ”11” => - - Sub
                            Rout <= Y ; AddSub <= ’1’ ; Gin <= ’1’ ;
                        WHEN OTHERS => - - Load, Move
                    END CASE ;
                WHEN OTHERS => - - define signals asserted in time step T3
                    CASE I IS
                        WHEN ”00” => - - Load
                        WHEN ”01” => - - Move
                        WHEN OTHERS => - - Add, Sub
                            Gout <= ’1’ ; Rin <= X ; Done <= ’1’ ;
                    END CASE ;
            END CASE ;
        END PROCESS ;
        reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
        reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
        reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
        reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
        regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
        alu: WITH AddSub SELECT
            Sum <= A + BusWires WHEN ’0’,
                   A − BusWires WHEN OTHERS ;
        regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
        Sel <= Rout & Gout & Extern ;
        WITH Sel SELECT
            BusWires <= R0 WHEN ”100000”,
                        R1 WHEN ”010000”,
                        R2 WHEN ”001000”,
                        R3 WHEN ”000100”,
                        G WHEN  ”000010”,
                        Data WHEN OTHERS ;
    END Behavior ;
```

Figure 7.74 Alternative code for the processor (Part *b*).

effort of analyzing Table 7.3 to generate the logic expressions for the control signals used for Figure 7.73. By using the style of code in Figure 7.74, these expressions are produced automatically by the VHDL compiler as a result of analyzing the CASE statements. The style of code in Figure 7.74 is less prone to careless errors. Also, using this style of code it would be straightforward to provide additional capabilities in the processor, such as adding other operations.

We synthesized a circuit to implement the code in Figure 7.74 in a chip. Figure 7.75 gives an example of the results of a timing simulation. Each clock cycle in which $w = 1$ in this timing diagram indicates the start of an operation. In the first such operation, at 250 ns in the simulation time, the values of both inputs F and Rx are 00. Hence the operation corresponds to "*Load* $R0$, *Data*." The value of *Data* is 2A, which is loaded into $R0$ on the next positive clock edge. The next operation loads 55 into register $R1$, and the subsequent operation loads 22 into $R2$. At 850 ns the value of the input F is 10, while $Rx = 01$ and $Ry = 00$. This operation is "*Add* $R1$, $R0$." In the following clock cycle, the contents of $R1$ (55) appear on the bus. This data is loaded into register A by the clock edge at 950 ns, which also results in the contents of $R0$ (2A) being placed on the bus. The adder/subtractor module generates the correct sum (7F), which is loaded into register G at 1050 ns. After this clock edge the new contents of G (7F) are placed on the bus and loaded into register $R1$ at 1150 ns. Two more operations are shown in the timing diagram. The one at 1250 ns ("*Move* $R3$, $R1$") copies the contents of $R1$ (7F) into $R3$. Finally, the operation starting at 1450 ns ("*Sub* $R3$, $R2$") subtracts the contents of $R2$ (22) from the contents of $R3$ (7F), producing the correct result, $7F - 22 = 5D$.

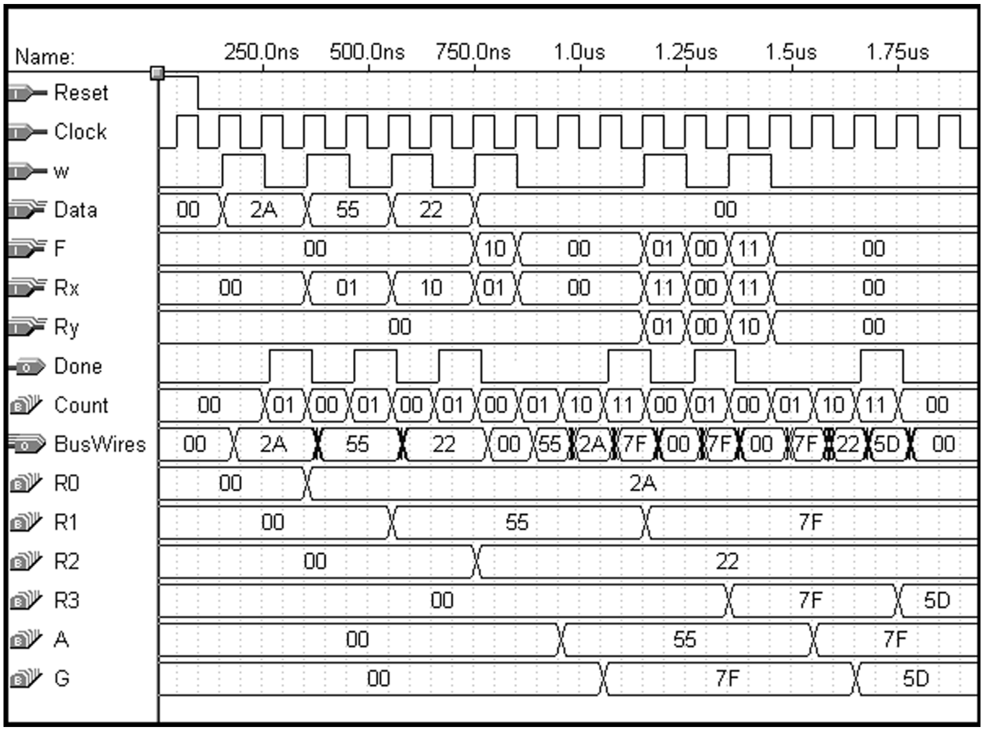


Figure 7.75 Timing simulation for the VHDL code in Figure 7.74.

7.14.3 Reaction Timer

We showed in Chapter 3 that electronic devices operate at remarkably fast speeds, with the typical delay through a logic gate being less than 1 ns. In this example we use a logic circuit to measure the speed of a much slower type of device—a person.

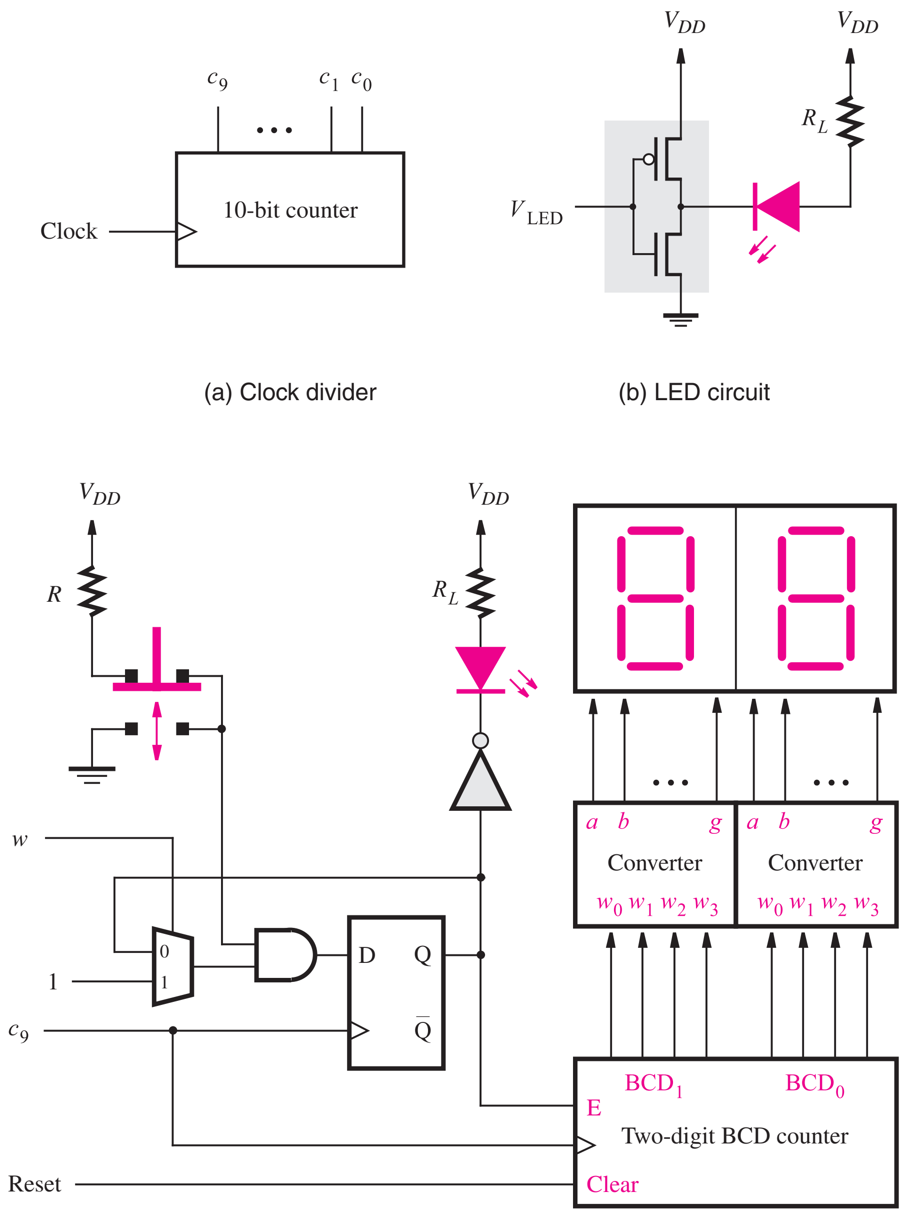
We will design a circuit that can be used to measure the reaction time of a person to a specific event. The circuit turns on a small light, called a *light-emitting diode (LED)*. In response to the LED being turned on, the person attempts to press a switch as quickly as possible. The circuit measures the elapsed time from when the LED is turned on until the switch is pressed.

To measure the reaction time, a clock signal with an appropriate frequency is needed. In this example we use a 100 Hz clock, which measures time at a resolution of 1/100 of a second. The reaction time can then be displayed using two digits that represent fractions of a second from 00/100 to 99/100.

Digital systems often include high-frequency clock signals to control various subsystems. In this case assume the existence of an input clock signal with the frequency 102.4 kHz. From this signal we can derive the required 100 Hz signal by using a counter as a *clock divider*. A timing diagram for a four-bit counter is given in Figure 7.22. It shows that the least-significant bit output, Q_0, of the counter is a periodic signal with half the frequency of the clock input. Hence we can view Q_0 as dividing the clock frequency by two. Similarly, the Q_1 output divides the clock frequency by four. In general, output Q_i in an n-bit counter divides the clock frequency by 2^{i+1}. In the case of our 102.4 kHz clock signal, we can use a 10-bit counter, as shown in Figure 7.76*a*. The counter output c_9 has the required 100 Hz frequency because $102400\text{ Hz}/1024 = 100\text{ Hz}$.

The reaction timer circuit has to be able to turn an LED on and off. The graphical symbol for an LED is shown in blue in Figure 7.76*b*. Small blue arrows in the symbol represent the light that is emitted when the LED is turned on. The LED has two terminals: the one on the left in the figure is the *cathode*, and the terminal on the right is the *anode*. To turn the LED on, the cathode has to be set to a lower voltage than the anode, which causes a current to flow through the LED. If the voltages on its two terminals are equal, the LED is off.

Figure 7.76*b* shows one way to control the LED, using an inverter. If the input voltage $V_{LED} = 0$, then the voltage at the cathode is equal to V_{DD}; hence the LED is off. But if $V_{LED} = V_{DD}$, the cathode voltage is 0 V and the LED is on. The amount of current that flows is limited by the value of the resistor R_L. This current flows through the LED and the NMOS transistor in the inverter. Since the current flows *into* the inverter, we say that the inverter *sinks* the current. The maximum current that a logic gate can sink without sustaining permanent damage is usually called I_{OL}, which stands for the “maximum current when the output is low.” The value of R_L is chosen such that the current is less than I_{OL}. As an example assume that the inverter is implemented inside a PLD device. The typical value of I_{OL}, which would be specified in the data sheet for the PLD, is about 12 mA. For $V_{DD} = 5$ V, this leads to $R_L \approx 450\ \Omega$ because $5\ V/450\ \Omega = 11$ mA (there is actually a small voltage drop across the LED when it is turned on, but we ignore this for simplicity). The amount of light emitted by the LED is proportional to the current flow. If 11 mA is insufficient, then the inverter should be implemented in a



(a) Clock divider

(b) LED circuit

(c) Push-button switch, LED, and 7-segment displays

Figure 7.76 A reaction-timer circuit.

buffer chip, like those described in section 3.5, because buffers provide a higher value of I_{OL}.

The complete reaction-timer circuit is illustrated in Figure 7.76*c*, with the inverter from part (*b*) shaded in grey. The graphical symbol for a push-button switch is shown in the top left of the diagram. The switch normally makes contact with the top terminals, as depicted in the figure. When depressed, the switch makes contact with the bottom terminals; when released, it automatically springs back to the top position. In the figure the switch is connected such that it normally produces a logic value of 1, and it produces a 0 pulse when pressed.

When depressed, the push-button switch causes the D flip-flop to be synchronously reset. The output of this flip-flop determines whether the LED is on or off, and it also provides the count enable input to a two-digit BCD counter. As discussed in section 7.11, each digit in a BCD counter has four bits that take the values 0000 to 1001. Thus the counting sequence can be viewed as decimal numbers from 00 to 99. A circuit for the BCD counter is given in Figure 7.28. In Figure 7.76*c* both the flip-flop and the counter are clocked by the c_9 output of the clock divider in part (*a*) of the figure. The intended use of the reaction-timer circuit is to first depress the switch to turn off the LED and disable the counter. Then the *Reset* input is asserted to clear the contents of the counter to 00. The input w normally has the value 0, which keeps the flip-flop cleared and prevents the count value from changing. The reaction test is initiated by setting $w = 1$ for one c_9 clock cycle. After the next positive edge of c_9, the flip-flop output becomes a 1, which turns on the LED. We assume that w returns to 0 after one clock cycle, but the flip-flop output remains at 1 because of the 2-to-1 multiplexer connected to the D input. The counter is then incremented every 1/100 of a second. Each digit in the counter is connected through a code converter to a 7-segment display, which we described in the discussion for Figure 6.25. When the user depresses the switch, the flip-flop is cleared, which turns off the LED and stops the counter. The two-digit display shows the elapsed time to the nearest 1/100 of a second from when the LED was turned on until the user was able to respond by depressing the switch.

VHDL Code

To describe the circuit in Figure 7.76*c* using VHDL code, we can make use of subcircuits for the BCD counter and the 7-segment code converter. The code for the latter subcircuit is given in Figure 6.47 and is not repeated here. Code for the BCD counter, which represents the circuit in Figure 7.28, is shown in Figure 7.77. The two-digit BCD output is represented by the 2 four-bit signals *BCD*1 and *BCD*0. The *Clear* input is used to provide a synchronous reset for both digits in the counter. If $E = 1$, the count value is incremented on the positive clock edge, and if $E = 0$, the count value is unchanged. Each digit can take the values from 0000 to 1001.

Figure 7.78 gives the code for the reaction timer. The input signal *Pushn* represents the value produced by the push-button switch. The output signal *LEDn* represents the output of the inverter that is used to control the LED. The two 7-segment displays are controlled by the seven-bit signals *Digit*1 and *Digit*0.

In Figure 7.56 we showed how a register, R, can be designed with a control signal R_{in}. If $R_{in} = 1$ data is loaded into the register on the active clock edge and if $R_{in} = 0$, the stored contents of the register are not changed. The flip-flop in Figure 7.76 is used in the same

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY BCDcount IS
    PORT ( Clock        : IN      STD_LOGIC ;
           Clear, E     : IN      STD_LOGIC ;
           BCD1, BCD0   : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END BCDcount ;

ARCHITECTURE Behavior OF BCDcount IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            IF Clear = '1' THEN
                BCD1 <= "0000" ; BCD0 <= "0000" ;
            ELSIF E = '1' THEN
                IF BCD0 = "1001" THEN
                    BCD0 <= "0000" ;
                    IF BCD1 = "1001" THEN
                        BCD1 <= "0000";
                    ELSE
                        BCD1 <= BCD1 + '1' ;
                    END IF ;
                ELSE
                    BCD0 <= BCD0 + '1' ;
                END IF ;
            END IF ;
        END IF;
    END PROCESS;
END Behavior ;
```

Figure 7.77 Code for the two-digit BCD counter in Figure 7.28.

way. If $w = 1$, the flip-flop is loaded with the value 1, but if $w = 0$ the stored value in the flip-flop is not changed. This circuit is described by the process labeled *flipflop* in Figure 7.78, which also includes a synchronous reset input. We have chosen to use a synchronous reset because the flip-flop output is connected to the enable input E on the BCD counter. As we know from the discussion in section 7.3, it is important that all signals connected to flip-flops meet the required setup and hold times. The push-button switch can be pressed at any time and is not synchronized to the c_9 clock signal. By using a synchronous reset for the flip-flop in Figure 7.76, we avoid possible timing problems in the counter.

The flip-flop output is called *LED*, which is inverted to produce the *LEDn* signal that controls the LED. In the device used to implement the circuit, *LEDn* would be generated by

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reaction IS
    PORT ( c9, Reset        : IN      STD_LOGIC ;
           w, Pushn         : IN      STD_LOGIC ;
           LEDn             : OUT     STD_LOGIC ;
           Digit1, Digit0   : BUFFER  STD_LOGIC_VECTOR(1 TO 7) ) ;
END reaction ;

ARCHITECTURE Behavior OF reaction IS
    COMPONENT BCDcount
        PORT ( Clock        : IN      STD_LOGIC ;
               Clear, E     : IN      STD_LOGIC ;
               BCD1, BCD0   : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
    END COMPONENT ;
    COMPONENT seg7
        PORT ( bcd   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
               leds  : OUT   STD_LOGIC_VECTOR(1 TO 7) ) ;
    END COMPONENT ;
    SIGNAL LED : STD_LOGIC ;
    SIGNAL BCD1, BCD0 : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
BEGIN
    flipflop: PROCESS
    BEGIN
        WAIT UNTIL c9'EVENT AND c9 = '1' ;
        IF Pushn = '0' THEN
            LED <= '0' ;
        ELSIF w = '1' THEN
            LED <= '1' ;
        END IF ;
    END PROCESS ;

    LEDn <= NOT LED ;
    counter: BCDcount PORT MAP ( c9, Reset, LED, BCD1, BCD0 ) ;
    seg1 : seg7 PORT MAP ( BCD1, Digit1 ) ;
    seg0 : seg7 PORT MAP ( BCD0, Digit0 ) ;
END Behavior ;
```

Figure 7.78 Code for the reaction timer.

a buffer that is connected to an output pin on the chip package. If a PLD is used, this buffer has the associated value of $I_{OL} = 12$ mA that we mentioned earlier. At the end of Figure 7.78, the BCD counter and 7-segment code converters are instantiated as subcircuits.

A simulation of the reaction-timer circuit implemented in a chip is shown in Figure 7.79. Initially, *Pushn* is set to 0 to simulate depressing the switch to turn off the LED, and

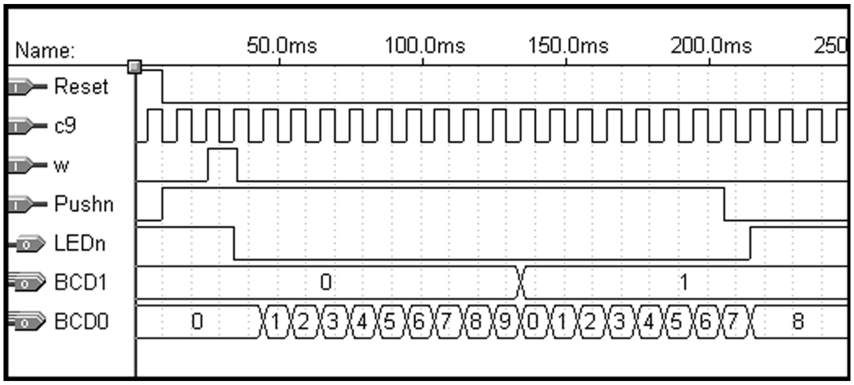


Figure 7.79 Simulation of the reaction-timer circuit.

then *Pushn* returns to 1. Also, *Reset* is asserted to clear the counter. When w changes to 1, the circuit sets *LEDn* to 0, which represents the LED being turned on. After some amount of time, the switch will be depressed. In the simulation we arbitrarily set *Pushn* to 0 after 18 c_9 clock cycles. Thus this choice represents the case when the person's reaction time is about 0.18 seconds. In human terms this duration is a very short time; for electronic circuits it is a very long time. An inexpensive personal computer can perform tens of millions of operations in 0.18 seconds!

7.14.4 Register Transfer Level (RTL) Code

At this point, we have introduced most of the VHDL constructs that are needed for synthesis. Most of our examples give behavioral code, utilizing IF-THEN-ELSE statements, CASE statements, FOR loops, and so on. It is possible to write behavioral code in a style that resembles a computer program, in which there is a complex flow of control with many loops and branches. With such code, sometimes called *high-level* behavioral code, it is difficult to relate the code to the final hardware implementation; it may even be difficult to predict what circuit a high-level synthesis tool will produce. In this book we do not use the high-level style of code. Instead, we present VHDL code in such a way that the code can be easily related to the circuit that is being described. Most design modules presented are fairly small, to facilitate simple descriptions. Larger designs are built by interconnecting the smaller modules. This approach is usually referred to as the *register-transfer level* (RTL) style of code. It is the most popular design method used in practice. RTL code is characterized by a straightforward flow of control through the code; it comprises well-understood subcircuits that are connected together in a simple way.

7.15 Timing Analysis of Flip-flop Circuits

In Figure 7.15 we showed the timing parameters associated with a D flip-flop. A simple circuit that uses this flip-flop is given in Figure 7.80. We wish to calculate the maximum clock frequency, F_{max}, for which this circuit will operate properly, and also determine if the circuit suffers from any hold time violations. In the literature, this type of analysis of circuits is usually called *timing analysis*. We will assume that the flip-flop timing parameters have the values $t_{su} = 0.6$ ns, $t_h = 0.4$ ns, and $0.8 \text{ ns} \leq t_{cQ} \leq 1.0$ ns. A range of minimum and maximum values is given for t_{cQ} because, as we mentioned in section 7.4.4, this is the usual way of dealing with variations in delay that exist in integrated circuit chips.

To calculate the minimum period of the clock signal, $T_{min} = 1/F_{max}$, we need to consider all paths in the circuit that start and end at flip-flops. In this simple circuit there is only one such path, which starts when data is loaded into the flip-flop by a positive clock edge, propagates to the Q output after the t_{cQ} delay, propagates through the NOT gate, and finally must meet the setup requirement at the D input. Therefore

$$T_{min} = t_{cQ} + t_{NOT} + t_{su}$$

Since we are interested in the longest delay for this calculation, the maximum value of t_{cQ} should be used. For the calculation of t_{NOT} we will assume that the delay through any logic gate can be calculated as $1 + 0.1k$, where k is the number of inputs to the gate. For a NOT gate this gives 1.1 ns, which leads to

$$T_{min} = 1.0 + 1.1 + 0.6 = 2.7 \text{ ns}$$
$$F_{max} = 1/2.7 \text{ ns} = 370.37 \text{ MHz}$$

It is also necessary to check if there are any hold time violations in the circuit. In this case we need to examine the shortest possible delay from a positive clock edge to a change in the value of the D input. The delay is given by $t_{cQ} + t_{NOT} = 0.8 + 1.1 = 1.9$ ns. Since $1.9 \text{ ns} > t_h = 0.4$ ns there is no hold time violation.

As another example of timing analysis of flip-flop circuits, consider the counter circuit shown in Figure 7.81. We wish to calculate the maximum clock frequency for which this circuit will operate properly assuming the same flip-flop timing parameters as we did for

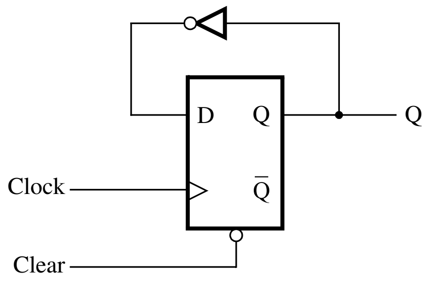


Figure 7.80 A simple flip-flop circuit.

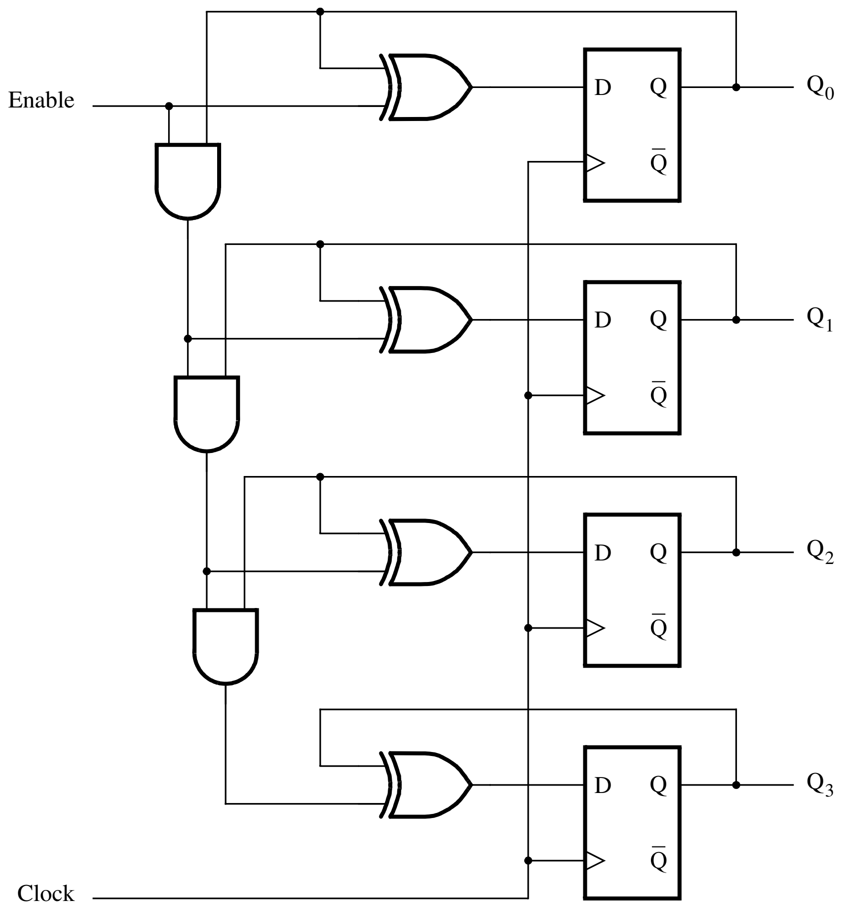


Figure 7.81 A 4-bit counter.

Figure 7.80. We will again assume that the propagation delay through a logic gate can be calculated as $1 + 0.1k$.

There are many paths in this circuit that start and end at flip-flops. The longest such path starts at flip-flop Q_0 and ends at flip-flop Q_3. The longest path in a circuit is often called a *critical* path. The delay of the critical path includes the clock-to-Q delay of flip-flop Q_0, the propagation delay through three AND gates, and one XOR-gate delay. We must also account for the setup time of flip-flop Q_3. This gives

$$T_{min} = t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su}$$

Using the maximum value of t_{cQ} gives

$$T_{min} = 1.0 + 3(1.2) + 1.2 + 0.6 \text{ ns} = 6.4 \text{ ns}$$
$$F_{max} = 1/6.4 \text{ ns} = 156.25 \text{ MHz}$$

The shortest paths through the circuit are from each flip-flop to itself, through an XOR gate. The minimum delay along each such path is $t_{cQ} + t_{XOR} = 0.8 + 1.2 = 2.0$ ns. Since 2.0 ns $> t_h = 0.4$ ns there are no hold time violations.

In the above analysis we assumed that the clock signal arrived at exactly the same time at all four flip-flops. We will now repeat this analysis assuming that the clock signal still arrives at flip-flops Q_0, Q_1, and Q_2 simultaneously, but that there is a delay in the arrival of the clock signal at flip-flop Q_3. Such a variation in the arrival time of a clock signal at different flip-flops is called *clock skew*, t_{skew}, and can be caused by a number of factors.

In Figure 7.81 the critical path through the circuit is from flip-flop Q_0 to Q_3. However, the clock skew at Q_3 has the effect of reducing this delay, because it provides additional time before data is loaded into this flip-flop. Taking a clock skew of 1.5 ns into account, the delay of the path from flip-flop Q_0 to Q_3 is given by $t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su} - t_{skew} = 6.4 - 1.5$ ns $= 4.9$ ns. There is now a different critical path through the circuit, which starts at flip-flop Q_0 and ends at Q_2. The delay of this path gives

$$\begin{aligned} T_{min} &= t_{cQ} + 2(t_{AND}) + t_{XOR} + t_{su} \\ &= 1.0 + 2(1.2) + 1.2 + 0.6 \text{ ns} \\ &= 5.2 \text{ ns} \\ F_{max} &= 1/5.2 \text{ ns} = 192.31 \text{ MHz} \end{aligned}$$

In this case the clock skew results in an increase in the circuit's maximum clock frequency. But if the clock skew had been negative, which would be the case if the clock signal arrived earlier at flip-flop Q_3 than at other flip-flops, then the result would have been a reduced F_{max}.

Since the loading of data into flip-flop Q_3 is delayed by the clock skew, it has the effect of increasing the hold time requirement of this flip-flop to $t_h + t_{skew}$, for all paths that end at Q_3 but start at Q_0, Q_1, or Q_2. The shortest such path in the circuit is from flip-flop Q_2 to Q_3 and has the delay $t_{cQ} + t_{AND} + t_{XOR} = 0.8 + 1.2 + 1.2 = 3.2$ ns. Since 3.2 ns $> t_h + t_{skew} = 1.9$ ns there is no hold time violation.

If we repeat the above hold time analysis for clock skew values $t_{skew} \geq 3.2 - t_h = 2.8$ ns, then hold time violations will exist. Thus, if $t_{skew} \geq 2.8$ ns the circuit will not work reliably at any clock frequency. Due to the complications in circuit timing that arise in the presence of clock skew, a good digital circuit design approach is to ensure that the clock signal reaches all flip-flops with the smallest possible skew. We discuss clock synchronization issues in section 10.3.

7.16 Concluding Remarks

In this chapter we have presented circuits that serve as basic storage elements in digital systems. These elements are used to build larger units such as registers, shift registers, and counters. Many other texts that deal with this material are available [3–11]. We have illustrated how circuits with flip-flops can be described using VHDL code. More information on VHDL can be found in [12–17]. In the next chapter a more formal method for designing circuits with flip-flops will be presented.

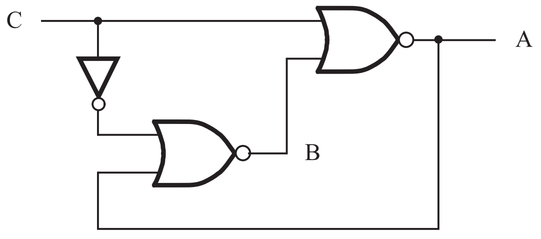
7.17 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

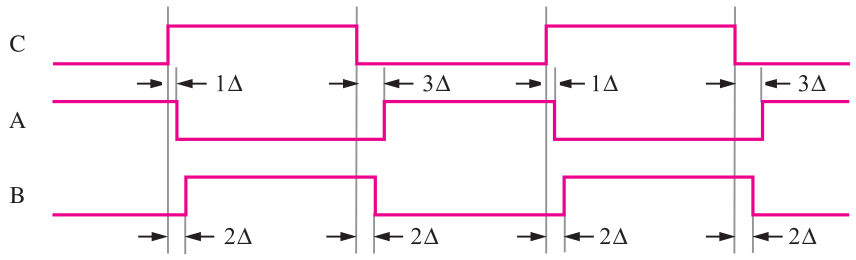
Example 7.13 **Problem:** Consider the circuit in Figure 7.82*a*. Assume that the input C is driven by a square wave signal with a 50% duty cycle. Draw a timing diagram that shows the waveforms at points A and B. Assume that the propagation delay through each gate is Δ seconds.

Solution: The timing diagram is shown in Figure 7.82*b*.

Example 7.14 **Problem:** Determine the functional behavior of the circuit in Figure 7.83. Assume that input w is driven by a square wave signal.



(a) Circuit



(b) Timing diagram

Figure 7.82 Circuit for Example 7.13.

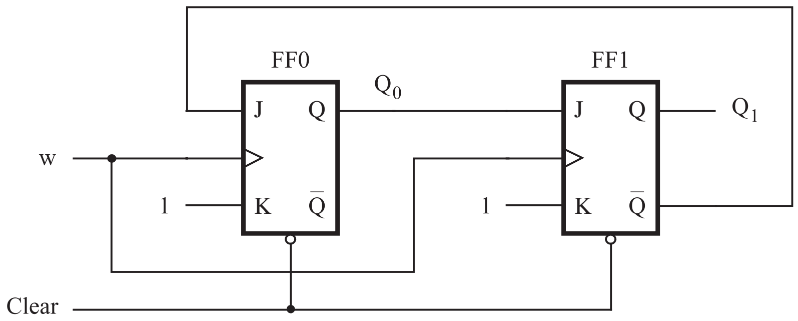


Figure 7.83 Circuit for Example 7.14.

Time interval	FF0			FF1		
	J_0	K_0	Q_0	J_1	K_1	Q_1
Clear	1	1	0	0	1	0
t_1	1	1	1	1	1	0
t_2	0	1	0	0	1	1
t_3	1	1	0	0	1	0
t_4	1	1	1	1	1	0

Figure 7.84 Summary of the behavior of the circuit in Figure 7.83.

Solution: When both flip-flops are cleared, their outputs are $Q_0 = Q_1 = 0$. After the Clear input goes high, each pulse on the *w* input will cause a change in the flip-flops as indicated in Figure 7.84. Note that the figure shows the state of the signals after the changes caused by the rising edge of a pulse have taken place.

In consecutive time intervals the values of Q_1 Q_0 are 00, 01, 10, 00, 01, and so on. Therefore, the circuit generates the counting sequence: 0, 1, 2, 0, 1, and so on. Hence, the circuit is a modulo-3 counter.

Example 7.15

Problem: Figure 7.70 shows a circuit that generates four timing control signals T_0, T_1, T_2, and T_3. Design a circuit that generates six such signals, T_0 to T_5.

Solution: The scheme of Figure 7.70 can be extended by using a modulo-6 counter, given in Figure 7.26, and a decoder that produces the six timing signals. A simpler alternative is possible by using a Johnson counter. Using three D-type flip-flops in a structure depicted in Figure 7.30, we can generate six patterns of bits $Q_0Q_1Q_2$ as shown in Figure 7.85. Then,

Clock cycle	Q_0	Q_1	Q_2	Control signal
0	0	0	0	$T_0 = \overline{Q}_0\overline{Q}_2$
1	1	0	0	$T_1 = Q_0\overline{Q}_1$
2	1	1	0	$T_2 = Q_1\overline{Q}_2$
3	1	1	1	$T_3 = Q_0Q_2$
4	0	1	1	$T_4 = \overline{Q}_0Q_1$
5	0	0	1	$T_5 = \overline{Q}_1Q_2$

Figure 7.85 Timing signals for Example 7.15.

using only six more two-input AND gates, as shown in the figure, we can obtain the desired signals. Note that the patterns $Q_0Q_1Q_2$ equal to 010 and 101 cannot occur in the Johnson counter, so these cases are treated as don't care conditions.

Example 7.16 **Problem:** Design a circuit that can be used to control a vending machine. The circuit has five inputs: Q (quarter), D (dime), N (nickel), *Coin*, and *Resetn*. When a coin is deposited in the machine, a coin-sensing mechanism generates a pulse on the appropriate input (Q, D, or N). To signify the occurrence of the event, the mechanism also generates a pulse on the line *Coin*. The circuit is reset by using the *Resetn* signal (active low). When at least 30 cents has been deposited, the circuit activates its output, Z. No change is given if the amount exceeds 30 cents.

Design the required circuit by using the following components: a six-bit adder, a six-bit register, and any number of AND, OR, and NOT gates.

Solution: Figure 7.86 gives a possible circuit. The value of each coin is represented by a corresponding five-bit number. It is added to the current total, which is held in register *S*. The required output is

$$Z = s_5 + s_4s_3s_2s_1$$

The register is clocked by the negative edge of the *Coin* signal. This allows for a propagation delay through the adder, and ensures that a correct sum will be placed into the register.

In Chapter 9 we will show how this type of control circuit can be designed using a more structured approach.

Example 7.17 **Problem:** Write VHDL code to implement the circuit in Figure 7.86.

Solution: Figure 7.87 gives the desired code.

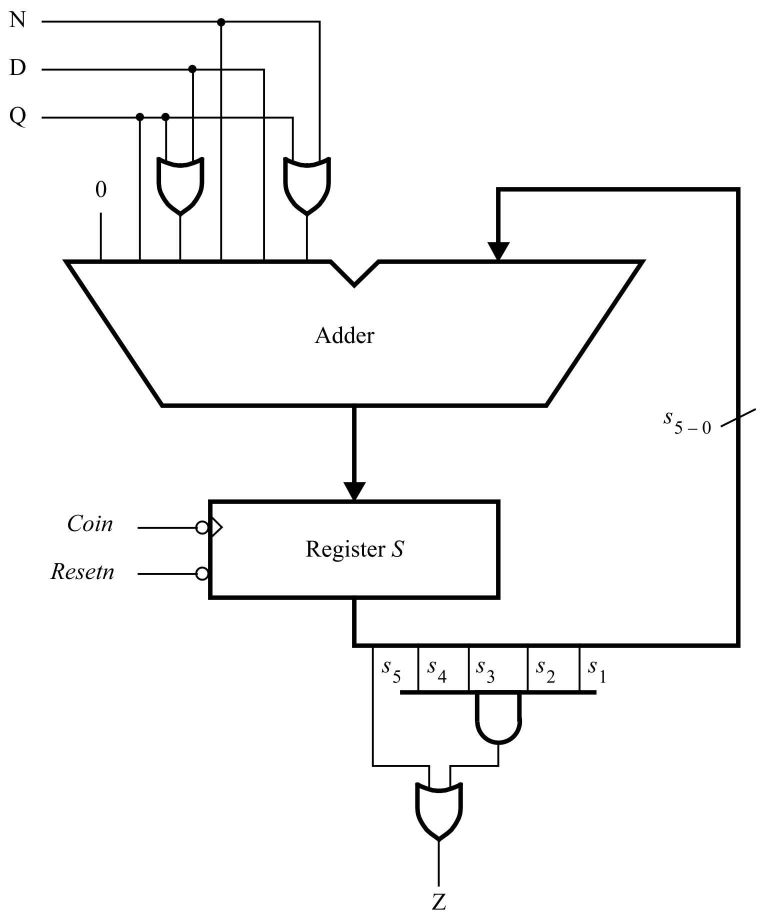


Figure 7.86 Circuit for Example 7.16.

Example 7.18

Problem: In section 7.15 we presented a timing analysis for the counter circuit in Figure 7.81. Redesign this circuit to reduce the logic delay between flip-flops, so that the circuit can operate at a higher maximum clock frequency.

Solution: As we showed in section 7.15, the performance of the counter circuit is limited by the delay through its cascaded AND gates. To increase the circuit's performance we can refactor the AND gates as illustrated in Figure 7.88. The longest delay path in this redesigned circuit, which starts at flip-flop Q_0 and ends at Q_3, provides the minimum clock period

$$\begin{aligned} T_{min} &= t_{cQ} + t_{AND} + t_{XOR} + t_{su} \\ &= 1.0 + 1.4 + 1.2 + 0.6 \text{ ns} = 4.2 \text{ ns} \end{aligned}$$

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY vend IS
    PORT ( N, D, Q, Resetn, Coin  : IN    STD_LOGIC ;
           Z                      : OUT  STD_LOGIC ) ;
END vend ;

ARCHITECTURE Behavior OF vend IS
    SIGNAL X:  STD_LOGIC_VECTOR(4 DOWNTO 0) ;
    SIGNAL S:  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
BEGIN
    X(0) <= N OR Q ;
    X(1) <= D ;
    X(2) <= N ;
    X(3) <= D OR Q ;
    X(4) <= Q ;
    PROCESS ( Resetn, Coin )
    BEGIN
        IF Resetn = '0' THEN
            S <= "000000" ;
        ELSIF Coin'EVENT AND Coin = '0' THEN
            S <= ('0' & X) + S ;
        END IF ;
    END PROCESS ;
    Z <= S(5) OR (S(4) AND S(3) AND S(2) AND S(1)) ;
END Behavior ;
```

Figure 7.87 Code for Example 7.17.

The redesigned counter has a maximum clock frequency of $F_{max} = 1/4.2$ ns $= 238.1$ MHz, compared to the result for the original counter, which was 156.25 MHz.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

7.1 Consider the timing diagram in Figure P7.1. Assuming that the *D* and *Clock* inputs shown are applied to the circuit in Figure 7.12, draw waveforms for the Q_a, Q_b, and Q_c signals.

7.2 Can the circuit in Figure 7.3 be modified to implement an SR latch? Explain your answer.

7.3 Figure 7.5 shows a latch built with NOR gates. Draw a similar latch using NAND gates. Derive its characteristic table and show its timing diagram.

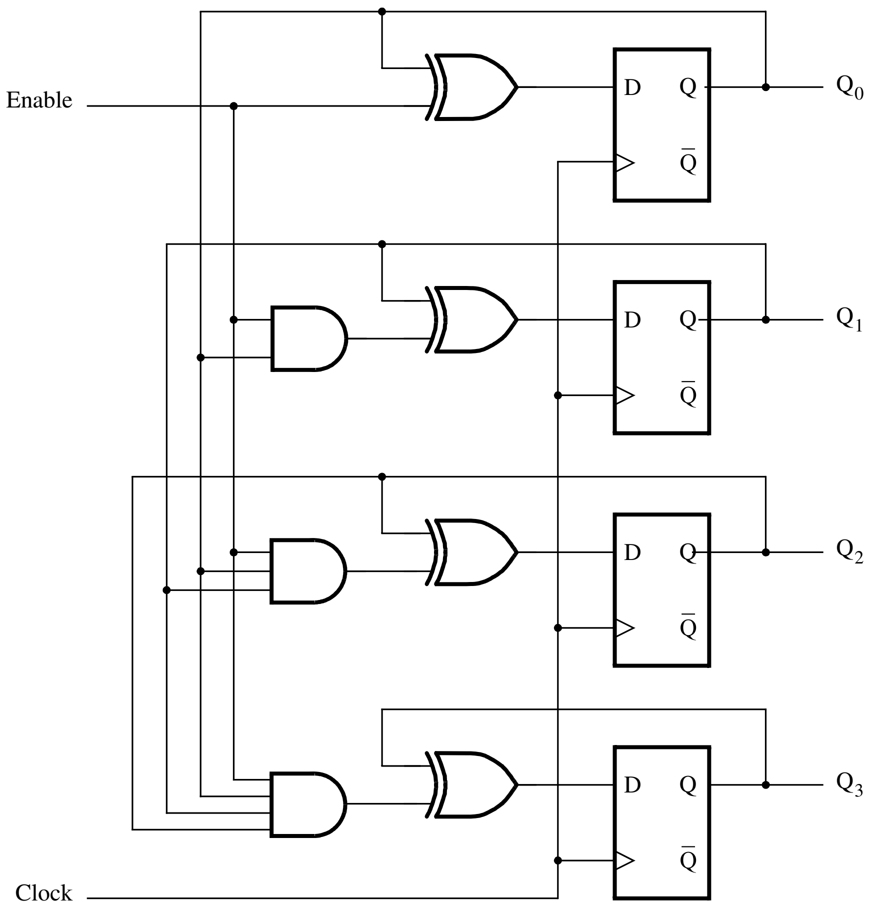


Figure 7.88 A faster 4-bit counter.

***7.4** Show a circuit that implements the gated SR latch using NAND gates only.

7.5 Given a 100-MHz clock signal, derive a circuit using D flip-flops to generate 50-MHz and 25-MHz clock signals. Draw a timing diagram for all three clock signals, assuming reasonable delays.

***7.6** An SR flip-flop is a flip-flop that has set and reset inputs like a gated SR latch. Show how an SR flip-flop can be constructed using a D flip-flop and other logic gates.

7.7 The gated SR latch in Figure 7.6*a* has unpredictable behavior if the S and R inputs are both equal to 1 when the *Clk* changes to 0. One way to solve this problem is to create a *set-dominant* gated SR latch in which the condition $S = R = 1$ causes the latch to be set to 1. Design a set-dominant gated SR latch and show the circuit.

7.8 Show how a JK flip-flop can be constructed using a T flip-flop and other logic gates.

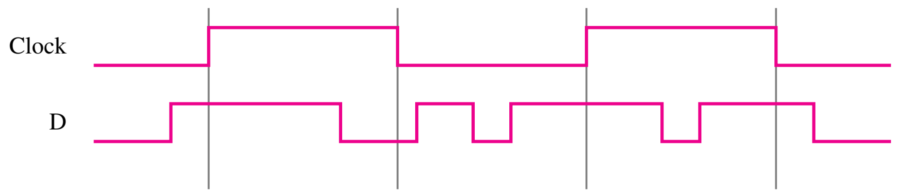


Figure P7.1 Timing diagram for Problem 7.1.

***7.9** Consider the circuit in Figure P7.2. Assume that the two NAND gates have much longer (about four times) propagation delay than the other gates in the circuit. How does this circuit compare with the circuits that we discussed in this chapter?

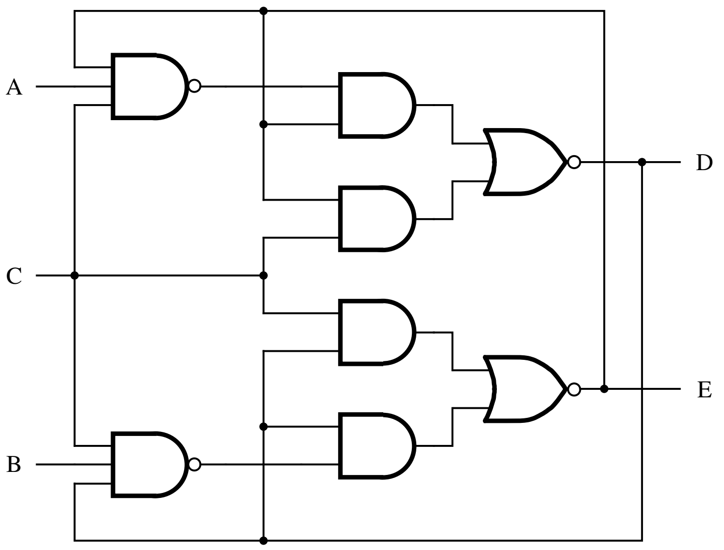


Figure P7.2 Circuit for Problem 7.9.

7.10 Write VHDL code that represents a T flip-flop with an asynchronous clear input. Use behavioral code, rather than structural code.

7.11 Write VHDL code that represents a JK flip-flop. Use behavioral code, rather than structural code.

7.12 Synthesize a circuit for the code written for problem 7.11 by using your CAD tools. Simulate the circuit and show a timing diagram that verifies the desired functionality.

7.13 A universal shift register can shift in both the left-to-right and right-to-left directions, and it has parallel-load capability. Draw a circuit for such a shift register.

7.14 Write VHDL code for a universal shift register with *n* bits.

7.15 Design a four-bit synchronous counter with parallel load. Use T flip-flops, instead of the D flip-flops used in section 7.9.3.

***7.16** Design a three-bit up/down counter using T flip-flops. It should include a control input called $\overline{\text{Up}}$/Down. If $\overline{\text{Up}}$/Down $= 0$, then the circuit should behave as an up-counter. If $\overline{\text{Up}}$/Down $= 1$, then the circuit should behave as a down-counter.

7.17 Repeat problem 7.16 using D flip-flops.

***7.18** The circuit in Figure P7.3 looks like a counter. What is the sequence that this circuit counts in?

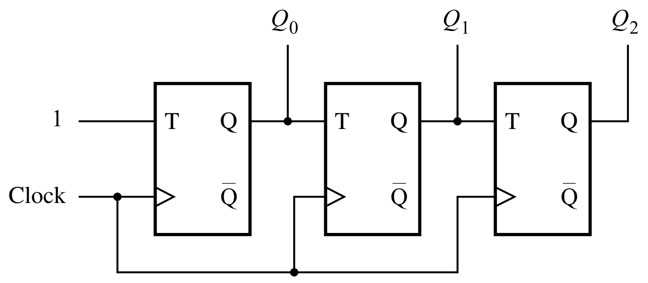


Figure P7.3 The circuit for Problem 7.18.

7.19 Consider the circuit in Figure P7.4. How does this circuit compare with the circuit in Figure 7.17? Can the circuits be used for the same purposes? If not, what is the key difference between them?

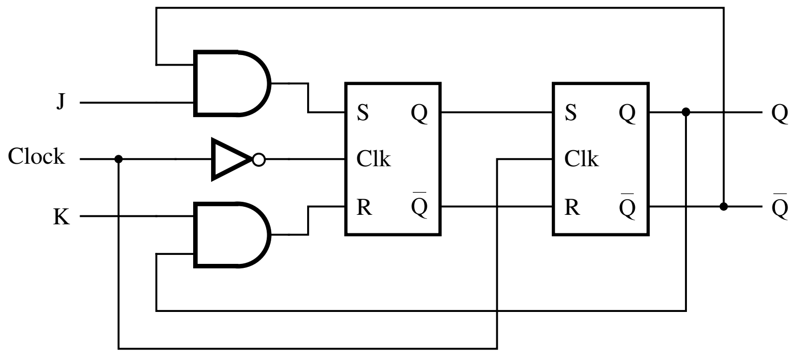


Figure P7.4 Circuit for Problem 7.19.

7.20 Construct a NOR-gate circuit, similar to the one in Figure 7.11*a*, which implements a negative-edge-triggered D flip-flop.

7.21 Write behavioral VHDL code that represents a 24-bit up/down-counter with parallel load and asynchronous reset.

7.22 Modify the VHDL code in Figure 7.52 by adding a parameter that sets the number of flip-flops in the counter.

7.23 Write behavioral VHDL code that represents a modulo-12 up-counter with synchronous reset.

***7.24** For the flip-flops in the counter in Figure 7.25, assume that $t_{su} = 3$ ns, $t_h = 1$ ns, and the propagation delay through a flip-flop is 1 ns. Assume that each AND gate, XOR gate, and 2-to-1 multiplexer has the propagation delay equal to 1 ns. What is the maximum clock frequency for which the circuit will operate correctly?

7.25 Write hierarchical code (structural) for the circuit in Figure 7.28. Use the counter in Figure 7.25 as a subcircuit.

7.26 Write VHDL code that represents an eight-bit Johnson counter. Synthesize the code with your CAD tools and give a timing simulation that shows the counting sequence.

7.27 Write behavioral VHDL code in the style shown in Figure 7.51 that represents a ring counter. Your code should have a parameter N that sets the number of flip-flops in the counter.

***7.28** Write behavioral VHDL code that describes the functionality of the circuit shown in Figure 7.42.

7.29 Figure 7.65 gives VHDL code for a digital system that swaps the contents of two registers, $R1$ and $R2$, using register $R3$ for temporary storage. Create an equivalent schematic using your CAD tools for this system. Synthesize a circuit for this schematic and perform a timing simulation.

7.30 Repeat problem 7.29 using the control circuit in Figure 7.59.

7.31 Modify the code in Figure 7.67 to use the control circuit in Figure 7.59. Synthesize the code for implementation in a chip and perform a timing simulation.

7.32 In section 7.14.2 we designed a processor that performs the operations listed in Table 7.3. Design a modified circuit that performs an additional operation Swap Rx, Ry. This operation swaps the contents of registers Rx and Ry. Use three bits $f_2 f_1 f_0$ to represent the input F shown in Figure 7.71 because there are now five operations, rather than four. Add a new register, named Tmp, into the system, to be used for temporary storage during the swap operation. Show logic expressions for the outputs of the control circuit, as was done in section 7.14.2.

7.33 A ring oscillator is a circuit that has an odd number, n, of inverters connected in a ringlike structure, as shown in Figure P7.5. The output of each inverter is a periodic signal with a certain period.
(a) Assume that all the inverters are identical; hence they all have the same delay, called t_p. Let the output of one of the inverters be named f. Give an equation that expresses the period of the signal f in terms of n and t_p.

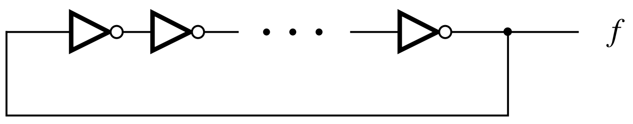


Figure P7.5 A ring oscillator.

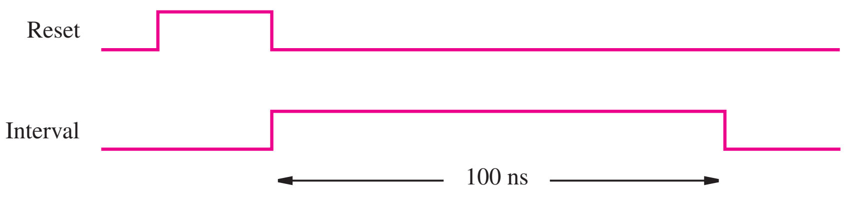
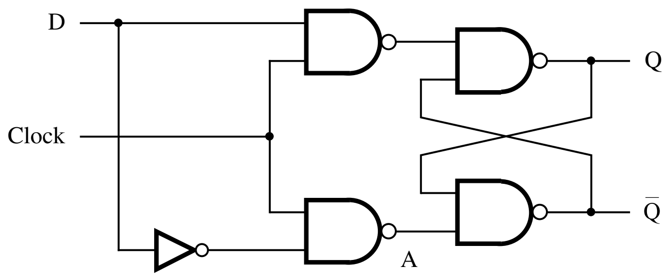


Figure P7.6 Timing of signals for Problem 7.33

(b) For this part you are to design a circuit that can be used to experimentally measure the delay t_p through one of the inverters in the ring oscillator. Assume the existence of an input called *Reset* and another called *Interval*. The timing of these two signals is shown in Figure P7.6. The length of time for which *Interval* has the value 1 is known. Assume that this length of time is 100 ns. Design a circuit that uses the *Reset* and *Interval* signals and the signal f from part (a) to experimentally measure t_p. In your design you may use logic gates and subcircuits such as adders, flip-flops, counters, registers, and so on.

7.34 A circuit for a gated D latch is shown in Figure P7.7. Assume that the propagation delay through either a NAND gate or an inverter is 1 ns. Complete the timing diagram given in the figure, which shows the signal values with 1 ns resolution.



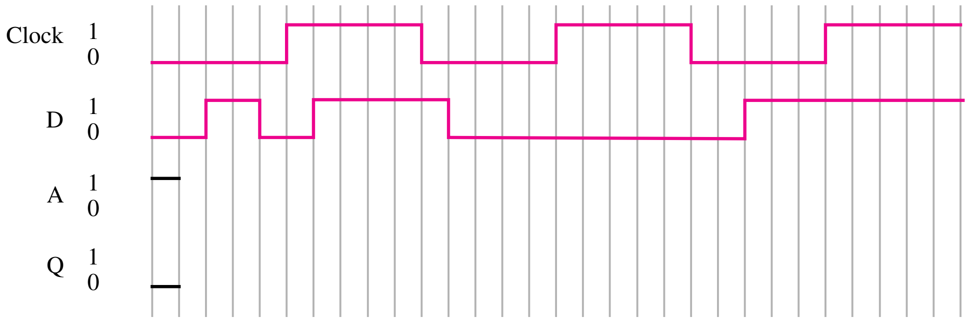


Figure P7.7 Circuit and timing diagram for Problem 7.34.

***7.35** A logic circuit has two inputs, *Clock* and *Start*, and two outputs, *f* and *g*. The behavior of the circuit is described by the timing diagram in Figure P7.8. When a pulse is received on the *Start* input, the circuit produces pulses on the *f* and *g* outputs as shown in the timing diagram. Design a suitable circuit using only the following components: a three-bit resettable positive-edge-triggered synchronous counter and basic logic gates. For your answer assume that the delays through all logic gates and the counter are negligible.

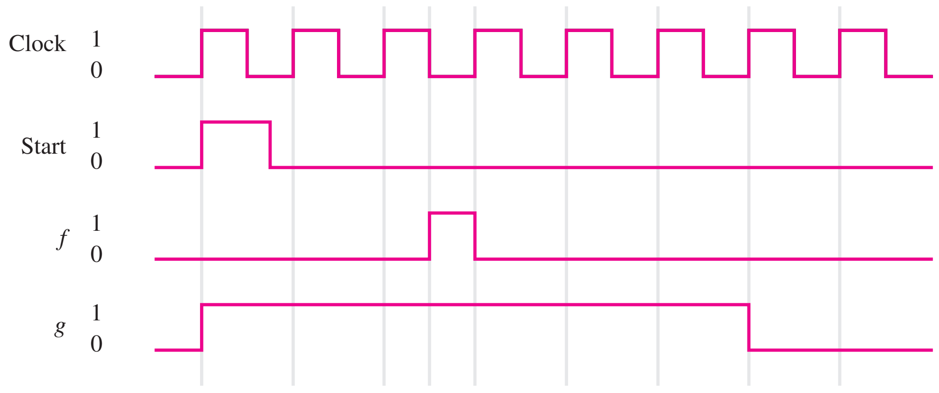


Figure P7.8 Timing diagram for Problem 7.35.

7.36 Write behavioral VHDL code for a four-digit BCD counter.

7.37 Determine the maximum clock frequency that can be used for the circuit in Figure 7.25. Use the timing parameters given in section 7.15.

7.38 Repeat problem 7.37 for the circuit in Figure 7.60.

7.39 (a) Draw a circuit that could be synthesized from the VHDL code in Figure P7.9.
(b) How would you modify this code to specify a crossbar switch?

7.40 A digital control circuit has three inputs: *Start, Stop* and *Clock*, as well as an output signal *Run*. The *Start* and *Stop* signals are of indeterminate duration and may span many clock cycles. When the *Start* signal goes to 1, the circuit must generate *Run* = 1. The *Run* signal must remain high until the *Stop* signal goes to 1, at which time it has to return to 0. All changes in the *Run* signal must be synchronized with the *Clock* signal.
(a) Design the desired control circuit.
(b) Write VHDL code that specifies the desired circuit.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( x1, x2, s: IN     STD_LOGIC ;
                y1, y2    : OUT  STD_LOGIC) ;
END problem ;

ARCHITECTURE Behavior OF problem IS
BEGIN
    PROCESS ( x1, x2, s )
    BEGIN
        IF s = '0' THEN
            y1 <= x1 ;
            y2 <= x2 ;
        ELSIF s = '1' THEN
            y1 <= x2 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure P7.9 Code for Problem 7.39.

References

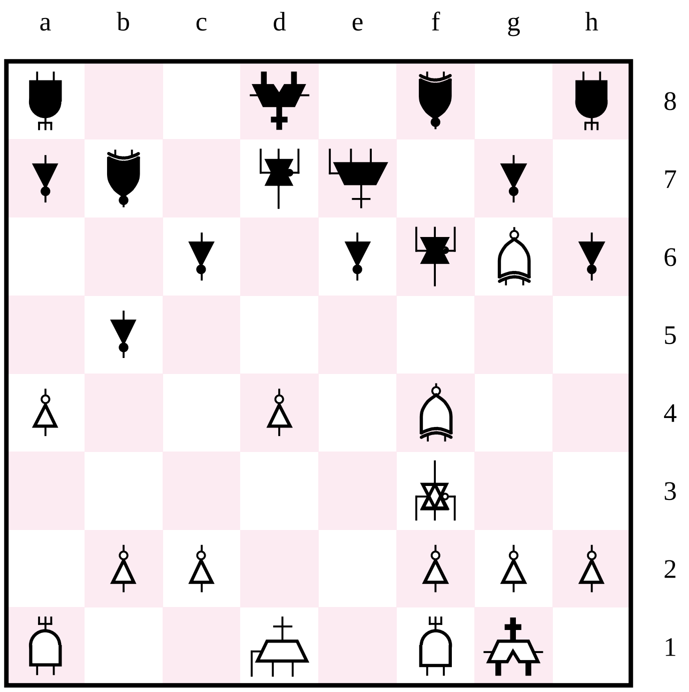
1. V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed., (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 3rd ed., (Morgan Kaufmann: San Francisco, Ca., 2004).
3. R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., (Pearson Prentice-Hall: Upper Saddle River, N.J., 2005).
4. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed. (Prentice-Hall: Englewood Cliffs, N.J., 2005).
5. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
6. M. M. Mano, *Digital Design*, 3rd ed. (Prentice-Hall: Upper Saddle River, N.J., 2002).
7. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, N.J., 1997).
8. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).

9. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, N.J., 1995).
10. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, Ma., 1993).
11. E. J. McCluskey, *Logic Design Principles*, (Prentice-Hall: Englewood Cliffs, N.J., 1986).
12. Institute of Electrical and Electronics Engineers, "1076-1993 IEEE Standard VHDL Language Reference Manual," 1993.
13. D. L. Perry, *VHDL*, 3rd ed., (McGraw-Hill: New York, 1998).
14. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2nd ed. (McGraw-Hill: New York, 1998).
15. J. Bhasker, *A VHDL Primer*, 3rd ed. (Prentice-Hall: Englewood Cliffs, N.J., 1998).
16. K. Skahill, *VHDL for Programmable Logic*, (Addison-Wesley: Menlo Park, Ca., 1996).
17. A. Dewey, *Analysis and Design of Digital Systems with VHDL*, (PWS Publishing Co.: Boston, Ma., 1997).

appendix

A

VHDL Reference



12. a2–a4, Bc8–b7

This appendix describes the features of VHDL that are used in this book. It is meant to serve as a convenient reference for the reader. Hence only brief descriptions are provided, along with examples. The reader is encouraged to first study the introduction to VHDL in sections 2.9 and 4.12.5.

Another useful source of information on VHDL is the MAX+plusII CAD system that accompanies the book. The on-line help included with the software describes how to use VHDL with MAX+plusII, and the "templates" provided with the Text Editor tool are a convenient guide to VHDL syntax. We describe how to access these features of the CAD tools in Appendix B.

In some ways VHDL uses an unusual syntax for describing logic circuits. The prime reason is that VHDL was originally intended to be a language for documenting and simulating circuits, rather than for describing circuits for synthesis. This appendix is not meant to be a comprehensive VHDL manual. While we discuss almost all the features of VHDL that are useful in the synthesis of logic circuits, we do not discuss any of the features that are useful only for simulation of circuits or for other purposes. Although the omitted features are not needed for any of the examples used in this book, a reader who wishes to learn more about using VHDL can refer to specialized books [1–7].

How *Not* to Write VHDL Code

In section 2.9 we mentioned the most common problem encountered by designers who are just beginning to write VHDL code. The tendency for the novice is to write code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesizing such code. This book contains more than 100 examples of complete VHDL code that represents a wide range of logic circuits. In all of these examples, the code is easily related to the described logic circuit. The reader is encouraged to adopt the same style of code. A good general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the VHDL code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to describe.

Since VHDL is a complex language, errors in syntax and usage are quite common. Some problems encountered by our students, as novice designers, are listed at the end of this appendix in section A.11. The reader may find it useful to examine these errors in an effort to avoid them when writing code.

Once complete VHDL code is written for a particular design, it is useful to analyze the resulting circuit synthesized by the CAD tools. Much can be learned about VHDL, logic circuits, and logic synthesis by studying the circuits that are produced automatically by the CAD tools.

A.1 DOCUMENTATION IN VHDL CODE

Documentation can be included in VHDL code by writing a comment. The two characters '-', '-' denote the beginning of the comment. The VHDL compiler ignores any text on a line after the '--'.

Example A.1

```
-- this is a VHDL comment
```

A.2 Data Objects

Information is represented in VHDL code as data objects. Three kinds of data objects are provided: signals, constants, and variables. For describing logic circuits, the most important data objects are signals. They represent the logic signals (wires) in the circuit. The constants and variables are also sometimes useful for describing logic circuits, but they are used infrequently.

A.2.1 Data Object Names

The rules for specifying data object names are simple: any alphanumeric character may be used in the name, as well as the '_' underscore character. There are four caveats. A name cannot be a VHDL keyword, it must begin with a letter, it cannot end with an '_' underscore, and it cannot have two successive '_' underscores. Thus examples of legal names are *x*, *x*1, *x_y*, and *Byte*. Some examples of illegal names are 1*x*, *_y*, *x__y*, and *entity*. The latter name is not allowed because it is a VHDL keyword. We should note that VHDL is not case sensitive. Hence x is the same as X, and ENTITY is the same as entity. To make the examples of VHDL code in this book more readable, we use uppercase letters in all keywords.

To avoid confusion when using the word signal, which can mean either a VHDL data object or a logic signal in a circuit, we sometimes write the VHDL data object as SIGNAL.

A.2.2 Data Object Values and Numbers

We use SIGNAL data objects to represent individual logic signals in a circuit, multiple logic signals, and binary numbers (integers). The value of an individual SIGNAL is specified using apostrophes, as in '0' or '1'. The value of a multibit SIGNAL is given with double quotes. An example of a four-bit SIGNAL value is "1001", and an eight-bit value is "10011000". Double quotes can also be used to denote a binary number. Hence while "1001" can represent the four SIGNAL values '1', '0', '0', '1', it can also mean the integer $(1001)_2 = (9)_{10}$. Integers can alternatively be specified in decimal by not using quotes, as in 9 or 152. The values of CONSTANT or VARIABLE data objects are specified in the same way as for SIGNAL data objects.

A.2.3 SIGNAL Data Objects

SIGNAL data objects represent the logic signals, or wires, in a circuit. There are three places in which signals can be declared in VHDL code: in an entity declaration (see section

A.4.1), in the declarative section of an architecture (see section A.4.2), and in the declarative section of a package (see section A.5). A signal has to be declared with an associated *type*, as follows:

SIGNAL signal_name : type_name ;

The signal's *type_name* determines the legal values that the signal can have and its legal uses in VHDL code. In this section we describe 10 signal types: BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, SIGNED, UNSIGNED, INTEGER, ENUMERATION, and BOOLEAN.

A.2.4 BIT and BIT_VECTOR Types

These types are predefined in the VHDL Standards IEEE 1076 and IEEE 1164. Hence no library is needed to use these types in the code. Objects of BIT type can have the values '0' or '1'. An object of BIT_VECTOR type is a linear array of BIT objects.

Example A.2

```
SIGNAL  x1    : BIT ;
SIGNAL  C     : BIT_VECTOR (1 TO 4) ;
SIGNAL  Byte : BIT_VECTOR (7 DOWNTO 0) ;
```

The signals *C* and *Byte* illustrate the two possible ways of defining a multibit data object. The syntax "lowest_index TO highest_index" is useful for a multibit signal that is simply an array of bits. In the signal *C* the most-significant (left-most) bit is referenced using lowest_index, and the least-significant (right-most) bit is referenced using highest_index. The syntax "highest_index DOWNTO lowest_index" is useful if the signal represents a binary number. In this case the most-significant (left-most) bit has the index highest_index, and the least-significant (right-most) bit has the index lowest_index.

The multibit signal *C* represents four BIT objects. It can be used as a single four-bit quantity, or each bit can be referred to individually. The syntax for referring to the signals individually is $C(1)$, $C(2)$, $C(3)$, or $C(4)$. An assignment statement such as

```
C  <= "1010" ;
```

results in $C(1) = 1$, $C(2) = 0$, $C(3) = 1$, and $C(4) = 0$.

The signal *Byte* comprises eight BIT objects. The assignment statement

```
Byte <= "10011000" ;
```

results in $Byte(7) = 1$, $Byte(6) = 0$, and so on to $Byte(0) = 0$.

A.2.5 STD_LOGIC and STD_LOGIC_VECTOR Types

The STD_LOGIC type was added to the VHDL Standard in IEEE 1164. It provides more flexibility than the BIT type. To use this type, we must include the two statements

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
```

These statements provide access to the *std_logic_1164* package, which defines the STD_LOGIC type. We describe VHDL packages in section A.5. In general, they are used as a place to store VHDL code, such as the code that defines a type, which can then be used in other source code files. The following values are legal for a STD_LOGIC data object: 0, 1, Z, −, L, H, U, X, and W. Only the first four are useful for synthesis of logic circuits. The value Z represents high impedance, and − stands for "don't care." The value L stands for "weak 0," H means "weak 1," U means "uninitialized," X means "unknown," and W means "weak unknown." The STD_LOGIC_VECTOR type represents an array of STD_LOGIC objects.

Example A.3

```
SIGNAL x1, x2, Cin, Cout, Sel  : STD_LOGIC ;
SIGNAL C                       : STD_LOGIC_VECTOR (1 TO 4) ;
SIGNAL X, Y, S                 : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
```

STD_LOGIC objects are often used in logic expressions in VHDL code. STD_LOGIC_VECTOR signals can be used as binary numbers in arithmetic circuits by including in the code the statement

```
USE ieee.std_logic_signed.all ;
```

The *std_logic_signed* package specifies that it is legal to use the STD_LOGIC_VECTOR signals with arithmetic operators, like + (see section A.7.1). The VHDL compiler should generate a circuit that works for signed numbers. An alternative is to use the package *std_logic_unsigned*. In this case the compiler should generate a circuit that works for unsigned numbers.

A.2.6 STD_ULOGIC TYPE

In this book we use the STD_LOGIC type in most examples of VHDL code. This type is actually a *subtype* of the STD_ULOGIC type. Signals that have the STD_ULOGIC type can take the same values as the STD_LOGIC signals that we have been using. The only difference between STD_ULOGIC and STD_LOGIC has to do with the concept of a *resolution function*. In VHDL a resolution function is used to determine what value a signal should take if there are two sources for that signal. For example, two tri-state buffers could both have their outputs connected to a signal, *x*. At some given time one buffer might produce the output value 'Z' and the other buffer might produce the value 1. A resolution function is used to determine that the value of *x* should be 1 in this case. The STD_LOGIC type allows multiple sources for a signal; it resolves the correct value using a resolution function that is provided as part of the *std_logic_1164* package. The STD_ULOGIC type does not permit signals to have multiple sources. We have introduced STD_ULOGIC for completeness only; it is not used in this book.

A.2.7 SIGNED AND UNSIGNED TYPES

The *std_logic_signed* and *std_logic_unsigned* packages mentioned in section A.2.5 make use of another package, called *std_logic_arith*. This package defines the type of circuit that should be used to implement the arithmetic operators, such as +. The *std_logic_arith* package defines two signal types, SIGNED and UNSIGNED. These types are identical to the STD_LOGIC_VECTOR type because they represent an array of STD_LOGIC signals. The purpose of the SIGNED and UNSIGNED types is to allow the user to indicate in the VHDL code what kind of number representation is being used. The SIGNED type is used in code for circuits that deal with signed (2's complement) numbers, and the UNSIGNED type is used in code that deals with unsigned numbers.

Example A.4 Assume that A and B are signals with the SIGNED type. Assume that A is assigned the value "1000", and B is assigned the value "0001". VHDL provides relational operators (see Table A.1 in section A.3) that can be used to compare the values of two signals. The comparison $A < B$ evaluates to true because the signed values are $A = -8$ and $B = 1$. On the other hand, if A and B are defined with the UNSIGNED type, then $A < B$ evaluates to false because the unsigned values are $A = 8$ and $B = 1$.

The *std_logic_signed* package specifies that STD_LOGIC_VECTOR signals should be treated like SIGNED signals. Similarly, *std_logic_unsigned* specifies that STD_LOGIC_VECTOR signals should be treated like UNSIGNED signals. It is an arbitrary choice whether code is written using STD_LOGIC_VECTOR signals in conjunction with the *std_logic_signed* or *std_logic_unsigned* packages or using SIGNED and UNSIGNED signals with the *std_logic_arith* package.

The *std_logic_arith* package, and hence the *std_logic_signed* and *std_logic_unsigned* packages, are not actually a part of the VHDL standards. They are provided by Synopsys Inc., which is a vendor of CAD software. However, these packages are included with most CAD systems that support VHDL, and they are widely used in practice.

A.2.8 INTEGER TYPE

The VHDL standard defines the INTEGER type for use with arithmetic operators. In this book the STD_LOGIC_VECTOR type is usually preferred in code for arithmetic circuits, but the INTEGER type is used occasionally. An INTEGER signal represents a binary number. The code does not specifically give the number of bits in the signal, as it does for STD_LOGIC_VECTOR signals. By default, an INTEGER signal has 32 bits and can represent numbers from $-(2^{31} - 1)$ to $2^{31} - 1$. This is one number less than the normal 2's complement range; the reason is simply that the VHDL standard specifies an equal number of negative and positive numbers. Integers with fewer bits can also be declared, using the RANGE keyword.

Example A.5

```
SIGNAL X : INTEGER RANGE −127 TO 127 ;
```

This defines *X* as an eight-bit signed number.

A.2.9 BOOLEAN TYPE

An object of type BOOLEAN can have the values TRUE or FALSE, where TRUE is equivalent to 1 and FALSE is 0.

Example A.6

```
SIGNAL Flag : Boolean ;
```

A.2.10 ENUMERATION TYPE

A SIGNAL of ENUMERATION type is one for which the possible values that the signal can have are user specified. The general form of an ENUMERATION type is

```
TYPE enumerated_type_name IS (name {, name}) ;
```

The curly brackets indicate that one or more additional items can be included. We use these brackets in this manner in several places in the appendix. The most common example of using the ENUMERATION type is for specifying the states for a finite-state machine.

Example A.7

```
TYPE State_type IS (stateA, stateB, stateC) ;
SIGNAL y : State_type ;
```

This declares a signal named *y*, for which the legal values are *stateA*, *stateB*, and *stateC*. When the code is translated by the VHDL compiler, it automatically assigns bit patterns (codes) to represent *stateA*, *stateB*, and *stateC*.

A.2.11 CONSTANT Data Objects

A CONSTANT is a data object whose value cannot be changed. Unlike a SIGNAL, a CONSTANT does not represent a wire in a circuit. The general form of a CONSTANT declaration is

```
CONSTANT constant_name : type_name := constant_value ;
```

The purpose of a constant is to improve the readability of code, by using the name of the constant in place of a value or number.

Example A.8

```
CONSTANT Zero : STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000" ;
```

Then the word *Zero* can be used in the code to indicate the constant value "0000".

A.2.12 VARIABLE Data Objects

A VARIABLE, unlike a SIGNAL, does not necessarily represent a wire in a circuit. VARIABLE data objects are sometimes used to hold the results of computations and for the index variables in loops. We will give some examples in section A.9.7.

A.2.13 Type Conversion

VHDL is a strongly type-checked language, which means that it does not permit the value of a signal of one type to be assigned to another signal that has a different type. Even for signals that intuitively seem compatible, such as BIT and STD_LOGIC, using the two types together is not permitted. To avoid this problem, we generally use only the STD_LOGIC and STD_LOGIC_VECTOR types in this book. When it is necessary to use code that has a mixture of types, type-conversion functions can be used to convert from one type to another.

Assume that X is defined as an eight-bit STD_LOGIC_VECTOR signal and Y is an INTEGER signal defined with the range from 0 to 255. An example of a conversion function that allows the value of Y to be assigned to X is

```
X <= CONV_STD_LOGIC_VECTOR(Y, 8) ;
```

This conversion function has two parameters: the name of the signal to be converted and the number of bits in X. The function is provided as part of the *std_logic_arith* package; hence that package must be included in the code using the appropriate LIBRARY and USE clauses. Other conversion functions are described in the MAX+plusII on-line help.

A.2.14 ARRAYS

We said above that the BIT_VECTOR and STD_LOGIC_VECTOR types are arrays of BIT and STD_LOGIC signals, respectively. The definitions of these arrays, which are provided as part of the VHDL standards, are

```
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE < >) OF BIT ;
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE < >) OF STD_LOGIC ;
```

The sizes of the arrays are not set in the definitions; the syntax (NATURAL RANGE < >) has the effect of allowing the user to set the size of the array when a data object of either type is declared. Arrays of any type can be defined by the user. For example

```
TYPE Byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC ;
SIGNAL X : Byte ;
```

declares the signal *X* with the type *Byte*, which is an eight-element array of STD_LOGIC data objects.

An example that defines a two-dimensional array is

```
TYPE RegArray IS ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL R : RegArray ;
```

This code defines *R* as an array with four elements. Each element is an eight-bit STD_LOGIC_VECTOR signal. The syntax $R(i)$, where $3 \geq i \geq 0$, is used to refer to element *i* of the array. The syntax $R(i)(j)$, where $7 \geq j \geq 0$, is used to refer to one bit in the array $R(i)$. This bit has the type STD_LOGIC. An example using the *RegArray* type is given in section 10.2.6.

A.3 OPERATORS

VHDL provides Boolean operators, arithmetic operators, and relational operators. They are categorized in an unusual way, shown in Table A.1, according to the precedence of the operators. Note that operators in the same category do not have precedence over one another. There is no precedence among any Boolean operators. Thus a logic expression

```
x1 AND x2 AND x3 OR x4
```

does not have the $x_1x_2x_3 + x_4$ meaning that would be expected because AND does not have precedence over OR. In fact, this expression is not even legal in VHDL as written above. To be both legal and have the desired meaning, it must be written as

```
(x1 AND x2 AND x3) OR x4
```

For the relational operators, /= means *not equal*, <= means *less than or equal*, and >= means *greater than or equal*.

Table A.1 The VHDL operators.

	Operator Class	Operator
Highest precedence	Miscellaneous	**, ABS, NOT
	Multiplying	*, /, MOD, REM
	Sign	+, −
	Adding	+, −, &
	Relational	=, /=, <, <=, >, >=
Lowest precedence	Logical	AND, OR, NAND, NOR, XOR, XNOR

A.4 VHDL Design Entity

A circuit or subcircuit described with VHDL code is called a *design entity*, or just *entity*. Figure A.1 shows the general structure of an entity. It has two main parts: the *entity declaration*, which specifies the input and output signals for the entity, and the *architecture*, which gives the circuit details.

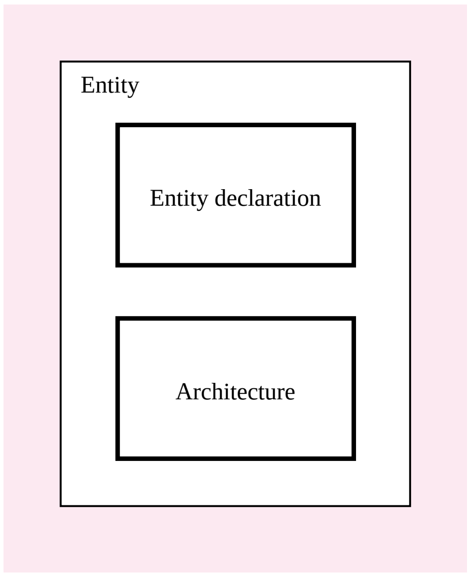


Figure A.1 The general structure of a VHDL design entity.

A.4.1 ENTITY Declaration

The input and output signals in an entity are specified using the ENTITY declaration, as indicated in Figure A.2. The name of the entity can be any legal VHDL name. The square brackets indicate an optional item. The input and output signals are specified using the keyword PORT. Whether each port is an input, output, or bidirectional signal is specified by the *mode* of the port. The available modes are summarized in Table A.2. If the mode of a port is not specified, it is assumed to have the mode IN.

Table A.2 The possible modes for signals that are entity ports.

Mode	**Purpose**
IN	Used for a signal that is an input to an entity.
OUT	Used for a signal that is an output from an entity. The value of the signal can not be used inside the entity. This means that in an assignment statement, the signal can appear only to the left of the <= operator.
INOUT	Used for a signal that is both an input to an entity and an output from the entity.
BUFFER	Used for a signal that is an output from an entity. The value of the signal can be used inside the entity, which means that in an assignment statement, the signal can appear both on the left and right sides of the <= operator.

A.4.2 Architecture

An ARCHITECTURE provides the circuit details for an entity. The general structure of an architecture is shown in Figure A.3. It has two main parts: the *declarative region* and the *architecture body*. The declarative region appears preceding the BEGIN keyword. It can be used to declare signals, user-defined types, and constants. It can also be used to declare components and to specify attributes; we discuss the COMPONENT and ATTRIBUTE keywords in sections A.6 and A.10.13, respectively.

The functionality of the entity is specified in the architecture body, which follows the BEGIN keyword. This specification involves statements that define the logic functions in the circuit, which can be given in a variety of ways. We will discuss a number of possibilities in the sections that follow.

```
ENTITY entity_name IS
    PORT ( [SIGNAL] signal_name {, signal_name} : [mode] type_name {;
                SIGNAL] signal_name {, signal_name}  : [mode] type_name } ) ;
END entity_name ;
```

Figure A.2 The general form of an entity declaration.

```
ARCHITECTURE architecture_name OF entity_name IS
      [SIGNAL declarations]
      [CONSTANT declarations]
      [TYPE declarations]
      [COMPONENT declarations]
      [ATTRIBUTE specifications]
BEGIN
      {COMPONENT instantiation statement ;}
      {CONCURRENT ASSIGNMENT statement ;}
      {PROCESS statement ;}
      {GENERATE statement ;}
END [architecture_name] ;
```

Figure A.3 The general form of an architecture.

Example A.9 Figure A.4 gives the VHDL code for an entity named *fulladd*, which represents a full-adder circuit. (The full-adder is discussed in section 5.2.) The entity declaration specifies the input and output signals. The input port *Cin* is the carry-in, and the bits to be added are the input ports *x* and *y*. The output ports are the sum, *s*, and the carry-out, *Cout*. The input and output signals are called the *ports* of the entity. This term is adopted from the electrical jargon in which it refers to an input or output connection in an electrical circuit.

The architecture defines the functionality of the full-adder using logic equations. The name of the architecture can be any legal VHDL name. We chose the name LogicFunc for this simple example. In terms of the general form of the architecture in Figure A.3, a logic equation is a type of concurrent assignment statement. These statements are described in section A.7.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
      PORT ( Cin, x, y   : IN     STD_LOGIC ;
                  s, Cout     : OUT  STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
      s <= x XOR y XOR Cin ;
      Cout <= (x AND y) OR (x AND Cin) OR (y AND Cin) ;
END LogicFunc ;
```

Figure A.4 Code for a full-adder.

A.5 Package

A VHDL package serves as a repository. It is used to hold VHDL code that is of general use, like the code that defines a type. The package can be included for use in any number of other source code files, which can then use the definitions provided in the package. Like an architecture, introduced in section A.4.2, a package can have two main parts: the *package declaration* and the *package body*. The *package_body* is an optional part, which we do not use in this book; one use of a package body is to define VHDL functions, such as the conversion functions introduced in section A.2.13.

The general form of a package declaration is depicted in Figure A.5. Definitions provided in the package, such as the definition of a type, can be used in any source code file that includes the statements

```
LIBRARY library_name ;
USE library_name.package_name.all ;
```

The *library_name* represents the location in the computer file system where the package is stored. A library can either be provided as part of a CAD system, in which case it is termed a *system library*, or be created by the user, in which case it is called a *user library*. An example of a system library is the *ieee* library. We discussed four packages in that library in section A.2: *std_logic_1164*, *std_logic_signed*, *std_logic_unsigned*, and *std_logic_arith*.

A special case of a user library is represented by the file-system directory where the VHDL source code file that declares a package is stored. This directory can be referred to by the library name *work*, which stands for *working directory*. Hence, if a source code file that contains a package declaration called *user_package_name* is compiled, then the package can be used in another source code file (which is stored in the same file-system directory) by including the statements

```
LIBRARY work ;
USE work.user_package_name.all ;
```

Actually, for the special case of the *work* library, the LIBRARY clause is not required, because the work library is always accessible.

Figure A.5 shows that the package declaration can be used to declare signals and components. Components are discussed in the next section. A signal declared in a package can be used by any design entity that accesses the package. Such signals are similar in

```
PACKAGE package_name IS
    [TYPE declarations]
    [SIGNAL declarations]
    [COMPONENT declarations]
END package_name ;
```

Figure A.5 The general form of a PACKAGE declaration.

concept to global variables used in computer programming languages. In contrast, a signal declared in an architecture can be used only inside that architecture. Such signals are analogous to local variables in a programming language.

A.6 Using Subcircuits

A VHDL entity defined in one source code file can be used as a subcircuit in another source code file. In VHDL jargon the subcircuit is called a *component*. A subcircuit must be declared using a *component declaration*. This statement specifies the name of the subcircuit and gives the names of its input and output ports. The component declaration can appear either in the declaration region of an architecture or in a package declaration. The general form of the statement is shown in Figure A.6. The syntax used is similar to the syntax in an entity declaration.

Once a component declaration is given, the component can be *instantiated* as a subcircuit. This is done using a *component instantiation* statement. It has the general form

```
instance_name : component_name PORT MAP (
    formal_name  => actual_name  {, formal_name  =>  actual_name} ) ;
```

Each *formal_name* is the name of a port in the subcircuit. Each *actual_name* is the name of a signal in the code that instantiates the subcircuit. The syntax "formal_name =>" is provided so that the order of the signals listed after the PORT MAP keywords does not have to be the same as the order of the ports in the corresponding COMPONENT declaration. In VHDL jargon this is called the *named association*. If the signal names following the PORT MAP keywords are given in the same order as in the COMPONENT declaration, then "formal_name =>" is not needed. This is called the *positional association*.

An example using a component (subcircuit) is shown in Figure A.7. It gives the code for a four-bit ripple-carry adder built using four instances of the *fulladd* subcircuit. The inputs to the adder are the carry-in, *Cin*, and the 2 four-bit numbers X and Y. The output is the four-bit sum, S, and the carry-out, *Cout*. We have chosen the name Structure in the architecture because the hierarchical style of code that uses subcircuits is often called the *structural* style. Observe that a three-bit signal, C, is declared to represent the carry-outs from stages 0, 1, and 2. This signal is declared in the architecture, rather than in the entity declaration, because it is used internally in the circuit and is not an input or output port.

```
COMPONENT component_name
     [GENERIC (parameter_name : integer := default_value {;
                    parameter_name : integer := default_value} ) ;]
     PORT ( [SIGNAL] signal_name {, signal_name} : [mode] type_name {;
             SIGNAL] signal_name {, signal_name}  : [mode] type_name } ) ;
END COMPONENT ;
```

Figure A.6 The general form of a component declaration.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder IS
    PORT ( Cin    : IN     STD_LOGIC ;
           X, Y   : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S      : OUT    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout   : OUT    STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN     STD_LOGIC ;
               s, Cout   : OUT STD_LOGIC) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin , X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP (
        x => X(3), y => Y(3), Cin => C(3), s => S(3), Cout => Cout ) ;
END Structure ;
```

Figure A.7 Code for a four-bit adder, using component instantiation.

The next statement in the architecture gives the component declaration for the *fulladd* subcircuit. The architecture body instantiates four copies of the full-adder subcircuit. In the first three instantiation statements, we have used positional association because the signals are listed in the same order as given in the declaration for the fulladd component in Figure A.4. The last instantiation statement gives an example of named association. Note that it is legal to use the same name for a signal in the architecture that is used for a port name in a component. An example of this is the *Cout* signal. The signal names used in the instantiation statements implicitly specify how the component instances are interconnected to create the adder entity.

A second example of component instantiation is shown in Figure A.8. A package called *lpm_components* in the library named *lpm* is included in the code. This package represents a collection of components called the *Library of Parameterized Modules (LPM)*, which is a standardized library of circuit building blocks that are generally useful for implementing logic circuits. The MAX+plusII CAD system includes the LPM components as standard building blocks for creating logic circuits. Information about the components in the library can be found in the MAX+plusII on-line help. We describe how to access this information in Tutorial 3.

The code in Figure A.8 instantiates the LPM component called *lpm_add_sub*, which is introduced in section 5.5.1. It represents an adder/subtractor circuit. The GENERIC

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY lpm ;
USE lpm.lpm_components.all ;

ENTITY adderLPM IS
    PORT ( Cin     : IN    STD_LOGIC ;
           X, Y    : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S       : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout    : OUT   STD_LOGIC ) ;
END adderLPM ;

ARCHITECTURE Structure OF adderLPM IS
BEGIN
    instance: lpm_add_sub
        GENERIC MAP (LPM_WIDTH => 4)
        PORT MAP (
            dataa => X, datab => Y, Cin => Cin, result => S, Cout => Cout ) ;
END Structure ;
```

Figure A.8 Instantiating a four-bit adder from the LPM library.

keyword is used to set the number of bits in the adder/subtractor to 4. We discuss generics in section A.8. The function of each PORT on the *lpm_add_sub* component is self-evident from the port names used in the instantiation statement.

A.6.1 Declaring a COMPONENT in a Package

Figure A.5 shows that a component declaration can be given in a package. An example is shown in Figure A.9. It defines the package named *fulladd_package*, which provides the component declaration for the *fulladd* entity. This package can be stored in a separate

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT ( Cin, x, y   : IN    STD_LOGIC ;
               s, Cout     : OUT   STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Figure A.9 An example of a package declaration.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder IS
    PORT ( Cin    : IN    STD_LOGIC ;
           X, Y   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S      : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout   : OUT   STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

Figure A.10 Using a component defined in a package.

source code file or can be included at the end of the file that defines the *fulladd* entity (see Figure A.4). Any source code that includes the statement "USE work.fulladd_package.all" can use the *fulladd* component as a subcircuit. Figure A.10 shows how a four-bit ripple-carry adder entity can be written to use the package. The code is the same as that in Figure A.7 except that it includes the extra USE clause for the package and deletes the component declaration statement from the architecture.

A.7 Concurrent Assignment Statements

A concurrent assignment statement is used to assign a value to a signal in an architecture body. An example was given in Figure A.4, in which the logic equations illustrate one type of concurrent assignment statement. VHDL provides four different types of concurrent assignment statements: simple signal assignment, selected signal assignment, conditional signal assignment, and generate statements.

A.7.1 Simple Signal Assignment

A simple signal assignment statement is used for a logic or an arithmetic expression. The general form is

```
signal_name <= expression ;
```

where <= is the VHDL *assignment operator*. The following examples illustrate its use.

```
SIGNAL x1, x2, x3, f  : STD_LOGIC ;
       .
       .
       .
f <= (x1 AND x2) OR x3 ;
```

This defines *f* in a logic expression, which involves single-bit quantities. VHDL also supports multibit logic expressions, as in

```
SIGNAL A, B, C  : STD_LOGIC_VECTOR (1 TO 3) ;
       .
       .
       .
C <= A AND B ;
```

This results in $C(1)=A(1)\cdot B(1)$, $C(2)=A(2)\cdot B(2)$, and $C(3)=A(3)\cdot B(3)$.

An example of an arithmetic expression is

```
SIGNAL X, Y, S  : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
       .
       .
       .
S <= X + Y ;
```

This represents a four-bit adder, without carry-in and carry-out. We can alternatively declare a carry-in signal, *Cin*, and a five-bit signal, *Sum*, as follows

```
SIGNAL Cin  : STD_LOGIC ;
SIGNAL Sum  : STD_LOGIC_VECTOR (4 DOWNTO 0) ;
```

Then the statement

```
Sum <= ('0' & X) + Y + Cin ;
```

represents the four-bit adder with carry-in and carry-out. The four sum bits are *Sum*(3) to *Sum*(0), while the carry-out is the bit *Sum*(4). The syntax ('0' & X) uses the VHDL *concatenate operator*, &, to put a 0 on the left end of the signal *X*. The reader should not confuse this use of the & symbol with the logical AND operation, which is the usual meaning of this symbol; in VHDL the logical AND is indicated by the word AND, and & means concatenate. The concatenate operation prepends a 0 digit onto *X*, creating a five-bit number. VHDL requires at least one of the operands of an arithmetic expression to have the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Behavior OF adder IS
    SIGNAL Sum : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(3 DOWNTO 0) ;
    Cout <= Sum(4) ;
END Behavior ;
```

Figure A.11 Code for a four-bit adder, using arithmetic expressions.

same number of bits as the signal used to hold the result. The complete code for the four-bit adder with carry signals is given in Figure A.11. We should note that this is a different way (it is actually a better way) to describe a four-bit adder, in comparison with the structural code in Figure A.7. Observe that the statement “S <= Sum(3 DOWNTO 0)” assigns the lower four bits of the *Sum* signal, which are the four sum bits, to the output *S*.

A.7.2 Assigning Signal Values Using OTHERS

Assume that we wish to set all bits in the signal *S* to 0. As we already know, one way to do so is to write “S <= "0000" ;”. If the number of bits in *S* is large, a more convenient way of expressing the assignment statement is to use the OTHERS keyword, as in

```
S <= (OTHERS => '0') ;
```

This statement also sets all bits in *S* to 0. But it has the benefit of working for any number of bits, not just four. In general, the meaning of (OTHERS => *Value*) is to set each bit of the destination operand to *Value*. An example of code that uses this construct is shown in Figure A.28.

A.7.3 SELECTED SIGNAL ASSIGNMENT

A selected signal assignment statement is used to set the value of a signal to one of several alternatives based on a selection criterion. The general form is

```
[label:] - - an optional label can be placed here
WITH expression SELECT
    signal_name <= expression WHEN constant_value{,
                   expression WHEN constant_value} ;
```

Example A.10

```
SIGNAL x1, x2, Sel, f : STD_LOGIC ;
      .
      .
      .
WITH Sel SELECT
   f <=  x1 WHEN '0',
         x2 WHEN OTHERS ;
```

This code describes a 2-to-1 multiplexer with *Sel* as the select input. In a selected signal assignment, all possible values of the select input, *Sel* in this case, must be explicitly listed in the code. The word OTHERS provides an easy way to meet this requirement. OTHERS represents all possible values not already listed. In this case the other possible values are 1, Z, −, and so on. Another requirement for the selected signal assignment is that each WHEN clause must specify a criterion that is mutually exclusive of the criteria in all other WHEN clauses.

A.7.4 CONDITIONAL SIGNAL ASSIGNMENT

Similar to the selected signal assignment, the conditional signal assignment is used to set a signal to one of several alternative values. The general form is

```
[label:]
signal_name <=  expression WHEN logic_expression ELSE
                {expression WHEN logic_expression ELSE}
                expression ;
```

An example is

```
f <= '1' WHEN x1 = x2 ELSE '0' ;
```

One key difference in comparison with the selected signal assignment has to be noted. The conditions listed after each WHEN clause need not be mutually exclusive, because the conditions are given a priority from the first listed to the last listed. This is illustrated by the example in Figure A.12. The code represents a priority encoder in which the highest-priority request is indicated as the output of the circuit. (Encoder circuits are described in Chapter 6.) The output, *f*, of the priority encoder comprises two bits whose values depend on the three inputs, *req*1, *req*2, and *req*3. If *req*1 is 1, then *f* is set to 01. If *req*2 is 1, then *f* is set to 10, but only if *req*1 is not also 1. Hence *req*1 has higher priority than *req*2. Similarly, *req*1 and *req*2 have higher priority than *req*3. Thus if *req*3 is 1, then *f* is 11, but only if neither *req*1 nor *req*2 is also 1. For this priority encoder, if none of the three inputs is 1, then *f* is assigned the value 00.

A.7.5 GENERATE Statement

There are two variants of the GENERATE statement: the FOR GENERATE and the IF GENERATE. The general form of both types is shown in Figure A.13. The IF GENERATE statement is seldom needed, but FOR GENERATE is often used in practice. It provides a convenient way of repeating either a logic equation or a component instantiation. Figure A.14 illustrates its use for component instantiation. The code in the figure is equivalent to the code given in Figure A.7.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY priority IS
    PORT ( req1, req2, req3  : IN   STD_LOGIC ;
                f            : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    f <= "01" WHEN req1 = '1' ELSE
         "10" WHEN req2 = '1' ELSE
         "11" WHEN req3 = '1' ELSE
         "00" ;
END Behavior;
```

Figure A.12 A priority encoder described with a conditional signal assignment.

```
generate_label:
FOR index_variable IN range GENERATE
        statement ;
        {statement ;}
END GENERATE ;


generate_label:
IF expression GENERATE
        statement ;
        {statement ;}
END GENERATE ;
```

Figure A.13 The general forms of the GENERATE statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder IS
       PORT ( Cin    : IN    STD_LOGIC ;
              X, Y   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
              S      : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
              Cout   : OUT   STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
       SIGNAL C : STD_LOGIC_VECTOR(0 TO 4) ;
BEGIN
       C(0) <= Cin ;
       Generate_label:
       FOR i IN 0 TO 3 GENERATE
              bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
       END GENERATE ;
       Cout <= C(4) ;
END Structure ;
```

Figure A.14 An example of component instantiation with FOR GENERATE.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY addern IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( Cin    : IN    STD_LOGIC ;
           X, Y   : IN    STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           S      : OUT   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Cout   : OUT   STD_LOGIC ) ;
END addern ;

ARCHITECTURE Structure OF addern IS
    SIGNAL C : STD_LOGIC_VECTOR(0 TO n) ;
BEGIN
    C(0) <= Cin ;
    Generate_label:
    FOR i IN 0 TO n−1 GENERATE
        stage: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
    END GENERATE ;
    Cout <= C(4) ;
END Structure ;
```

Figure A.15 An *n*-bit adder.

A.8 Defining an Entity with GENERICs

The code in Figure A.14 represents an adder for four-bit numbers. It is possible to make this code more general by introducing a parameter in the code that represents the number of bits in the adder. In VHDL jargon such a parameter is called a GENERIC. Figure A.15 gives the code for an *n*-bit adder entity, named *addern*. The GENERIC keyword is used to define the number of bits, *n*, to be added. This parameter is used in the code, both in the definitions of the signals X, Y, and S and in the FOR GENERATE statement that instantiates the *n* full-adders.

It is possible to use the GENERIC feature with components that are instantiated as subcircuits in other code. In section A.10.10 we give an example that uses the *addern* entity as a subcircuit.

A.9 Sequential Assignment Statements

The order in which the concurrent assignment statements in an architecture body appear does not affect the meaning of the code. Many types of logic circuits can be described using these statements. However, VHDL also provides another type of statements, called *sequential assignment statements*, for which the order of the statements in the code can affect the semantics of the code. There are three variants of the sequential assignment statements: IF statement, CASE statement, and loop statements.

A.9.1 PROCESS STATEMENT

Since the order in which the sequential statements appear in VHDL code is significant, whereas the ordering of concurrent statements is not, the sequential statements must be separated from the concurrent statements. This is accomplished using a PROCESS statement. The PROCESS statement appears inside an architecture body, and it encloses other statements within it. The IF, CASE, and LOOP statements can appear only inside a process. The general form of a PROCESS statement is shown in Figure A.16. Its structure is somewhat similar to an architecture. VARIABLE data objects can be declared (only) inside the process. Any variable declared can be used only by the code within the process; we say that the *scope* of the variable is limited to the process. To use the value of such a variable outside the process, the variable's value can be assigned to a signal. The various elements of the process are best explained by giving some examples. But first we need to introduce the IF, CASE, and LOOP statements.

The IF, CASE, and LOOP statements can be used to describe either combinational or sequential circuits. We will introduce these statements by giving some examples of combinational circuits because they are easier to understand. Sequential circuits are described in section A.10.

A.9.2 IF STATEMENT

The general form of an IF statement is given in Figure A.17. An example using an IF statement for combinational logic is

```
[process_label:]
PROCESS [( signal name {, signal name} )]
        [VARIABLE declarations]
BEGIN
        [WAIT statement]
        [Simple Signal Assignment Statements]
        [Variable Assignment Statements]
        [IF Statements]
        [CASE Statements]
        [LOOP Statements]
END PROCESS [process_label] ;
```

Figure A.16 The general form of a PROCESS statement.

```
IF expression THEN
    statement ;
    {statement ;}
ELSIF expression THEN
    statement ;
    {statement ;}
ELSE
    statement ;
    {statement ;}
END IF ;
```

Figure A.17 The general form of an IF statement.

```
IF Sel = '0' THEN
    f <= x1 ;
ELSE
    f <= x2 ;
END IF ;
```

This code defines the 2-to-1 multiplexer that was used as an example of a selected signal assignment in the previous section. Examples of sequential logic described with IF statements are given in section A.10.

A.9.3 CASE Statement

The general form of a CASE statement is shown in Figure A.18. The *constant_value* can be a single value, such as 2, a list of values separated by the | pipe, such as 2|3, or a range, such as 2 to 4. An example of a CASE statement used to describe combinational logic is

```
CASE expression IS
    WHEN constant_value =>
        statement ;
        {statement ;}
    WHEN constant_value =>
        statement ;
        {statement ;}
    WHEN OTHERS =>
        statement ;
        {statement ;}
END CASE ;
```

Figure A.18 The general form of a CASE statement.

```
CASE Sel IS
    WHEN '0' =>
        f <= x1 ;
    WHEN OTHERS =>
        f <= x2 ;
END CASE ;
```

This code represents the same 2-to-1 multiplexer described in section A.9.2 using the IF statement. Similar to a selected signal assignment, all possible valuations of the expression used for the WHEN clauses must be listed; hence the OTHERS keyword is needed. Also, all WHEN clauses in the CASE statement must be mutually exclusive. Examples of sequential circuits described with the CASE statement are given in section A.10.11.

A.9.4 Loop Statements

VHDL provides two types of loop statements: the FOR-LOOP statement and the WHILE-LOOP statement. Their general forms are shown in Figure A.19. These statements are used to repeat one or more sequential assignment statements in much the same way as a FOR GENERATE statement is used to repeat concurrent assignment statements. Examples of the FOR-LOOP are given in section A.9.7.

A.9.5 Using a Process for a Combinational Circuit

An example of a PROCESS statement is shown in Figure A.20. It includes the code for the IF statement from section A.9.2. The signals *Sel*, *x*1, and *x*2 are shown in parentheses after the PROCESS keyword. They indicate which signals the process depends on and are called the *sensitivity list* of the process. For a process that describes combinational logic, as in this example, the sensitivity list includes all input signals used inside the process.

```
[loop_label:]
FOR variable_name IN range LOOP
    statement ;
    {statement ;}
END LOOP [loop_label] ;


[loop_label:]
WHILE boolean_expression LOOP
    statement ;
    {statement ;}
END LOOP [loop_label] ;
```

Figure A.19 The general forms of FOR-LOOP and WHILE-LOOP statements.

```
PROCESS ( Sel, x1, x2 )
BEGIN
    IF Sel =  ’0’ THEN
        f <=  x1
    ELSE
        f <=  x2 ;
    END IF ;
END PROCESS ;
```

Figure A.20 A PROCESS statement.

In VHDL jargon a process is described as follows. When the value of a signal in the sensitivity list changes, the process becomes *active*. Once active, the statements inside the process are "evaluated" in sequential order. Any signal assignments made in the process take effect only after all the statements inside the process have been evaluated. We say that the signal assignment statements inside the process are *scheduled* and will take effect at the end of the process.

The process describes a logic circuit and is translated into logic equations in the same manner as the concurrent assignment statements in an architecture body. The concept of the process statements being evaluated in sequence provides a convenient way of understanding the semantics of the code inside a process. In particular, a key concept is that if multiple assignments are made to a signal inside a process, only the last one to be evaluated has any effect. This is illustrated in the next example.

A.9.6 Statement Ordering

The IF statement in Figure A.20 describes a multiplexer that assigns either of two inputs, *x*1 or *x*2, to the output *f*. Another way of describing the multiplexer with an IF statement is shown in Figure A.21. The statement "f <= x1 ;" is evaluated first. However, the signal *f* may not actually be changed to the value of *x*1, because there may be a subsequent assignment to *f* in the code inside the process statement. At this point in the process, *x*1 represents the *default* value for *f* if no other assignment to *f* is evaluated. If we assume

```
PROCESS ( Sel, x1, x2 )
BEGIN
    f <=  x1 ;
    IF Sel = 1 THEN
        f <=  x2 ;
    END IF ;
END PROCESS ;
```

Figure A.21 An example illustrating the ordering of statements within a PROCESS.

that *Sel* = 1, then the statement "f <= x2 ;" will be evaluated. The effect of this second assignment to *f* is to override the default assignment. Hence the result of the process is that *f* is set to the value *x*2 when *Sel* = 1. If we assume that *Sel* = 0, then the IF condition fails and *f* is assigned its default value, *x*1.

This example illustrates the effect of the ordering of statements inside a process. If the two statements were reversed in order, then the IF statement would be evaluated first and the statement "f <= x1 ;" would be evaluated last. Hence the process would always result in *f* being set to the value of *x*1.

Implied Memory

Consider the process in Figure A.22. It is the same as the process in Figure A.21 except that the default assignment statement "f <= x1 ;" has been removed. Because the process does not specify a default value for *f*, and there is no ELSE clause in the IF statement, the meaning of the process is that *f* should retain its present value when the IF condition is not satisfied. The following expression is generated by the VHDL compiler for this process

$$f = Sel \cdot x2 + \overline{Sel} \cdot f$$

Hence when *Sel* = 0, the value of *x*2 is "remembered" at the output *f*. In VHDL jargon this is called *implied memory* or *implicit memory*. Although it is rarely useful for combinational circuits, we will show shortly that implied memory is the key concept used to describe sequential circuits.

A.9.7 USING A VARIABLE IN A PROCESS

We mentioned earlier that VHDL provides VARIABLE data objects, in addition to SIGNAL data objects. Unlike a signal, a variable data object does not represent a wire in a circuit. Therefore, a variable can be used to describe the functionality of a logic circuit in ways that are not possible using a signal. This concept is illustrated in Figure A.23. The intent of the code is to describe a logic circuit that counts the number of bits in the three-bit signal *X* that are equal to 1. The count is output using the signal called *Count*, which is a two-bit unsigned integer. Notice that *Count* is declared with the mode *Buffer* because it is used in the architecture body on both the left and right sides of an assignment operator. Table A.2 explains the meaning of the *Buffer* mode.

```
PROCESS ( Sel, x2 )
BEGIN
    IF Sel = 1 THEN
        f <=  x2 ;
    END IF ;
END PROCESS ;
```

Figure A.22 An example of implied memory.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY numbits IS
    PORT ( X        : IN        STD_LOGIC_VECTOR(1 TO 3) ;
           Count    : BUFFER    INTEGER RANGE 0 TO 3 ) ;
END numbits ;

ARCHITECTURE Behavior OF numbits IS
BEGIN
    PROCESS ( X )    -- count the number of bits in X with the value 1
    BEGIN
        Count <= 0 ;    -- the 0 with no quotes is a decimal number
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END LOOP ;
    END PROCESS ;
END Behavior ;
```

Figure A.23 A FOR-LOOP that does not represent a sensible circuit.

Inside the process, *Count* is initially set to 0. No quotes are used for the number 0 in this case, because VHDL allows a decimal number, which we said in section A.2.2 is denoted with no quotes, to be assigned to an INTEGER signal. The code gives a FOR-LOOP with the loop index variable *i*. For the values of *i* from 1 to 3, the IF statement inside the FOR-LOOP checks the value of bit X(i); if it is 1, then the value of *Count* is incremented. The code given in the figure is legal VHDL code and can be compiled without generating any errors. However, it will not work as intended, and it does not represent a sensible logic circuit.

There are two reasons why the code in Figure A.23 will not work as intended. First, there are multiple assignment statements for the signal *Count* within the process. As explained for the previous example, only the last of these assignments will have any effect. Hence if any bit in *X* is 1, then the statement "Count <= '0' ;" will not have the desired effect of initializing *Count* to 0, because it will be overridden by the assignment statement in the FOR-LOOP. Also, the FOR-LOOP will not work as desired, because each iteration for which $X(1)$ is 1 will override the effect of the previous iteration. The second reason why the code is not sensible is that the statement "Count <= Count + '1' ;" describes a circuit with feedback. Since the circuit is combinational, such feedback will result in oscillations and the circuit will not be stable.

The desired behavior of the VHDL code in Figure A.23 can be achieved using a variable, instead of a signal. This is illustrated in Figure A.24, in which the variable *Tmp* is used instead of the signal *Count* inside the process. The value of *Tmp* is assigned to *Count* at the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Numbits IS
    PORT ( X        : IN    STD_LOGIC_VECTOR(1 TO 3) ;
           Count    : OUT   INTEGER RANGE 0 TO 3 ) ;
END Numbits ;

ARCHITECTURE Behavior OF Numbits IS
BEGIN
    PROCESS ( X )    - - count the number of bits in X equal to 1
        VARIABLE TMP : INTEGER ;
    BEGIN
        Tmp := 0 ;
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Tmp := Tmp + 1 ;
            END IF ;
        END LOOP ;
        Count <= Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.24 The FOR-LOOP from Figure A.23 using a variable.

end of the process. Observe that the assignment statements to *Tmp* are indicated with the := operator, as opposed to the <= operator. The := is called the *variable assignment operator*. Unlike <=, it does not result in the assignment being *scheduled* until the end of the process. The variable assignment takes place immediately. This *immediate* assignment solves the first of the two problems with the code in Figure A.23. The second problem is also solved by using a variable instead of a signal. Because the variable does not represent a wire in a circuit, the FOR-LOOP need not be literally interpreted as a circuit with feedback. By using the variable, the FOR-LOOP represents only a desired *behavior*, or *functionality*, of the circuit. When the code is translated, the VHDL compiler will generate a combinational circuit that implements the functionality expressed in the FOR-LOOP.

When the code in Figure A.24 is translated by the VHDL compiler, it produces the circuit with 2 two-bit adders shown in Figure A.25. It is possible to see how this circuit corresponds to the FOR-LOOP in the code. The result of the first iteration of the loop is that *Count* is set to the value of $X(1)$. The second iteration then adds $X(1)$ to $X(2)$. This is realized by the top adder in the figure. The third iteration adds $X(3)$ to the sum produced from the second iteration. This corresponds to the bottom adder. When this circuit is optimized by the logic synthesis algorithms, the resulting expressions for *Count* are

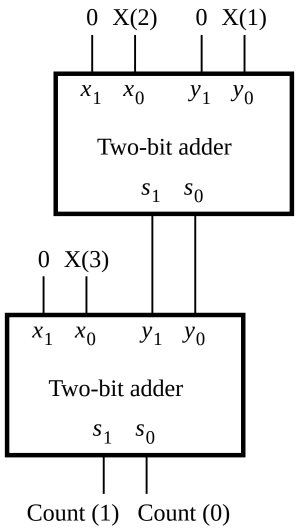


Figure A.25 The circuit generated from the code in Figure A.24.

$$\text{Count}(1) = X(1)X(2) + X(1)X(3) + X(2)X(3)$$
$$\text{Count}(0) = X(1) \oplus X(2) \oplus X(3)$$

These expressions represent a full-adder circuit, with *Count*(0) as the sum output and *Count*(1) as the carry-out. It is interesting to note that even though the VHDL code describes the desired behavior of the circuit in an abstract way, using a FOR-LOOP, in this example the logic synthesis algorithms produce the most efficient circuit, which is the full-adder. As we said at the beginning of this appendix and in section 2.9, the style of code in Figure A.24 should be avoided, because it is often difficult for the designer to envisage what logic circuit the code represents.

As another example of the use of a variable, Figure A.26 gives the code for an *n*-bit NAND gate entity, named *NANDn*. The number of inputs to the NAND gate is set by the GENERIC parameter *n*. The inputs are the *n*-bit signal X, and the output is f. The variable *Tmp* is defined in the architecture and originally set to the value of the input signal $X(1)$. In the FOR LOOP, *Tmp* is ANDed successively with input signals X (2) to X (n). Since *Tmp* is a variable data object, assignments to it take effect immediately; they are not scheduled to take effect at the end of the process. The complement of *Tmp* is assigned to f, thus completing the description of the *n*-input NAND operation.

Figure A.27 shows the same code given in Figure A.26 but with the data object *Tmp* defined as a signal, instead of as a variable. This code gives a wrong result, because only the last statement included in the process has any effect on *Tmp*. The code results in $Tmp = Tmp \cdot X(4)$, as determined by the last iteration of the FOR LOOP. Also, since *Tmp* is never initialized, its value is unknown. Hence the value of the output $f = \overline{Tmp}$ is unknown.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X   : IN    STD_LOGIC_VECTOR(1 TO n) ;
           f   : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
BEGIN
    PROCESS ( X )
        VARIABLE Tmp : STD_LOGIC ;
    BEGIN
        Tmp := X(1) ;
        AND_bits: FOR i IN 2 TO n LOOP
            Tmp := Tmp AND X(i) ;
        END LOOP AND_bits ;
        f <= NOT Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.26 Using a variable to describe an n-input NAND gate.

Figure A.28 shows one way to describe the n-input NAND gate using signals. Here *Tmp* is defined as an n-bit signal, which is set to contain n 1s using the (OTHERS => ’1’) construct. The conditional signal assignment specifies that f is 0 only if all bits in the input X are 1, thus describing the NAND operation.

A final example of variables used in a sequential circuit is given in section A.10.8. In general, using both variables and signals in VHDL code can lead to confusion because they imply different semantics. Since variables do not necessarily represent wires in a circuit, the meaning of code that uses variables is sometimes ill defined. To avoid confusion, in this book we use variables only for the loop indices in FOR GENERATE and FOR LOOP statements. Except for similar purposes, the reader should avoid using variables because they are not needed for describing logic circuits.

A.10 Sequential Circuits

Although combinational circuits can be described using either concurrent or sequential assignment statements, sequential circuits can be described only with sequential assignment statements. We now give some representative examples of sequential circuits.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X   : IN    STD_LOGIC_VECTOR(1 TO n) ;
           f   : OUT   STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
    SIGNAL Tmp : STD_LOGIC ;
BEGIN
    PROCESS ( X )
    BEGIN
        Tmp <= X(1) ;
        AND_bits: FOR i IN 2 TO n LOOP
            Tmp <= Tmp AND X(i) ;
        END LOOP AND_bits ;
        f <= NOT Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.27 The code from Figure A.26 using a signal.

A.10.1 A Gated D Latch

Figure A.29 gives the code for a gated D latch. The process sensitivity list includes both the latch's data input, *D*, and clock, *clk*. Hence whenever a change occurs in the value of either *D* or *clk*, the process becomes active. The IF statement specifies that Q should be set to the value of *D* whenever the clock is 1. There is no ELSE clause in the IF statement. As we explained for Figure A.22, this implies that Q should retain its present value when the IF condition is not met.

A.10.2 D Flip-Flop

Figure A.30 gives a process that is slightly different from the one in Figure A.29. The sensitivity list includes only the *Clock* signal, which means that the process is active only when the value of *Clock* changes. The condition in the IF statement looks unusual. The syntax Clock'EVENT represents a *change* in the value of the clock signal. In VHDL jargon 'EVENT is called an *attribute*, and combining 'EVENT with a signal name, such as *Clock*, yields a logical condition. The combination in the IF statement of the two conditions Clock'EVENT and Clock = '1' specifies that Q should be assigned the value of *D* when "a change occurs in the value of *Clock*, and *Clock* is now 1". This describes a low-to-high transition of the clock signal; hence the code describes a positive-edge-triggered D flip-flop.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X   : IN    STD_LOGIC_VECTOR(1 TO n) ;
           f   : OUT   STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
    SIGNAL Tmp : STD_LOGIC_VECTOR(1 TO n) ;
BEGIN
    Tmp <= (OTHERS => '1') ;
    f <= '0' WHEN X = Tmp ELSE '1' ;
END Behavior ;
```

Figure A.28 Using a signal to describe an *n*-input NAND gate.

The *std_logic_1164* package defines the two functions named *rising_edge* and *falling_edge*. They can be used as a short-form notation for the condition that checks for the occurrence of a clock edge. In Figure A.30 we could replace the line "IF Clock'EVENT AND Clock = '1' THEN" with the equivalent line "IF rising_edge(Clock) THEN". We do not use *rising_edge* or *falling_edge* in this book; they are mentioned for completeness.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
    PORT ( D, clk  : IN    STD_LOGIC ;
           Q       : OUT   STD_LOGIC ) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
    PROCESS ( D, clk )
    BEGIN
        IF clk = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.29 A gated D Latch.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock   : IN    STD_LOGIC ;
                Q           : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.30 D flip-flop.

A.10.3 Using a WAIT UNTIL Statement

The process in Figure A.31 uses a different syntax to describe a D flip-flop. Synchronization with the clock edge is specified using the statement "WAIT UNTIL Clock = '1' ;". This statement should be read as "wait for the next positive-edge of the clock signal." A process that uses a WAIT UNTIL statement is a special case because the sensitivity list is omitted. Use of this WAIT UNTIL statement implicitly specifies that the sensitivity list includes only *Clock*. For our purposes, which is using VHDL for synthesis of circuits, a process can include a WAIT UNTIL statement only if it is the first statement in the process.

A.10.4 A Flip-Flop with Asynchronous Reset

Figure A.32 gives a process that is similar to the one in Figure A.30. It describes a D flip-flop with an asynchronous reset, or clear, input. The reset signal has the name *Resetn*. When $Resetn = 0$, the flip-flop output Q is set to 0. Appending the letter *n* to a signal name is a widely used convention to denote an active-low signal.

A.10.5 Synchronous Reset

Figure A.33 shows how a flip-flop with a synchronous reset input can be described.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN    STD_LOGIC ;
           Q         : OUT   STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock = '1' ;
        Q <= D ;
    END PROCESS ;
END Behavior ;
```

Figure A.31 Equivalent code to Figure A.30, using a WAIT UNTIL statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN    STD_LOGIC ;
           Q                 : OUT   STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.32 D flip-flop with asynchronous reset.

A.10.6 INSTANTIATING A FLIP-FLOP FROM A LIBRARY

Because flip-flops are widely used in logic circuits, most CAD systems provide an assortment of flip-flop components that can be instantiated in VHDL code. An example of this is provided in Figure A.34. It uses a package named *maxplus2* in the library called *altera*. The maxplus2 package is part of the MAX+plusII system and includes many types of basic circuit elements. Figure A.34 instantiates the component named *dff*, which is a D flip-flop declared in the *maxplus2* package. The documentation provided in MAX+plusII specifies that the *dff* component has active-low asynchronous reset and preset inputs.

A.10.7 REGISTERS

One possible approach for describing a multibit register is to create an entity that instantiates multiple flip-flops. A more convenient method is illustrated in Figure A.35. It gives the same code shown in Figure A.32 but using the four-bit STD_LOGIC_VECTOR input *D* and the four-bit output Q. The code describes a four-bit register with asynchronous clear.

Figure A.36 gives the code for an entity named *regn*. It shows how the code in Figure A.35 can be extended to represent an *n*-bit register. The number of flip-flops is set by the generic parameter *n*.

The code in Figure A.37 shows how an enable input can be added to the *n*-bit register from Figure A.36. When the active clock edge occurs, the flip-flops in the register cannot

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN    STD_LOGIC ;
           Q                 : OUT   STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock = '1' ;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.33 D flip-flop with synchronous reset.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY altera ;
USE altera.maxplus2.all ;

ENTITY flipflop IS
    PORT ( D, Clock          : IN     STD_LOGIC ;
           Resetn, Presetn   : IN     STD_LOGIC ;
           Q                 : OUT    STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    Dff_instance: Dff PORT MAP (
        D, Clock, Resetn, Presetn, Q ) ;
END Behavior ;
```

Figure A.34 Instantiating a D flip-flop component.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg4 IS
    PORT ( D             : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Resetn, Clock : IN    STD_LOGIC ;
           Q             : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END reg4 ;

ARCHITECTURE Behavior OF reg4 IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= "0000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.35 Code for a four-bit register with asynchronous clear.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( D              : IN    STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Resetn, Clock  : IN    STD_LOGIC ;
           Q              : OUT   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.36 Code for an *n*-bit register with asynchronous clear.

change their stored values if the enable E is 0. If $E = 1$, the register responds to the active clock edge in the normal way.

A.10.8 SHIFT REGISTERS

An example of code that defines a four-bit shift register is shown in Figure A.38. The lines of code are numbered for ease of reference. The shift register has a serial input, *w*, and parallel outputs, Q. The right-most bit in the register is Q (4), and the left-most bit is Q (1); shifting is performed in the right-to-left direction. The architecture declares the signal *Sreg*, which is used to describe the shift operation. All assignments to *Sreg* are synchronized to the clock edge by the IF condition; hence *Sreg* represents the outputs of flip-flops. The statement in line 13 specifies that *Sreg*(4) is assigned the value of *w*. As we explained previously, this assignment does not take effect immediately but is scheduled to occur at the end of the process. In line 14 the current value of *Sreg*(4), before it is shifted as a result of line 13, is assigned to *Sreg*(3). Lines 15 and 16 complete the shift operation. They assign the current values of *Sreg*(3) and *Sreg*(2), before they are changed as a result of lines 14 and 15, to *Sreg*(2) and *Sreg*(1), respectively. Finally, *Sreg* is assigned to the Q outputs.

The key point that has to be appreciated in the code in Figure A.38 is that the assignment statements in lines 13 to 16 do not take effect until the end of the process. Hence all flip-

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( D          : IN    STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Resetn     : IN    STD_LOGIC ;
           E, Clock   : IN    STD_LOGIC ;
           Q          : OUT   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF E = '1' THEN
                Q <= D ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.37 VHDL code for an *n*-bit register with an enable input.

flops change their values at the same time, as required in the shift register. We could write the statements in lines 13 to 16 in any order without changing the meaning of the code.

In section A.9.7 we introduced variables and showed how they differ from signals. As another example of the semantics involved using variables, Figure A.39 gives the code from Figure A.38 but with *Sreg* declared as a variable, instead of as a signal. The statement in line 13 assigns the value of *w* to *Sreg* (4). Since *Sreg* is a variable, the assignment takes effect immediately. In line 14 the value of *Sreg* (4), which has already been changed to *w*, is assigned to *Sreg* (3). Hence line 14 results in *Sreg* (3) $= w$. Similarly, lines 15 and 16 set *Sreg* (2) and *Sreg* (1) to the value of *w*. The code does not describe the desired shift register, but rather loads all flip-flops with the value on the input *w*.

For the code in Figure A.39 to correctly describe a shift register, the ordering of lines 13 to 16 has to be reversed. Then the first assignment sets *Sreg* (1) to the value of *Sreg* (2), the second sets *Sreg* (2) to the value of *Sreg* (3), and so on. Each successive assignment is not affected by the one that precedes it; hence the semantics of using variables does not cause a problem. As we said in section A.9.7, it can be confusing to use both signals and variables at the same time because they imply different semantics.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
    PORT ( w, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC_VECTOR(1 TO 4) ) ;
END shift4 ;

ARCHITECTURE Behavior OF shift4 IS
    SIGNAL Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Sreg(4) <= w ;
            Sreg(3) <= Sreg(4) ;
            Sreg(2) <= Sreg(3) ;
            Sreg(1) <= Sreg(2) ;
        END IF ;
    END PROCESS ;
    Q <= Sreg ;
END Behavior ;
```

Figure A.38 Code for a four-bit shift register.

A.10.9 Counters

Figure A.40 shows the code for a four-bit counter with an asynchronous reset input. The counter also has an enable input. On the positive clock edge, if the enable E is 1, the count is incremented. If $E = 0$, the counter holds its current value. Because counters are commonly needed in logic circuits, most CAD systems provide a selection of counters that can be instantiated in a design. For example, MAX+plusII provides the counter defined by the LPM standard, which is a variable-width counter with options for enabling the counter, resetting the count to 0, and presetting the count to a specific number.

A.10.10 Using Subcircuits with GENERIC Parameters

We have shown several examples of VHDL entities that include generic parameters. When these subcircuits are used as components in other code, the generic parameters can be set to whatever values are needed. To give an example of component instantiation using generics, consider the circuit shown in Figure A.41. The circuit adds the binary number represented by the k-bit input X to itself a number of times. Such a circuit is often called an *accumulator*. To store the result of each addition operation, the circuit includes a k-bit register. The register has an asynchronous reset input, *Resetn*. It also has an enable input,

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
    PORT ( w, Clock  : IN    STD_LOGIC ;
               Q        : OUT STD_LOGIC_VECTOR(1 TO 4) ) ;
END shift4 ;

ARCHITECTURE Behavior OF shift4 IS
BEGIN
    PROCESS ( Clock )
        VARIABLE Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Sreg(4) := w ;
            Sreg(3) := Sreg(4) ;
            Sreg(2) := Sreg(3) ;
            Sreg(1) := Sreg(2) ;
        END IF ;
        Q <= Sreg ;
    END PROCESS ;
END Behavior ;
```

Figure A.39 The code from Figure A.38, using a variable.

E, which is controlled by a four-bit counter. The counter has an asynchronous clear input and a count enable input. The circuit operates by first clearing all bits in the register and counter to 0. Then in each clock cycle, the counter is incremented, and the sum outputs from the adder are stored in the register. When the counter reaches the value 1111, the enable inputs on both the register and counter are set to 0 by the NAND gate. Hence the circuit remains in this state until it is reset again. The final value stored in the register is equal to $15X$.

We can represent the accumulator circuit using several subcircuits described in this appendix: *addern* (Figure A.15), *NANDn* (Figure A.28), *regne*, and *count4*. We placed the component declaration statements for all of these subcircuits in one package, named *components*, which is shown in Figure A.42.

Complete code for the accumulator is given in Figure A.43. It uses the generic parameter k to represent the number of bits in the input X. Using this parameter in the code makes it easy to change the bit-width at a later time if desired. The architecture defines the signal *Sum* to represent the outputs of the adder; for simplicity, we ignore the possibility of arithmetic overflow and assume that the sum can be represented using k bits. The four-bit signal C represents the outputs from the counter. The *Stop* signal is connected to the enable inputs on the register and counter.

The statement labeled *adder* instantiates the *addern* subcircuit. The GENERIC MAP keywords are used to specify the value of the adder's generic parameter, n. The syntax

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY count4 IS
    PORT ( Resetn      : IN     STD_LOGIC ;
           E, Clock    : IN     STD_LOGIC ;
           Q           : OUT    STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
END count4 ;

ARCHITECTURE Behavior OF count4 IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

Figure A.40 An example of a counter.

(n => k) sets the number of bits in the adder to *k*. We do not need the carry-in port on the adder, but a signal must be connected to it. The signal *Zero_bit*, which is set to '0' in the code, is used as a placeholder for the carry-in port (the VHDL syntax does not permit a constant value, such as '1', to be associated directly with a port; hence a signal must be defined for this purpose). The *k*-bit data inputs to the adder are *X* and the output of the register, which is named *Result*. The sum output from the adder is named *Sum*, and the carry-out, which is not used in the circuit, is named *Cout*.

The *regne* subcircuit is instantiated in the statement labeled *reg*. GENERIC MAP is used to set the number of bits in the register to *k*. The *k*-bit register input is provided by the *Sum* output from the adder. The register's output is named *Result*; this signal represents the output of the accumulator circuit. It has the mode BUFFER in the entity declaration. This is required in the VHDL syntax for the signal to be connected to a port on an instantiated component.

The *count4* and *NANDn* components are instantiated in the statements labeled *Counter* and *NANDgate*. We do not have to use the GENERIC MAP keyword for *NANDn*, because the default value of its generic parameter is 4, which is the value needed in this application.

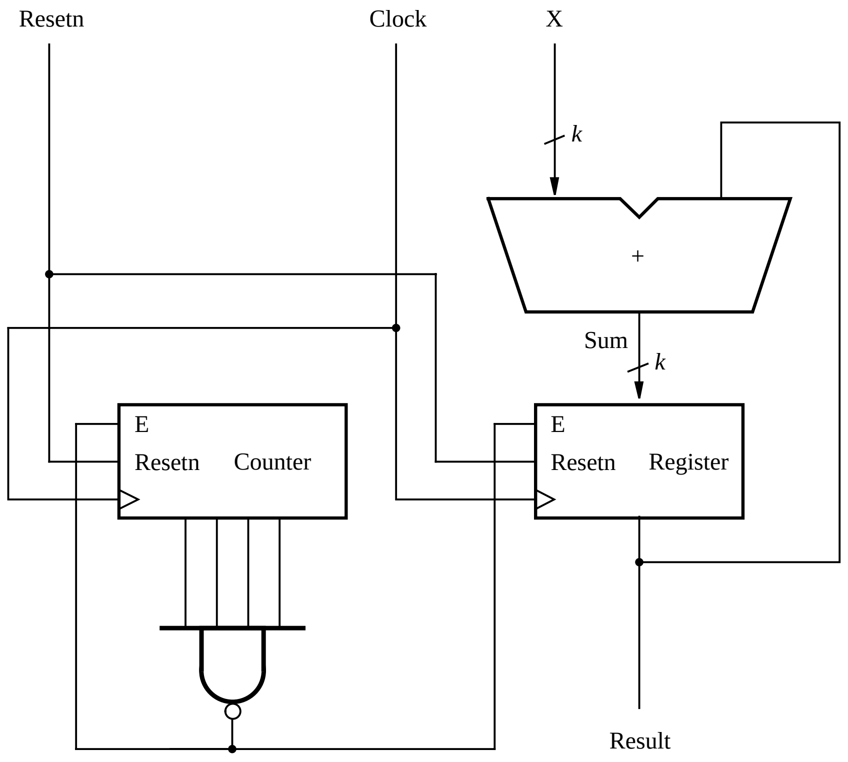


Figure A.41 The accumulator circuit.

A.10.11 A Moore-Type Finite State Machine

Figure A.44 shows the state diagram of a simple Moore machine. The code for this machine is shown in Figure A.45. The signal named *y* represents the state of the machine. It is declared with an enumerated type, *State_type*, that has the three possible values A, B, and C. When the code is compiled, the VHDL compiler automatically performs a state assignment to select appropriate bit patterns for the three states. The behavior of the machine is defined by the process with the sensitive list that comprises the reset and clock signals.

The VHDL code includes an asynchronous reset input that puts the machine in state A. The state table for the machine is defined using a CASE statement. Each WHEN clause corresponds to a present state of the machine, and the IF statement inside the WHEN clause specifies the next state to be reached after the next positive edge of the clock signal. Since the machine is of the Moore type, the output *z* can be defined as a separate concurrent assignment statement that depends only on the present state of the machine. Alternatively, the appropriate value for *z* could have been specified within each WHEN clause of the CASE statement.

An alternative way to describe a Moore-type finite state machine is given in the architecture in Figure A.46. Two signals are used to describe how the machine moves from one

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS

    COMPONENT addern - - n-bit adder
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( Cin   : IN   STD_LOGIC ;
               X, Y  : IN   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               S     : OUT  STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               Cout  : OUT  STD_LOGIC ) ;
    END COMPONENT ;

    COMPONENT regne - - n-bit register with enable
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( D         : IN   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               Resetn    : IN   STD_LOGIC ;
               E, Clock  : IN   STD_LOGIC ;
               Q         : OUT  STD_LOGIC_VECTOR(n−1 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT count4 - - 4-bit counter with enable
        PORT ( Resetn    : IN   STD_LOGIC ;
               E, Clock  : IN   STD_LOGIC ;
               Q         : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT NANDn - - n-bit AND gate
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( X  : IN   STD_LOGIC_VECTOR(1 TO n) ;
               f  : OUT  STD_LOGIC ) ;
    END COMPONENT ;

END components ;
```

Figure A.42 Component declarations for the accumulator circuit.

state to another state. The signal *y_present* represents the outputs of the state flip-flops, and the signal *y_next* represents the inputs to the flip-flops. The code has two processes. The top process describes a combinational circuit. It uses a CASE statement to specify the values that *y_next* should have for each value of *y_present*. The other process represents a sequential circuit, which specifies that *y_present* is assigned the value of *y_next* on the positive clock edge. The process also specifies that *y_present* should take the value *A* when *Resetn* is 0, which provides the asynchronous reset.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY accum IS
    GENERIC ( k : INTEGER := 8 ) ;
    PORT ( Resetn, Clock  : IN      STD_LOGIC ;
           X              : IN      STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
           Result         : BUFFER  STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
END accum ;

ARCHITECTURE Structure OF accum IS
    SIGNAL Sum : STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
    SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    SIGNAL Zero_bit, Cout, Stop : STD_LOGIC ;
BEGIN
    Zero_bit <= '0' ;
    adder: addern
        GENERIC MAP ( n => k )
        PORT MAP ( Zero_bit, X, Result, Sum, Cout ) ;
    reg: regne
        GENERIC MAP ( n => k )
        PORT MAP ( Sum, Resetn, Stop, Clock, Result ) ;
    Counter: count4
        PORT MAP ( Clock, Resetn, Stop, C ) ;
    NANDgate: NANDn
        PORT MAP ( C, Stop ) ;
END Structure ;
```

Figure A.43 Code for the accumulator circuit.

Although Figures A.45 and A.46 provide functionally equivalent code, when using the MAX+plusII CAD system, the code in Figure A.45 is preferable. MAX+plusII recognizes the code in Figure A.45 as a finite state machine. It reports all results of synthesizing or simulating the code in terms of the states of the machine. For example, when using the simulator CAD tool, the value of the *y* signal is reported using the names A, B, and C. If the code in Figure A.46 is used instead, then MAX+plusII reports only the logic values of the signals. For example, the value of the *y_present* signal is shown by the simulator as 00, or 01, and so on.

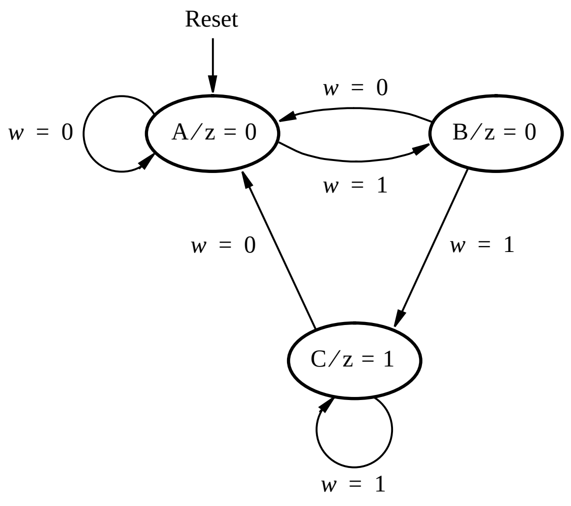


Figure A.44 State diagram of a simple Moore-type FSM.

A.10.12 A MEALY-TYPE FINITE STATE MACHINE

A state diagram for a simple Mealy machine is shown in Figure A.47. The corresponding code is given in Figure A.48. The code is the same as in Figure A.45 except that the output z is specified using a separate CASE statement. The CASE statement states that when the FSM is in state A, z should be 0, but when in state B, z should take the value of w. This CASE statement properly describes the logic needed for z. However, it is not obvious why we have used a second CASE statement in the code, rather than specify the value of z inside the CASE statement that defines the state table for the machine. This approach would not work properly because the CASE statement for the state table is nested inside the IF statement that waits for a clock edge to occur. Hence if we placed the code for z inside this CASE statement, then the value of z could change only as a result of a clock edge. This does not meet the requirements of the Mealy-type FSM, because the value of z depends not only on the state of the machine but also on the value of the input w.

A.10.13 MANUAL STATE ASSIGNMENT FOR A FINITE STATE MACHINE

Instead of having the VHDL compiler determine the state assignment, it is possible to encode the state bits manually. One way to do this in the MAX+plusII system is to use an ATTRIBUTE specification. An attribute provides information about a VHDL element, such as a type. An example showing how an attribute is used for a finite state machine is given in Figure A.49. The code represents the Moore machine from Figure A.45 with the addition of two ATTRIBUTE specifications. The attributes specify that the state encoding should be 00 for state A, 01 for state B, and 11 for state C.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY moore IS
    PORT ( Clock    : IN     STD_LOGIC ;
           w        : IN     STD_LOGIC ;
           Resetn   : IN     STD_LOGIC ;
           z        : OUT    STD_LOGIC ) ;
END moore ;

ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= C ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= C ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y = C ELSE '0' ;
END Behavior ;
```

Figure A.45 An example of a Moore-type finite state machine.

```
ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y_present, y_next : State_type ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
        END CASE ;
    END PROCESS ;

    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figure A.46 Code equivalent to Figure A.45, using two processes.

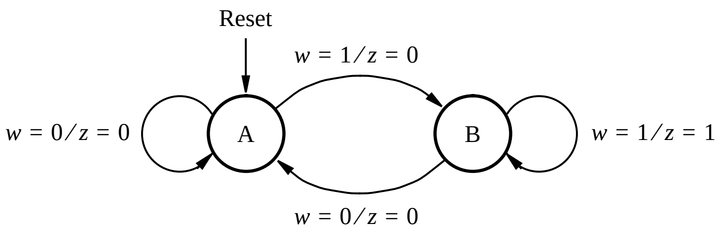


Figure A.47 State diagram of a Mealy-type FSM.

A.11 COMMON ERRORS IN VHDL CODE

This section lists some common errors that our students have made when writing VHDL code.

ENTITY and ARCHITECTURE Names

The name used in an ENTITY declaration and the corresponding ARCHITECTURE must be identical. The code

```
ENTITY adder IS
:
END adder ;

ARCHITECTURE Structure OF adder4 IS
:
END Structure ;
```

is erroneous because the ENTITY declaration uses the name *adder*, whereas the architecture uses the name *adder4*.

Missing Semicolon

Every VHDL statement must end with a semicolon.

Use of Quotes

Single quotes are used for single-bit data, double quotes for multibit data, and no quotes are used for integer data. Examples are given in section A.2.

Combinational versus Sequential Statements

Combinational statements include simple signal assignments, selected signal assignments, and generate statements. Simple signal assignments can be used either outside or inside a PROCESS statement. The other types of combinational statements can be used only outside a PROCESS statement.

Sequential statements include IF, CASE, and LOOP statements. Each of these types of statements can be used only inside a process statement.

Component Instantiation

The following statement contains two errors

```
control: shiftr GENERIC MAP ( K => 3 ) ;
                PORT MAP ( '1', Clock, w, Q ) ;
```

There should be no semicolon at the end of the first line, because the two lines represent a single VHDL statement. Also, it is illegal to associate a constant value ('1') with a port on a component. The following code shows how the two errors can be fixed

```
SIGNAL High ;
:
High <= '1' ;
control: shiftr GENERIC MAP ( K => 3 )
                PORT MAP ( High, Clock, w, Q ) ;
```

Label, Signal, and Variable Names

It is illegal to use any VHDL keyword as a label, signal, or variable name. For example, it is illegal to name a signal *In* or *Out*. Also, it is illegal to use the same name multiple times for any label, signal, or variable in a given VHDL design. A common error is to use the same name for a signal and a variable used as the index in a generate or loop statement. For instance, if the code uses the generate statement

```
Generate_label:
FOR i IN 0 TO 3 GENERATE
    bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
END GENERATE ;
```

then it is illegal to define a signal named *i* (or *I*, because VHDL does not distinguish between lower and uppercase letters).

Implied Memory

As shown in section A.10, implied memory is used to describe storage elements. Care must be taken to avoid unintentional implied memory. The code

```
IF LA = '1' THEN
    EA <= '1' ;
END IF ;
```

results in implied memory for the *EA* signal. If this is not desired, then the code can be fixed by writing

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mealy IS
    PORT ( Clock, Resetn  : IN   STD_LOGIC ;
           w              : IN   STD_LOGIC ;
           z              : OUT  STD_LOGIC ) ;
END mealy ;

ARCHITECTURE Behavior OF mealy IS
    TYPE State_type IS (A, B) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    PROCESS ( y, w )
    BEGIN
        CASE y IS
            WHEN A =>
                z <= '0' ;
            WHEN B =>
                z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure A.48 An example of a Mealy-type machine.

```
ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    ATTRIBUTE ENUM_ENCODING                   : STRING ;
    ATTRIBUTE ENUM_ENCODING OF State_type  : TYPE IS "00 01 11" ;
    SIGNAL y_present, y_next                  : State_type ;
BEGIN
    · · · etc.
```

Figure A.49 An example of specifying the state assignment manually.

```
IF LA = '1' THEN
    EA <= '1' ;
ELSE
    EA <= '0' ;
END IF ;
```

Implied memory also applies to CASE statements. The statement

```
CASE y IS
    WHEN S1 =>
        EA <= '1' ;
    WHEN S2 =>
        EB <= '1' ;
END CASE ;
```

does not specify the value of the *EA* signal when *y* is not equal to *S*1, and it does not specify the value of *EB* when *y* is not equal to *S*2. To avoid having implied memory for both *EA* and *EB*, these signals should be assigned default values, as in the code

```
EA <= '0' ; EB <= '0' ;
CASE y IS
    WHEN S1 =>
        EA <= '1' ;
    WHEN S2 =>
        EB <= '1' ;
END CASE ;
```

In general, the designer should attempt to write VHDL code that contains as few errors as possible because finding the source of an error can often be difficult.

A.12 CONCLUDING REMARKS

This appendix describes all the important VHDL constructs that are useful for the synthesis of logic circuits. As mentioned earlier, we do not discuss any features of VHDL that are useful only for simulation of circuits, or for other purposes. A reader who wishes to learn more about using VHDL can refer to specialized books [1–7].

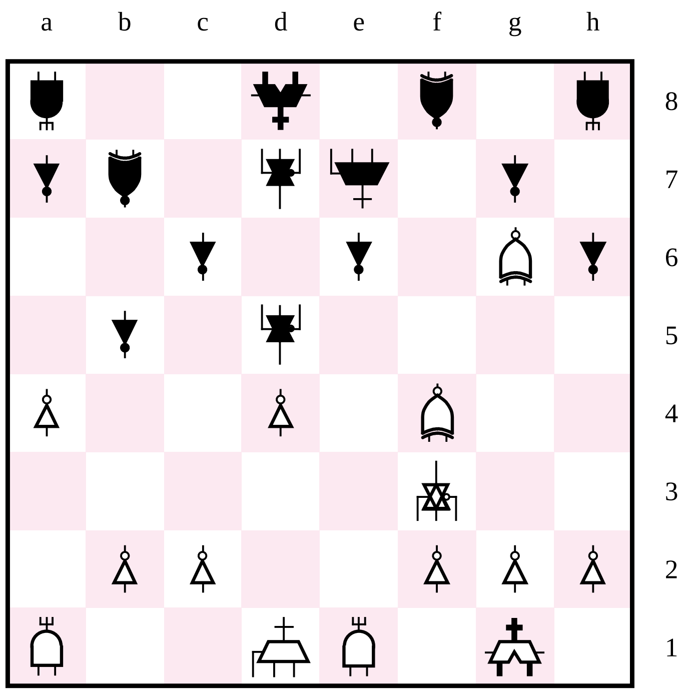
REFERENCES

1. Institute of Electrical and Electronics Engineers, “1076-1993 IEEE Standard VHDL Language Reference Manual,” 1993.
2. D. L. Perry, *VHDL*, 3rd ed. (McGraw-Hill: New York, 1998).
3. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems* (McGraw-Hill: New York, 1993).
4. J. Bhasker, *A VHDL Primer* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
5. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
6. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
7. S. Yalamanchili, *VHDL Starter’s Guide* (Prentice-Hall: Upper Saddle River, NJ, 1998).

appendix

B

Tutorial 1



13. Rf1–e1, Nf6–d5

MAX+plusII is one of the most sophisticated and easiest to use CAD systems available on the market. In this tutorial we introduce the design of logic circuits using MAX+plusII. Step-by-step instructions are presented for performing design entry with three methods: using schematic capture, writing VHDL code, and using a truth table. The tutorial also illustrates functional simulation.

B.1 INTRODUCTION

This tutorial assumes that the reader has access to a computer on which MAX+plusII is installed. Instructions for installing the copy of MAX+plusII provided with the book are included with the CD-ROM. The MAX+plusII software will run on several different types of computer systems. For this tutorial a computer running a Microsoft operating systems (Windows95, Windows98, or WindowsNT) is assumed. Although MAX+plusII operates similarly on all of the supported types of computers, there are some minor differences. A reader who is not using a Microsoft Windows operating system may experience some slight discrepancies from this tutorial. Examples of potential differences are the locations of files in the computer's file system and the exact appearance of windows displayed by the software. All such discrepancies are minor and will not affect the reader's ability to follow the tutorial.

This tutorial does not describe how to use the operating system provided on the computer. We assume that the reader already knows how to perform actions such as running programs, operating a mouse, moving, resizing, minimizing and maximizing windows, creating directories (folders) and files, and the like. A reader who is not familiar with these procedures will need to learn how to use the computer's operating system before proceeding.

B.1.1 GETTING STARTED

Each logic circuit, or subcircuit, being designed in MAX+plusII is called a *project*. The software works on one project at a time and keeps all information for that project in a single directory in the file system (we use the traditional term *directory* for a location in the file system, but in Microsoft Windows the term *folder* is used). To begin a new logic circuit design, the first step is to create a directory to hold its files. As part of the installation of the MAX+plusII software, a few sample projects are placed into a directory called *max2work*. To hold the design files for this tutorial, we created the subdirectory *max2work**tutorial1*. The location and name of the directory is not important; hence the reader may use any valid directory.

To create a directory to work in, use the normal utilities provided by the computer's operating system. MAX+plusII is not involved in this step. After the directory has been created, start the MAX+plusII software. You should see a window similar to the one in Figure B.1. This window is called the *MAX+plusII Manager*. It provides access to all the features of MAX+plusII, which the user selects with the computer mouse.

Most of the commands provided by MAX+plusII are accessed by using a set of menus that are located in the Manager window below the title bar. For example, in Figure B.1

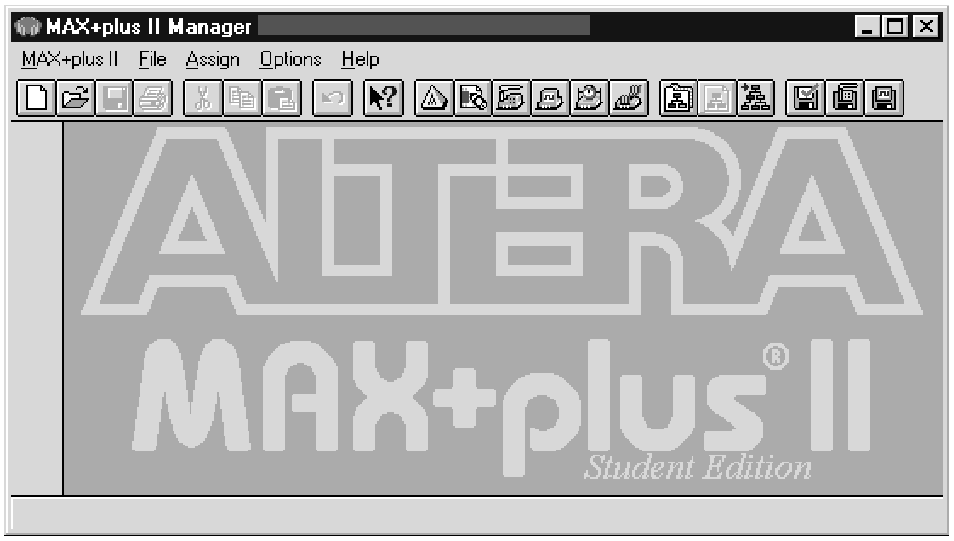


Figure B.1 The MAX+plusII Manager window.

clicking the left mouse button on the menu named File opens the menu shown in Figure B.2. Clicking the left mouse button on the entry Exit MAX+plusII Alt+F4 exits from MAX+plusII. In general, whenever the mouse is used to select something, the *left* button is used. Hence we will not normally specify which button to use. In the few cases when it is necessary to use the *right* mouse button, it will be specified explicitly. We should note that the Alt+F4 part of

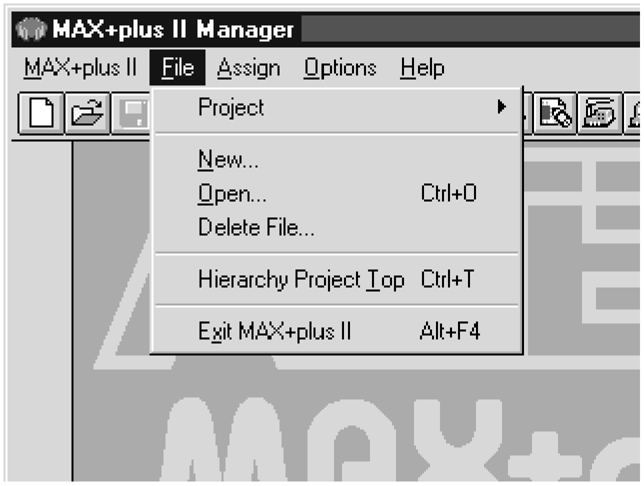


Figure B.2 The File menu in the Manager window.

the menu item indicates a keyboard shortcut; instead of using the mouse, the command can alternatively be invoked by the holding down the Alt key on the keyboard and pressing the F4 function key. Keyboard shortcuts are available for a few of the MAX+plusII commands, but commands are usually invoked using the mouse. For some commands it is necessary to access two or more menus in sequence. We use the convention Menu1 | Menu2 | Item to indicate that to select the desired command the user should first click the left mouse button on Menu1, then within this menu click on Menu2, and then within Menu2 click on Item. For example, File | Exit MAX+plusII describes how to use the mouse to exit from the MAX+plusII system.

The MAX+plusII system includes 11 main software modules, called *applications*. They can be accessed in two different ways. First, all the applications can be invoked via the MAX+plusII menu in the Manager window, as illustrated in Figure B.3. Second, some of the applications can be invoked using the small icons that appear below the Manager title bar. (If no icons are visible under the Manager title bar, select Options | Preferences to open the Preferences dialog box. Then use the mouse to place a check mark beside the entry for Show Toolbar and click OK.) To see which applications in Figure B.3 a particular icon is associated with, place the mouse pointer on top of the icon; the Manager displays a message near the bottom of the window that gives the name of the application.

The applications introduced in this tutorial include the Graphic Editor, Text Editor, Waveform Editor, Compiler, Simulator, Message Processor, and Hierarchy Display. The others are introduced in Tutorial 2.

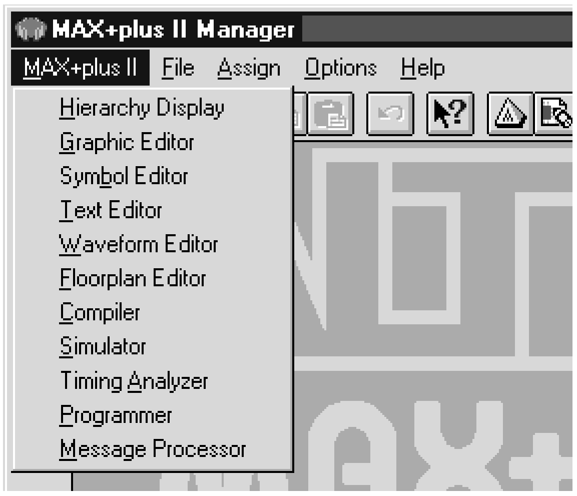


Figure B.3 The MAX+plus II menu in the Manager window.

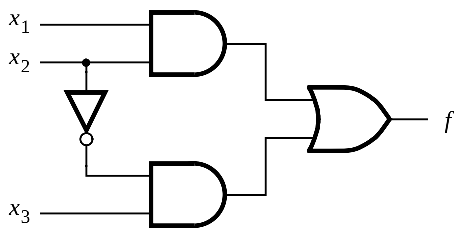
MAX+plusII On-Line Help

MAX+plusII provides comprehensive on-line documentation that answers most of the questions that may arise when using the software. The documentation is accessed from the Help menu in the Manager window. To get some idea of the extent of documentation provided, it is worthwhile for the reader to browse through the Help menu. For instance, selecting Help | MAX+plusII Table of Contents shows all the categories of documentation available.

The user can quickly search through the Help topics by selecting Help | Search for Help on, which opens a dialog box into which keywords can be entered. The available Help topics that match the keywords are automatically displayed. Two other methods are provided for quickly finding documentation for specific topics. First, while using any application, pressing the F1 function key on the keyboard opens a Help display that shows the commands available for that application. Second, in some instances holding down the Shift key and pressing the F1 key changes the mouse pointer into a *help* pointer. This feature is available when using the schematic capture tool provided in MAX+plusII. Clicking the help pointer on any circuit element in a schematic automatically displays any documentation that is available for that circuit element.

B.2 Design Entry Using Schematic Capture

In Chapter 2 we introduced three types of design entry methods: truth tables, schematic capture, and VHDL. This section illustrates the process of using the schematic capture tool provided in MAX+plusII, which is called the Graphic Editor. As a simple example, we will draw a schematic for the logic function $f = x_1x_2 + \overline{x}_2x_3$. A circuit diagram for f was shown in Figure 2.26 and is reproduced as Figure B.4*a*. The truth table for f is given in Figure B.4*b*. Chapter 2 also introduced functional simulation. After creating the schematic, we show how to use the functional simulator in MAX+plusII to verify the schematic's functionality.



(a) Circuit

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b) Truth table

Figure B.4 The logic function of Figure 2.26.

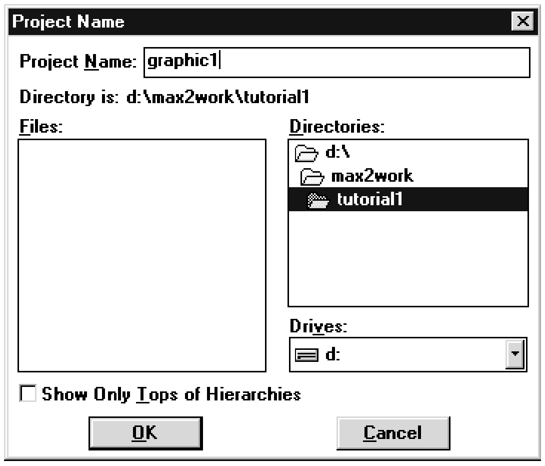
B.2.1 SPECIFYING THE PROJECT NAME

As a first step we will specify the name of the design project. In the Manager window select File | Project | Name to open the pop-up box illustrated in Figure B.5. It is necessary to specify the location of the directory where MAX+plusII will store any files created for the project. For this example the directory used is named d:\*max2work**tutorial1*. The disk drive designation, d:, is selected using the Drives pull-down menu shown in Figure B.5. The directory name is selected using the box labeled Directories. Use the mouse to double-click on the directory names displayed in the box until the proper directory is selected; the selected directory appears next to the words Directory is, as illustrated in the figure. In the box labeled Project Name, type *graphic1* as the name for this project and then click OK. Observe that the name of the project is displayed in the title bar of the Manager window.

B.2.2 USING THE GRAPHIC EDITOR

The next step is to draw the schematic. In the Manager window select MAX+plusII | Graphic Editor. The Graphic Editor window appears inside the Manager window. It may be helpful to move or resize the Graphic Editor window and to increase the size of the Manager window to provide more work space. In the screen capture in Figure B.6, the Graphic Editor window is maximized so that it fills the entire Manager window.

The title bar in Figure B.6 includes some menu names and icons that did not appear in Figure B.1. This is because the Manager window always indicates the features available in whatever application is currently being used. A number of icons that are used to invoke Graphic Editor features also appear along the left edge of the window. To see a description of the Graphic Editor feature associated with each icon, position the mouse on top of the icon; a message is displayed near the bottom of the window. Two of the most useful icons


Project Name

Project Name: graphic1

Directory is: d:\max2work\tutorial1

Figure B.5 Specifying the name and working directory for a project.

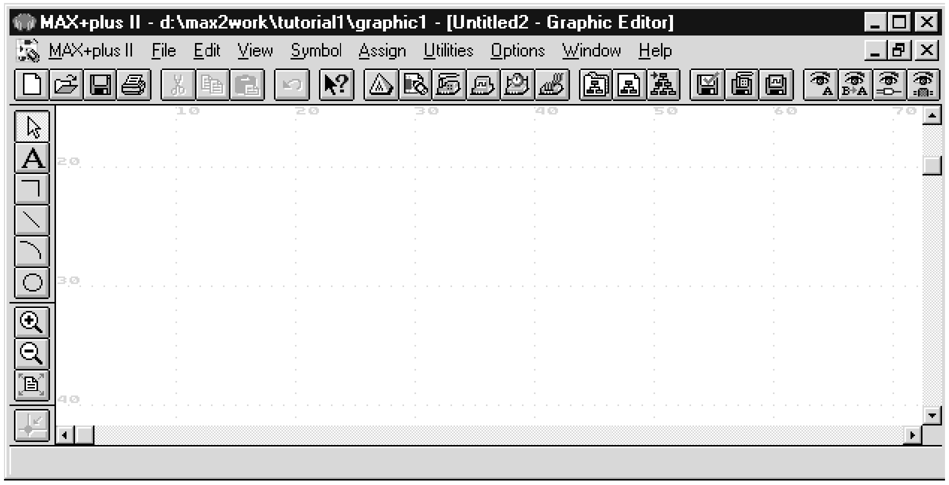
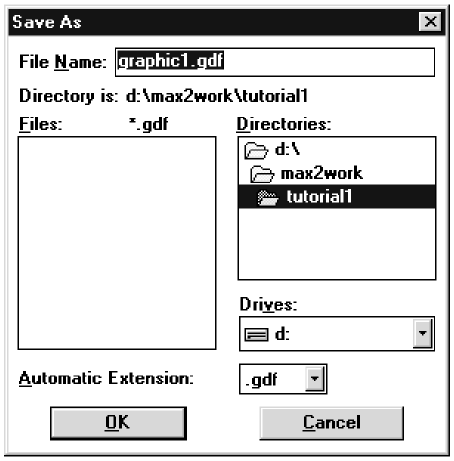

MAX+plus II - d:\max2work\tutorial1\graphic1 - [Untitled2 - Graphic Editor]
MAX+plus II File Edit View Symbol Assign Utilities Options Window Help

Figure B.6 The Graphic Editor display.

are the ones that look like a magnifying glass. These icons are used to see a larger or smaller view of the schematic.

Naming the Schematic

The schematic being created must be given a name. Select File | Save As to open the pop-up box depicted in Figure B.7. The directory that we chose for the project is already selected in the pop-up box. The Graphic Editor will create a separate file for the schematic and store it in the project's directory. In the box labeled File Name, type *graphic1.gdf*.


Save As

File Name: graphic1.gdf

Directory is: d:\max2work\tutorial1

Files: *.gdf

Directories: d:\ max2work tutorial1

Drives: d:

Automatic Extension: .gdf

Figure B.7 Specifying the name of a schematic.

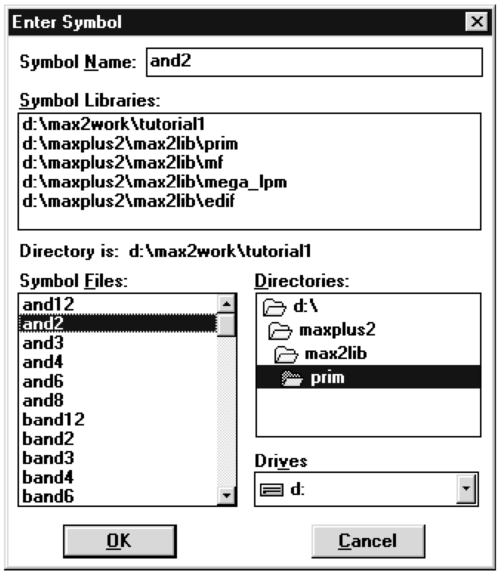
You must use exactly this name. The name *graphic1* must match the name of the project, and the filename extension *gdf*, which stands for *graphic design file*, must be used for all schematics. Click OK to return to the Graphic Editor.

Importing Logic-Gate Symbols

The Graphic Editor provides several libraries which contain circuit elements that can be imported into a schematic. For our simple example we will use a library called *Primitives*, which contains basic logic gates. To access the library, double-click on the blank space in the middle of the Graphic Editor display to open the pop-up box in Figure B.8 (another way to open this box is to select Symbol | Enter Symbol). The box labeled Symbol Libraries lists several available libraries, including the Primitives library. To open it, double-click on the line that ends with the word *prim*. A list of the logic gates in the library is automatically displayed in the Symbol Files box. Double-click on the *and2* symbol to import it into the schematic (you can alternatively click on *and2* and then click OK). A two-input AND-gate symbol now appears in the Graphic Editor window.

Any symbol in a schematic can be selected using the mouse. Position the mouse pointer on top of the AND-gate symbol in the schematic and click the mouse to select it. The symbol is highlighted in red. To move a symbol, select it and, while continuing to press the mouse button, drag the mouse to move the symbol. To make it easier to position the graphical symbols, a grid of guidelines can be displayed in the Graphic Editor window by selecting Options | Show Guidelines. Spacing between grid lines can be adjusted using Options | Guideline Spacing.

The logic function *f* requires a second two-input AND gate, a two-input OR gate, and a NOT gate. Use the following steps to import them into the schematic.


Enter Symbol

Symbol Name: and2

Symbol Libraries:
d:\max2work\tutorial1
d:\maxplus2\max2lib\prim
d:\maxplus2\max2lib\mf
d:\maxplus2\max2lib\mega_lpm
d:\maxplus2\max2lib\edif

Directory is: d:\max2work\tutorial1

Symbol Files:
and12
and2
and3
and4
and6
and8
band12
band2
band3
band4
band6

Directories:
d:\
maxplus2
max2lib
prim

Drives
d:

Figure B.8 Importing a logic gate from the Primitives library.

Position the mouse pointer over the AND-gate symbol that has already been imported. Press and hold down the Ctrl keyboard key and click and drag the mouse away from the AND-gate symbol. The Graphic Editor automatically imports a second instance of the AND-gate symbol. This shortcut procedure for making a copy of a circuit element is convenient when you need many instances of the same element in a schematic. Of course, an alternative approach is to import each instance of the symbol by opening the Primitives library as described above.

To import the OR-gate symbol, again double-click on a blank space in the Graphic Editor and then double-click on the Primitives library. In the box labeled Symbol Files, use the scroll bar to scroll down through the list of gates to find the symbol named *or2*. Import this symbol into the schematic. Next import the NOT gate using the same procedure. To orient the NOT gate so that it points downward, as depicted in Figure B.4*a*, select the NOT-gate symbol and then use the command Edit | Rotate | 270 to rotate the symbol 270 degrees counterclockwise. The symbols in the schematic can be moved by selecting them and dragging the mouse, as explained above. More than one symbol can be selected at the same time by clicking the mouse and dragging an outline around the symbols. The selected symbols are moved together by clicking on any one of them and moving it. Experiment with this procedure. Arrange the symbols so that the schematic appears similar to the one in Figure B.9.

Importing Input and Output Symbols

Now that the logic-gate symbols have been entered, it is necessary to import symbols to represent the input and output ports of the circuit. Open the Primitives library again. Click the mouse anywhere in the box labeled Symbol Files and then type the letter "i" to jump ahead in the list of symbols to those whose names begin with *i*. This shortcut can be used in addition to the scroll bars provided on the Symbol Files box. Import the symbol

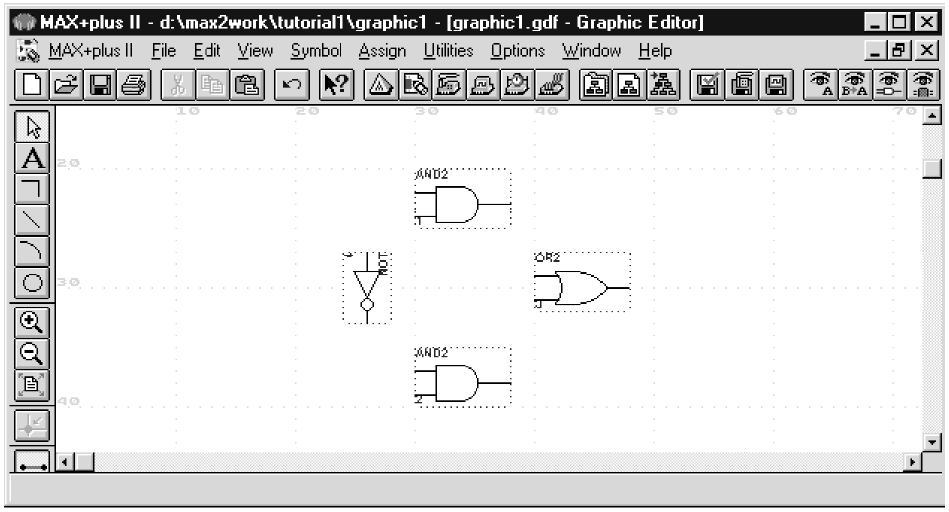

MAX+plus II - d:\max2work\tutorial1\graphic1 - [graphic1.gdf - Graphic Editor]
MAX+plus II File Edit View Symbol Assign Utilities Options Window Help
AND2
NOT
OR2
AND2

Figure B.9 A partially completed schematic for the circuit in Figure B.4.

named *input* into the schematic. Import two additional instances of the input symbol. To represent the output of the circuit, open the Primitives library and import the symbol named output. Arrange the symbols to appear as illustrated in Figure B.10.

Assigning Names to Input and Output Symbols

Point to the word PIN_NAME on the input pin symbol in the upper-left corner of the schematic and double-click the mouse. The pin name is selected, allowing a new pin name to be typed. Type $x1$ as the pin name. Hitting carriage return immediately after typing the pin name causes the mouse focus to move to the pin directly below the one currently being named. This method can be used to name any number of pins. Assign the names $x2$ and $x3$ to the middle and bottom input pins, respectively. Finally, assign the name f to the output pin.

Connecting Nodes with Wires

The next step is to draw lines (wires) to connect the symbols in the schematic together. Click on the icon that looks like an arrowhead along the left edge of the Manager window. This icon is called the Selection tool, and it allows the Graphic Editor to change automatically between the modes of selecting a symbol on the screen or drawing wires to interconnect symbols. The appropriate mode is chosen depending on where the mouse is pointing.

Move the mouse pointer on top of the $x1$ input symbol. The mouse pointer appears as an arrowhead when pointing anywhere on the symbol except at the right edge. The arrowhead means that the symbol will be selected if the mouse button is pressed. Move the mouse to point to the small line, called a *pinstub*, on the right edge of the $x1$ input symbol. The mouse pointer changes to a crosshair, which allows a wire to be drawn to connect the

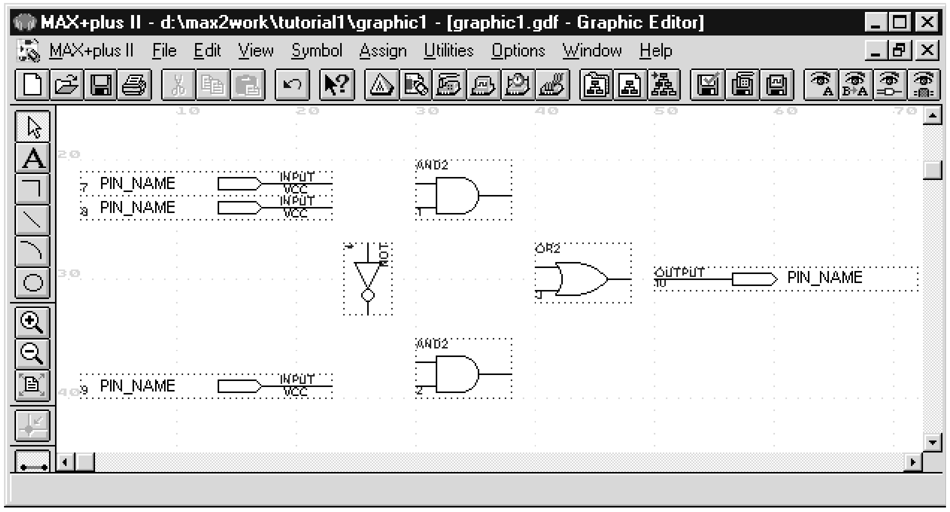


Figure B.10 Input and output symbols added to the schematic in Figure B.9.

pinstub to another location in the schematic. A connection between two or more pinstubs in a schematic is called a *node*. The name derives from electrical terminology, where the term *node* refers to any number of points in a circuit that are connected together by wires and thus have the same voltage.

Connect the input symbol for $x1$ to the AND gate at the top of the schematic as follows. While the mouse is pointing at the pinstub on the $x1$ symbol, click and hold the mouse button. Drag the mouse to the right until the line (wire) that is drawn reaches the pinstub on the top input of the AND gate; then release the button. The two pinstubs are now connected and represent a single node in the circuit.

Use the same procedure to draw a wire from the pinstub on the $x2$ input symbol to the other input on the AND gate. Then draw a wire from the pinstub on the input of the NOT gate upward until it reaches the wire connecting $x2$ to the AND gate. Release the mouse button and observe that a connecting dot is drawn automatically. The three pinstubs corresponding to the $x2$ input symbol, the AND-gate input, and the NOT-gate input now represent a single node in the circuit. Figure B.11 shows a magnified view of the part of the schematic that contains the connections drawn so far. To increase or decrease the portion of the schematic displayed on the screen, use the icons that look like magnifying glasses on the left side of the Manager window.

To complete the schematic, connect the output of the NOT gate to the lower AND gate and connect the input symbol for $x3$ to that AND gate as well. Connect the outputs of the two AND gates to the OR gate and connect the OR gate to the f output symbol. If any mistakes are made while connecting the symbols, erroneous wires can be selected with the mouse and then removed by pressing the Delete key or by selecting Edit | Delete. The finished schematic is depicted in Figure B.12. Save the schematic using File | Save.

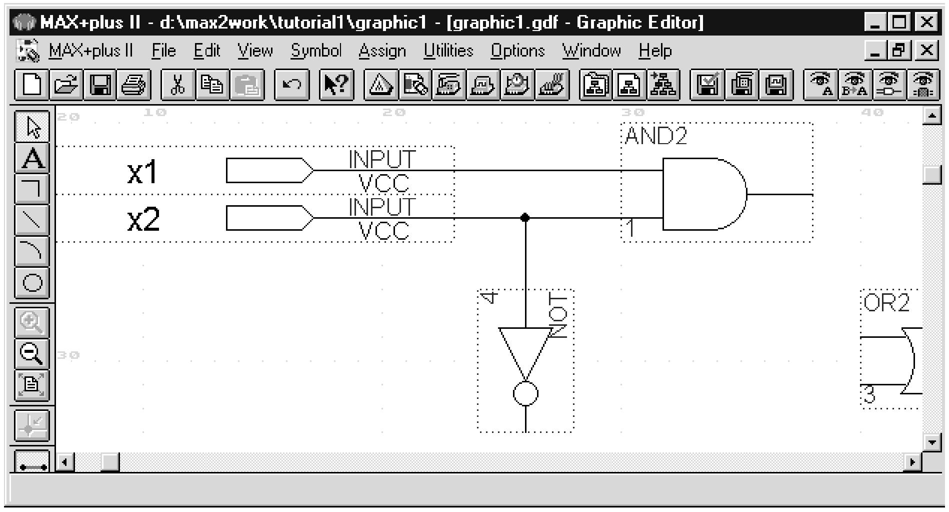

MAX+plus II - d:\max2work\tutorial1\graphic1 - [graphic1.gdf - Graphic Editor]
MAX+plus II File Edit View Symbol Assign Utilities Options Window Help
x1
x2
INPUT
VCC
INPUT
VCC
AND2
1
4
NOT
OR2
3

Figure B.11 Connecting the symbols in the schematic from Figure B.10.

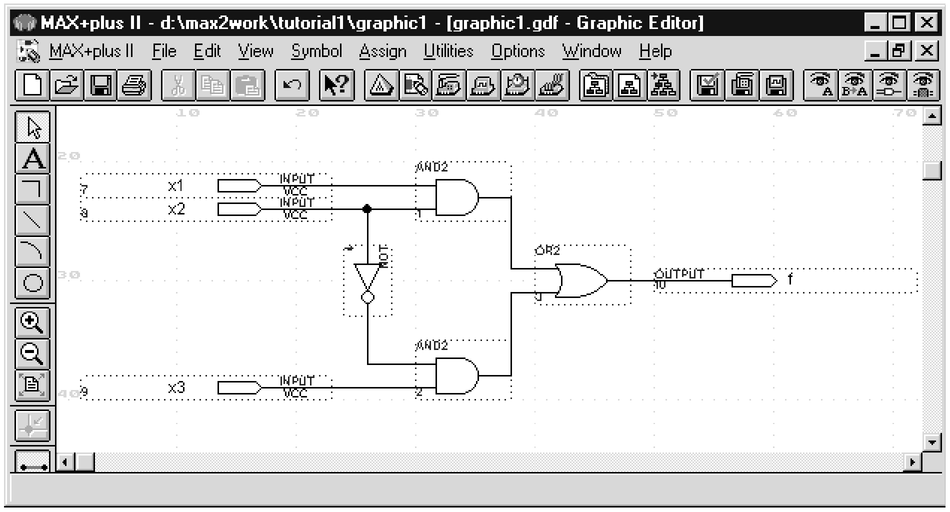

MAX+plus II - d:\max2work\tutorial1\graphic1 - [graphic1.gdf - Graphic Editor]
MAX+plus II File Edit View Symbol Assign Utilities Options Window Help
x1
x2
x3
INPUT
VCC
AND2
OR2
NOT
OUTPUT
f

Figure B.12 The completed schematic for the circuit in Figure B.4.

Since our example schematic is quite simple, it is easy to draw all the wires in the circuit without producing a messy diagram. However, in larger schematics some nodes that have to be connected may be far apart, in which case it is awkward to draw wires between them. In such cases the nodes are connected by assigning labels to them, instead of drawing wires. We will illustrate this method of connecting nodes in section D.3.1.

B.2.3 Synthesizing a Circuit from the Schematic

As we explained in section 2.8.2, after a schematic is entered into a CAD system, it is processed by initial synthesis tools. These tools analyze the schematic and generate a Boolean equation for each logic function in the circuit. In MAX+plusII the synthesis tools are controlled by the application program called the *Compiler*.

Using the Compiler

To open the Compiler window, click the mouse on the Compiler icon (it looks like a factory with a smoke stack) below the Manager window title bar or select MAX+plusII | Compiler.

For this tutorial we will use only the tools that are needed to allow us to perform a functional simulation of the schematic. To tell the Compiler to use these tools, select Processing | Functional SNF Extractor. The Compiler window should appear as shown in Figure B.13. The window shows three software modules that are invoked in sequence by the Compiler. The Compiler Netlist Extractor and Database Builder represent the initial synthesis tools. The module called Functional SNF Extractor creates a file, called a *simulator netlist file (SNF)*, which describes the functionality of the circuit and is used by the functional simulator.

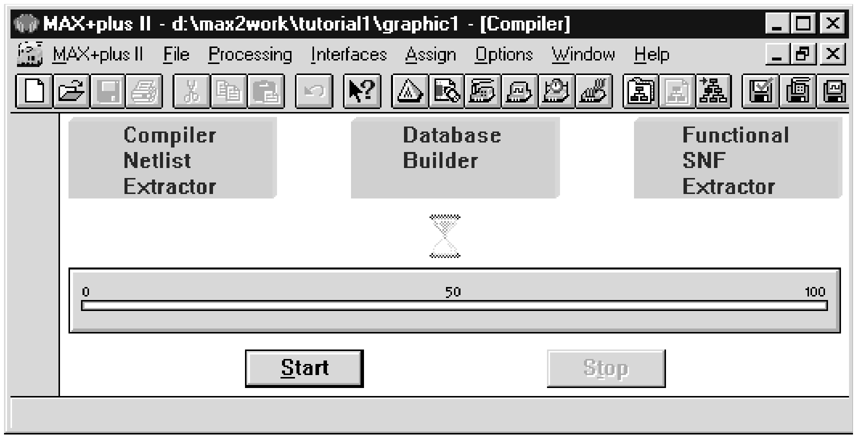


Figure B.13 The Compiler display.

Click the mouse on the Start button in the Compiler window. The Compiler indicates its progress by displaying a red progress bar and by placing an icon under each of the three software modules as they are executed. When the Compiler is finished, a window should be displayed that indicates zero warnings and zero errors. Click OK in this window to return to the Compiler window.

If the Compiler does not specify zero warnings and zero errors, then at least one mistake has been made when entering the schematic. In this case the Compiler opens a window called the Message Processor, which displays a message concerning each warning or error generated. An example showing how the Message Processor can be used to quickly locate and fix errors in a schematic is given in section B.2.5.

To close the Compiler window, use the *Close button* (it is an X) located in the top-right corner of its window.

B.2.4 Performing Functional Simulation

Before the schematic can be simulated, it is necessary to create the desired waveforms, called *test vectors*, to represent the input signals. For this tutorial we will use the MAX+plusII Waveform Editor to draw test vectors, but it is also possible to use a text editor to create test vectors in a plain text (ASCII) file. Documentation pertaining to ASCII test vectors can be opened by selecting Help | MAX+plusII Table of Contents. Click on Simulator, then click on Basic Tools, and finally click on Vector File (.vec).

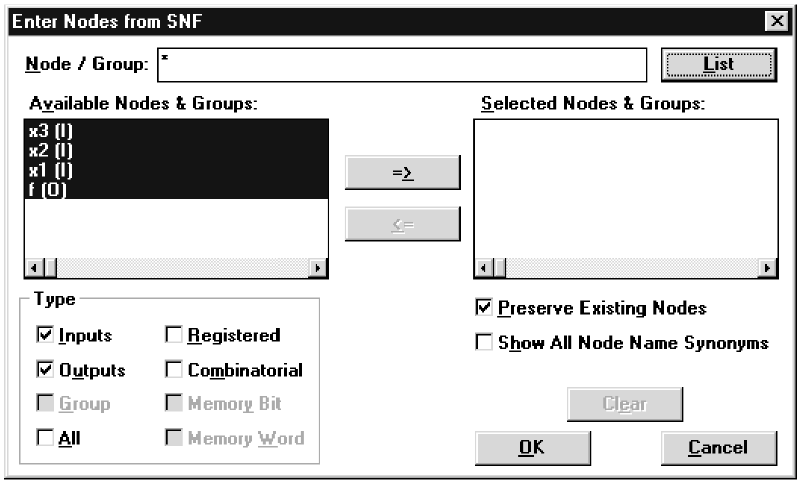
Using the Waveform Editor

Open the Waveform Editor window by selecting MAX+plusII | Waveform Editor. Because the Waveform Editor has many uses, it is necessary to indicate that we wish to enter test vectors for simulation purposes. Select File | Save As and type (if not already there) *graphic1.scf* in the box labeled File Name. A file with *scf* extension stores the waveforms that will be used as simulation test vectors.

Select Node | Enter Nodes from SNF to open the pop-up box shown in Figure B.14. Click on the List button in the upper-right corner of this box to display the names of the nodes in the current project in the box labeled Available Nodes & Groups. Click the mouse on the name $x3$ to highlight it. Click on the button labeled => to copy $x3$ into the box labeled Selected Nodes & Groups. Use the same procedure to select each of the other signals and copy them into the Selected Nodes & Groups box. It is also possible to select multiple nodes at the same time, by dragging the mouse upward or downward inside the Available Nodes & Groups box. Click OK to return to the Waveform Editor. The nodes $x1$, $x2$, $x3$, and f are now shown in the waveform display.

We will now specify the logic values to be used for the input signals during functional simulation. The logic values at the output f will be generated automatically by the simulator.

Select File | End Time to specify the total amount of time for which the circuit will be simulated. In the box labeled Time, type *160ns* to set the total simulation time to 160 nanoseconds. This amount of time is rather arbitrary because functional simulation does not include any timing delays, as discussed in section 2.8.3. The concept of *simulation time* will become more significant in Tutorial 2 when timing simulation is introduced. Click OK to return to the Waveform Editor. Select View | Fit in Window so that the entire time range from 0 to 160 ns is visible in the Waveform Editor display. In the Options menu make sure that Show Grid has a check mark next to it so that the Waveform Editor displays light vertical guidelines in the waveform area of the display. The guidelines provide a visual aid for positioning the mouse when drawing waveforms. Select Options | Grid Size and type *20ns* in the box labeled Grid Size. Click the mouse when pointing to any of the guidelines and observe that a vertical reference line is drawn at that point. We will use the reference line in Tutorial 2. Figure B.15 shows how the Waveform Editor window should look at this



Node / Group: *

Available Nodes & Groups:
x3 (I)
x2 (I)
x1 (I)
f (O)

Selected Nodes & Groups:

Type
☑ Inputs ☐ Registered
☑ Outputs ☐ Combinatorial
☐ Group ☐ Memory Bit
☐ All ☐ Memory Word

☑ Preserve Existing Nodes
☐ Show All Node Name Synonyms

Figure B.14 Selecting nodes for simulation.

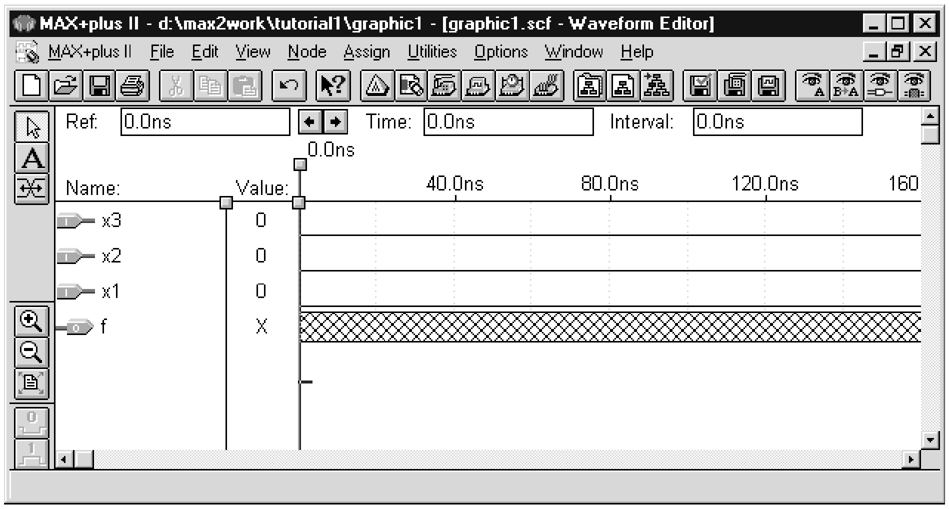

MAX+plus II - d:\max2work\tutorial1\graphic1 - [graphic1.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
Ref: 0.0ns
Time: 0.0ns
Interval: 0.0ns
0.0ns
Name:
Value:
40.0ns
80.0ns
120.0ns
160
x3 0
x2 0
x1 0
f X

Figure B.15 The Waveform Editor display.

point. The input waveforms are set to logic value 0, and the output is shown as a hashed-line pattern that indicates that the logic value has not yet been determined.

To thoroughly test the circuit during simulation, it is desirable to use as many different values of the input signals as possible. For our small example, there are only eight different valuations, and so it is easy to include all of them. To make all eight valuations fit in the 160 ns simulation time, the signal valuations have to change every 20 ns. To create the waveforms for the input signals, do the following.

Activate the *Waveform Editing* tool by pressing its icon on the left edge of the window. The icon is shown in the top-left corner of Figure B.16; it looks like two arrows pointing left and right. Position the mouse pointer over the waveform for input *x*3 at the 20 ns grid line. Press and drag the mouse to the right to highlight the section of the *x*3 waveform from 20 ns to 40 ns, as illustrated in Figure B.16. The Waveform Editing Tool automatically

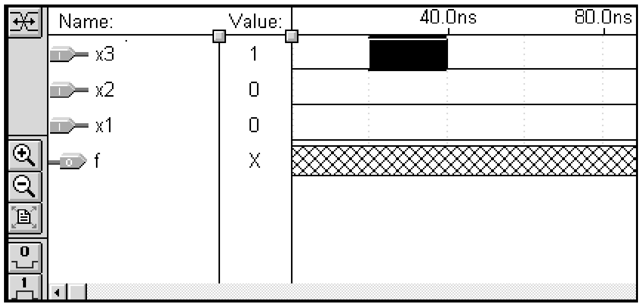


Figure B.16 Editing the waveform for *x*3 from Figure B.15.

changes the selected portion of the waveform from its present value 0 to the value 1. Next select the section of the waveform for *x*3 between 60 ns and 80 ns to set it to 1. Continue in this manner to set every second 20 ns section of *x*3 to 1.

An alternative way to draw waveforms is to use the Selection tool, which is activated by selecting the icon that looks like an arrowhead along the left edge of the window. Using the Selection tool, the procedure for drawing a waveform is to first select a section of the waveform by dragging the mouse over it. The highlighted section can be set to 1 by selecting Edit | Overwrite | High. The highlighted section can also be changed by using the buttons labeled 0 or 1 along the left edge of the window.

Use the Waveform Editing tool to set the waveform for *x*2 to 1 in the range from 40 ns to 80 ns, as well as from 120 ns to 160 ns. Also, set the waveform for *x*1 to 1 in the range from 80 ns to 160 ns. The waveforms drawn, as illustrated in Figure B.17, now include all eight input valuations. Select File | Save to save the waveforms in the *graphic1.scf* file.

Performing the Simulation

To open the Simulator window, shown in Figure B.18, click on its icon (it looks like a computer with a waveform on the screen) or Select MAX+plusII | Simulator. MAX+plusII provides both functional simulation and timing simulation. The type of simulation used by the Simulator application is determined automatically by the settings used in the Compiler application. The Simulator will perform a functional simulation in this case because we instructed the Compiler to generate information for functional simulation, as discussed for Figure B.13.

Observe in Figure B.18 that the Simulator specifies that it will use the file called *graphic1.scf* as the simulator input and will perform the simulation for the time range from 0 to 160 ns. Click the Start button to perform the simulation. The Simulator displays a message indicating that no errors were generated. Click OK to return to the Simulator window. The simulator stores the results of the simulation in the *graphic1.scf* file. To view the file, click on the Open SCF button in the simulator window, which automatically opens the Waveform Editor window and displays the file. As illustrated in Figure B.19, the Simulator creates a waveform for the output *f*. The reader should verify that the generated waveform corresponds to the truth table for *f* given in Figure B.4*b*. The Waveform Editor and Simulator windows can now be closed.

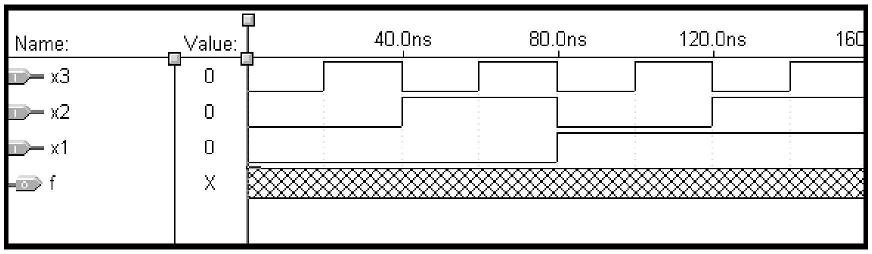
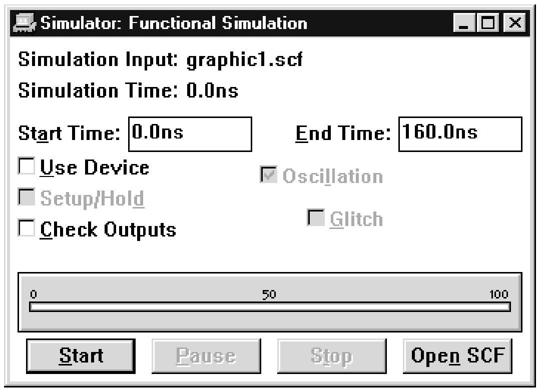


Figure B.17 The completed waveforms for *x*1, *x*2, and *x*3.

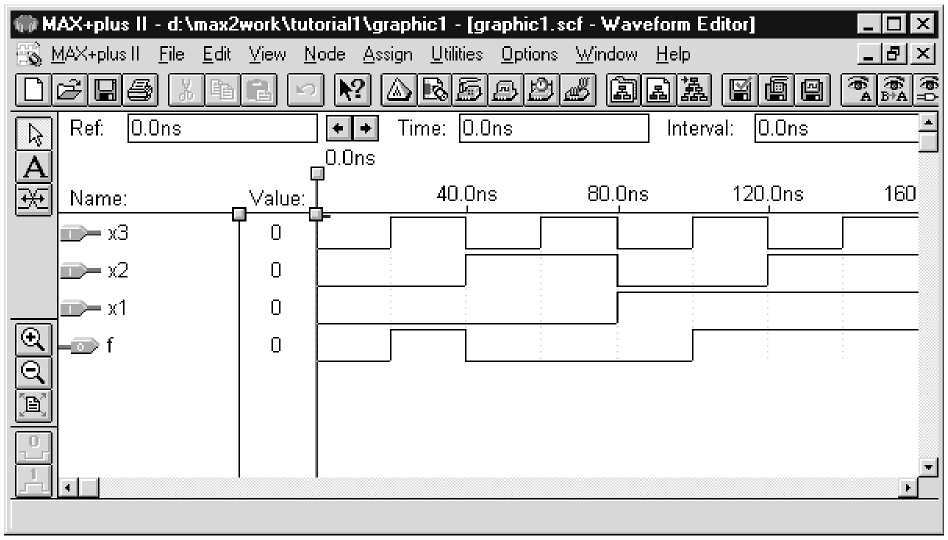


Simulation Input: graphic1.scf
Simulation Time: 0.0ns
Start Time: 0.0ns
End Time: 160.0ns

Figure B.18 The Simulator display.

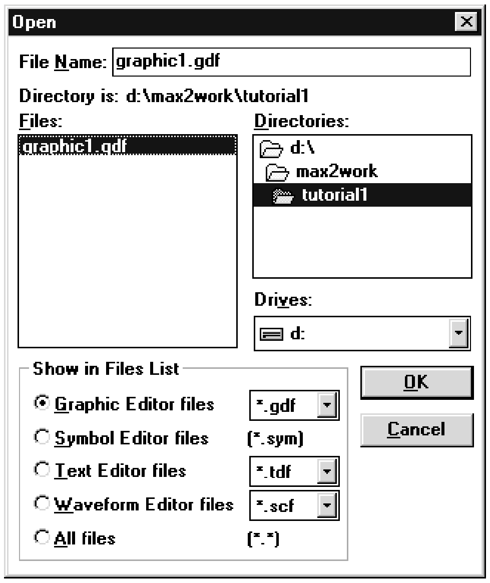
B.2.5 Using the Message Processor to Locate and Fix Errors

In the description in section B.2.3 of how the Compiler is used to synthesize a circuit from the schematic, we said that the Compiler should produce a message stating that no warnings or errors were generated. In this section we illustrate what happens when there is an error in the schematic. To insert an error in the schematic created for f, reopen the schematic by selecting File | Open to open the pop-up box shown in Figure B.20. In the box labeled Show in Files List, click on Graphic Editor Files. Then in the box labeled Files, click on the name



Ref: 0.0ns
Time: 0.0ns
Interval: 0.0ns

Figure B.19 Functional simulation results for the waveforms in Figure B.17.



Open

File Name: graphic1.gdf

Directory is: d:\max2work\tutorial1

Files: graphic1.gdf

Directories: d:\ max2work tutorial1

Drives: d:

Show in Files List

- [✓ Graphic Editor files] *.gdf
- Symbol Editor files (*.sym)
- Text Editor files *.tdf
- Waveform Editor files *.scf
- All files (*.*)

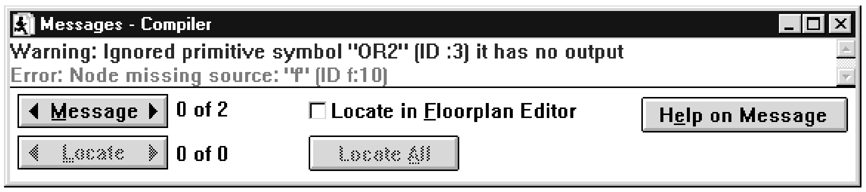
OK Cancel

Figure B.20 The dialog box used to reopen the schematic.

graphic1.gdf to put this name in the box labeled File Name. Alternatively, *graphic1.gdf* can be typed into the box rather than using the mouse to select it from the list of files. Click OK to open the file inside the Graphic Editor.

Use the mouse to select the wire that connects the output of the OR gate to the *f* output symbol. Delete the wire by pressing the Delete key; then save the schematic file. Open the Compiler window and run the synthesis tools again. The Compiler should produce a message stating that one warning and one error were found. Click OK. A window, called the Message Processor, is automatically opened to display the messages generated by the Compiler, as illustrated in Figure B.21. If the Message Processor window is obscured by some other window, select MAX+plusII | Message Processor to bring the Message Processor window to the foreground.

The warning message is produced because the OR-gate output is not connected to any other node in the schematic. The error message states that the *f* output symbol is



Messages - Compiler

```
Warning: Ignored primitive symbol "OR2" (ID :3) it has no output
Error: Node missing source: "f" (ID f:10)
```

Message 0 of 2 Locate in Floorplan Editor Help on Message

Locate 0 of 0 Locate All

Figure B.21 The Message Processor display.

not connected to anything. Although it is clear how to fix the error, since we created it purposely, in general some of the messages displayed by the Compiler when synthesizing larger circuits may not be obvious. In such cases it is possible to select a message with the mouse and then click on the Help on Message button in the Message Processor window; documentation that explains the message is automatically opened. Experiment with this feature for both the warning and error messages in Figure B.21.

Another convenient feature of the Message Processor is the Locate button in the lower-left corner of the window. It can be used to automatically display the section of the schematic where the error exists. Select the warning message and then click the Locate button. Observe that the Graphic Editor is automatically displayed with the OR gate highlighted. Next select the error message in the Message Processor window and then click the Locate button again. The *f* output symbol becomes highlighted in the Graphic Editor.

Use the Graphic Editor to redraw the missing wire between the OR-gate output and the *f* output symbol. Save the schematic and then use the Compiler to run the synthesis tools to see that the error is fixed. We have now completed our introduction to design using schematic capture. If any application windows are still open, close them to return to the Manager window.

B.3 Design Entry Using VHDL

This section illustrates the process of using MAX+plusII to implement logic functions by writing VHDL code. We will implement the function *f* from section B.2, where we used schematic capture. After typing the VHDL code, it will be simulated with the Functional Simulator.

B.3.1 Specifying the Project Name

We need a new project name for the VHDL design. In the Manager window select File | Project | Name. We will store the design files for the project in the same directory that we used for the schematic capture design created earlier. In the box labeled Project Name, type *example1* as the name for the project and then click OK. The name of the project is displayed in the title bar of the Manager window.

B.3.2 Using the Text Editor

MAX+plusII provides a text editor that can be used for typing VHDL code. Open the Text Editor window by selecting MAX+plusII | Text Editor. The first step is to specify a name for the file that will be created. Select File | Save As to open the pop-up box depicted in Figure B.22. Type *example1.vhd* in the box labeled File Name. You must use exactly this name. The name *example1* must match the name of the project, and the filename extension *vhd* must be used for all files that contain VHDL code. When File | Save As is selected, the Text Editor places the default name *example1.tdf* in the File Name box. The *tdf* extension stands

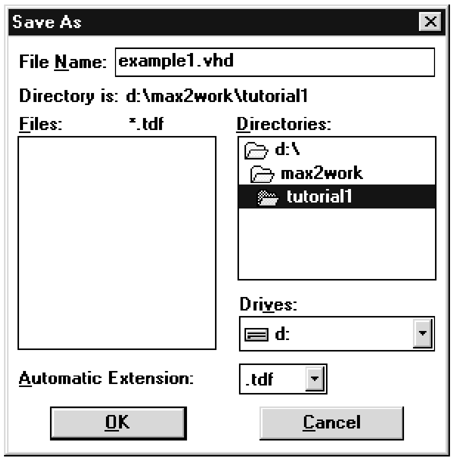
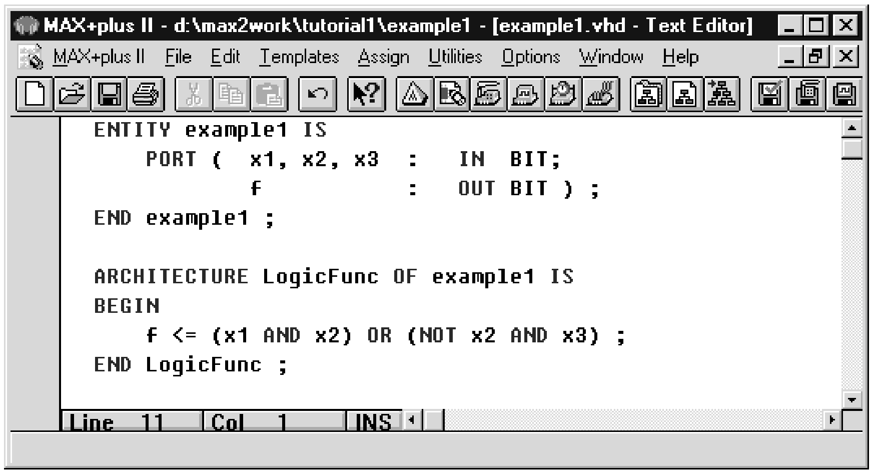


Figure B.22 Specifying a name for the VHDL design file.

for text design file. It is used for files that contain source code written in the Altera Hardware Description Language (AHDL), which is another language supported by the MAX+plusII system. Make sure to change the filename extension from *tdf* to *vhd*. We should mention that it is not necessary to use the Text Editor provided in MAX+plusII. Any text editor can be used to create the file named *example1.vhd*, as long as the text editor can generate a plain text (ASCII) file.

The VHDL code for this example is shown in Figure 2.29. Type the code into the Text Editor to obtain the display in Figure B.23. Most of the commands available in the Text Editor are self-explanatory. Text is entered at the *insertion point*, which is indicated by a thin vertical line. The insertion point can be moved either by using the keyboard arrow keys


MAX+plus II - d:\max2work\tutorial1\example1 - [example1.vhd - Text Editor]
MAX+plus II File Edit Templates Assign Utilities Options Window Help
Line 11 Col 1 INS

```
ENTITY example1 IS
    PORT (  x1, x2, x3  :   IN  BIT;
            f           :   OUT BIT ) ;
END example1 ;

ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Figure B.23 The Text Editor display showing the VHDL code from Figure 2.29.

or by using the mouse. Two features of the Text Editor are especially convenient for typing VHDL code. First, the editor can optionally display different types of VHDL statements in different colors. To turn on this option, open the Options menu and place a check mark next to the item named Syntax Coloring. Second, the editor can automatically indent the text on a new line so that it matches the previous line. To turn on this option, place a check mark beside Options | Auto-indent. Save the file.

Using VHDL Templates

The syntax of VHDL code is sometimes difficult for a designer to remember. To help with this issue, the Text Editor provides a collection of *VHDL templates*. The templates provide examples of various types of VHDL statements, such as an entity declaration, an architecture, and a signal assignment statement. It is worthwhile to browse through the templates by selecting Templates | VHDL Template to become familiar with this resource.

B.3.3 SYNTHESIZING A CIRCUIT FROM THE VHDL CODE

In section 2.8.2 we said that a VHDL compiler generates a logic circuit from VHDL code. The VHDL compiler provided by MAX+plusII is controlled by the Compiler application.

Using the Compiler

Open the Compiler window. As described for the design created with schematic capture earlier, select Processing | Functional SNF Extractor so that the Compiler will generate the information needed to perform functional simulation. Press the Start button in the Compiler window. If the VHDL code has been typed correctly, the Compiler will display a message that says that no errors or warnings were generated.

If the Compiler does not specify zero warnings and zero errors, then at least one mistake was made when typing the VHDL code. In this case the Message Processor window is opened, and it displays a message corresponding to each warning or error found. An example showing how the Message Processor can be used to quickly locate and fix errors in VHDL code is given in section B.3.5. The Compiler window can now be closed.

B.3.4 PERFORMING FUNCTIONAL SIMULATION

Functional simulation of the VHDL code is done in exactly the same way as the simulation described earlier for the design created with schematic capture. Open the Waveform Editor and select File | Save As to save the file with the name *example1.scf*. Following the procedure given in section B.2.4, select Node | Enter Nodes from SNF and import the nodes in the project into the Waveform Editor. Draw the waveforms for inputs $x1$, $x2$, and $x3$ shown in Figure B.17. It is also possible to open the previously drawn waveform file *graphic1.scf* and then "copy and paste" the waveforms for $x1$, $x2$, and $x3$. The procedure for copying waveforms is described in Help | MAX+plusII Table of Contents | Waveform Editor | Procedures | Copying, Cutting & Pasting Nodes and Groups. Open the Simulator and click on the Start button. The waveform generated by the Simulator for the output f should be the same as the waveform in Figure B.19.

B.3.5 Using the Message Processor to Debug VHDL Code

In section B.2.5 we showed that the Message Processor application can be used to quickly locate and fix errors in a schematic. A similar procedure is available for finding errors in VHDL code. To illustrate this, open the *example1.vhd* file with the Text Editor. In the fourth line, which reads "END example1 ;" delete the semicolon at the end of the statement. Save the *example1.vhd* file and then run the Compiler again. The Compiler generates one error, and the Message Processor window is opened, as illustrated in Figure B.24. The error message specifies that the problem was identified when processing line 6 in the VHDL source code file. Select the error message in the Message Processor window and then click the Locate button. The Text Editor window is automatically displayed with the insertion point at line 6.

Fix the error by reinserting the missing semicolon; then save the file and run the synthesis tools again, to confirm that the error is fixed. We have now completed the introduction to design using VHDL code. Close any open application windows to return to the Manager window.

B.4 Design Entry Using Truth Tables

This section describes the process of designing a logic circuit using a truth table. We will implement the truth table shown in Figure B.25. It will be entered into the CAD system by drawing a timing diagram with the Waveform Editor. We discuss the equivalence of truth tables and timing diagrams in section 2.4.1.

We need to specify a new project name for the truth table design. Using File | Project | Name, follow the procedure described in section B.3.1 to assign the name *timing1* to the project. Use the same directory as for the projects designed in the previous sections.

B.4.1 Using the Waveform Editor

Open the Waveform Editor window by selecting MAX+plusII Waveform Editor. The Waveform Editor can be used for multiple purposes. In section B.2.4 the editor was used to create

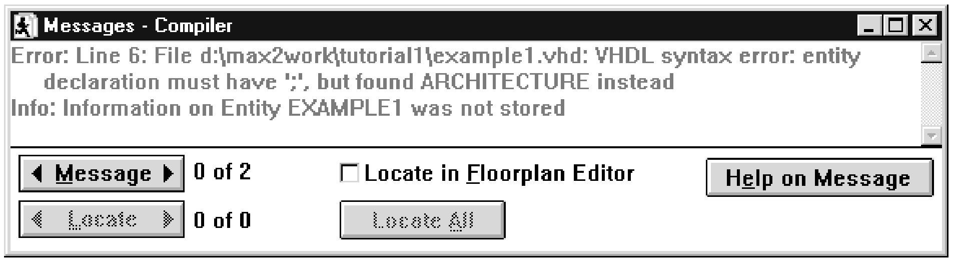

Error: Line 6: File d:\max2work\tutorial1\example1.vhd: VHDL syntax error: entity
declaration must have ';', but found ARCHITECTURE instead
Info: Information on Entity EXAMPLE1 was not stored

Figure B.24 The Message Processor window displaying an error in VHDL code.

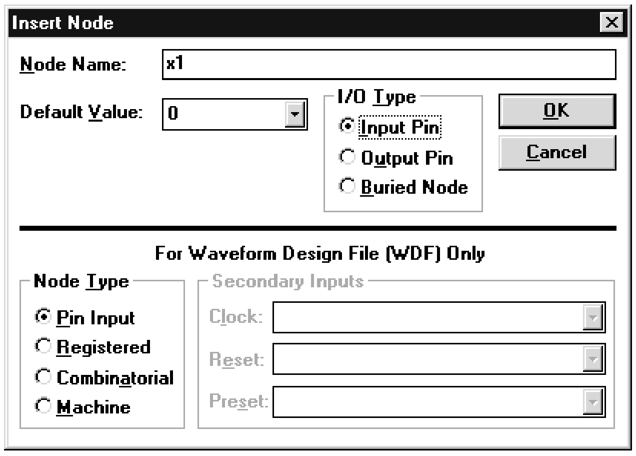
x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure B.25 A three-variable function.

input files for simulation. In this section the Waveform Editor will be used to create a different type of file, called a *waveform design file*. To specify the type of file to be created, select File | Save As. In the box labeled File Name, type the name *timing1.wdf*. You must use exactly this name. The name *timing1* must match the name of the project, and the filename extension *wdf* indicates that the waveforms will be used to describe a logic function, instead of being used as simulation input.

B.4.2 Creating the Timing Diagram

To create a timing diagram, it is first necessary to specify the input and output signals for the circuit. Select Node | Insert Node to open the pop-up box shown in Figure B.26. In the box labeled Node Name in Figure B.26, type $x1$. Since $x1$ is an input to the circuit, make


Insert Node

Node Name: x1

Default Value: 0

I/O Type: [✓ Input Pin] Output Pin Buried Node

For Waveform Design File (WDF) Only

Node Type: [✓ Pin Input] Registered Combinatorial Machine

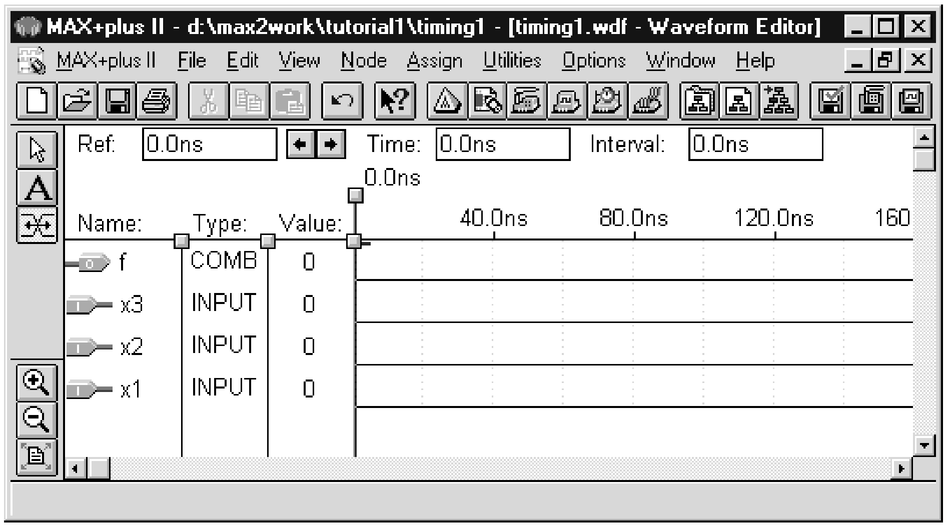
Secondary Inputs: Clock: Reset: Preset:

Figure B.26 Inserting a node into the Waveform Editor.

sure that Input Pin is selected in the box labeled I/O Type. Click OK. The input $x1$ appears in the Waveform Editor display. Use the same procedure to insert inputs $x2$ and $x3$ into the Waveform Editor display. Next, select Node | Insert Node again and type f in the Node Name box. Since f is the output for the circuit, make sure that Output Pin is selected in the box labeled I/O Type and then click OK. An alternative way to open the Insert Node pop-up box used above is to double-click in the Waveform Editor display in a blank space under the column labeled Name. The inserted node will be placed in the Waveform Editor window at the location where the mouse was double-clicked.

Having inserted the waveforms into the Waveform Editor, we will now draw a timing diagram to represent the truth table in Figure B.25. Since the truth table has eight rows, we will need to draw eight valuations of the inputs $x1$, $x2$, and $x3$. In section B.2.4 we set the size of the grid displayed in the Waveform Editor to 20 ns. If this same grid size is used, then the total time range needed in the Waveform Display is 160 ns. Select File | End Time and specify *160ns* as the total simulation time. To make the entire time range visible in the waveform display, select View | Fit in Window or type the shortcut command Ctrl+w (while holding down the Ctrl key, press the w key). The Waveform Editor window should now appear as shown in Figure B.27.

Following the procedure described in section B.2.4, modify the waveform for signal $x3$ so that it is 1 for every second 20 ns time range. Also, edit the waveform for $x2$ so that it is 1 for the time ranges from 40 ns to 80 ns and from 120 ns to 160 ns. Finally, set the waveform for $x1$ to 1 in the time range from 80 ns to 160 ns. Previously, when using the Waveform Editor, we did not specify a waveform for the output of the circuit, because the output waveform was generated by the simulator. However, in this case we need to specify a waveform for output f that corresponds to its truth table. In Figure B.25 the function is


MAX+plus II - d:\max2work\tutorial1\timing1 - [timing1.wdf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
Ref: 0.0ns
Time: 0.0ns
Interval: 0.0ns
0.0ns
40.0ns
80.0ns
120.0ns
160

Name:	Type:	Value:
f	COMB	0
x3	INPUT	0
x2	INPUT	0
x1	INPUT	0

Figure B.27 The Waveform Editor display for the truth-table design.

1 in the rows where $x1$, $x2$, and $x3$ have the valuations 001, 011, 101, 110, and 111. Use the Graphic Editor to change the waveform for f to 1 for the appropriate time ranges. For instance, f should be set to 1 in the time range from 20 ns to 40 ns because this represents the input valuation 001. After completing the waveform for f, the waveform display should appear as shown in Figure B.28. Notice that we have rearranged the waveforms, by moving f to the bottom, in comparison to Figure B.27. Waveforms can be moved by pointing the mouse at the small symbol, called the *node handle*, to the left of the signal name in the waveform display and then dragging the waveform upward or downward. Select File | Save to save the timing diagram in the *timing1.wdf* file.

B.4.3 Synthesizing a Circuit from the Waveforms

The next step is to use the MAX+plusII Compiler to perform the initial synthesis steps for the circuit. The Compiler will generate a Boolean expression to represent f, according to the truth table given by the timing diagram.

Use the same procedure described for the designs created with schematic capture and VHDL code. Open the Compiler window and select Processing | Functional SNF Extractor. Press the Start button in the Compiler window and then click OK in response to the Compiler message that says that no warnings or errors were found.

For the circuits designed in the previous sections, after logic synthesis was completed, the next step performed was functional simulation. It does not make sense to perform the functional simulation for the circuit designed in this section, because the waveforms that would be used as inputs to the simulator would be the same waveforms used to design the

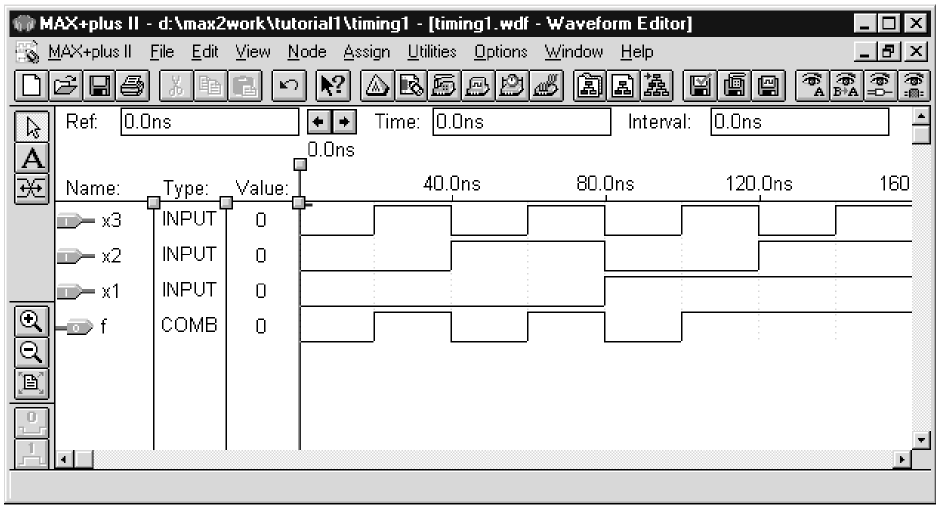


Figure B.28 The timing diagram representing the truth table in Figure B.25.

circuit! In the next section we will use the circuit synthesized from the timing diagram in this section as part of a larger circuit, and we will simulate the operation of the larger circuit.

The tutorial on design with truth tables is now complete, so close any open application windows to return to the Manager window.

B.5 Mixing Design-Entry Methods

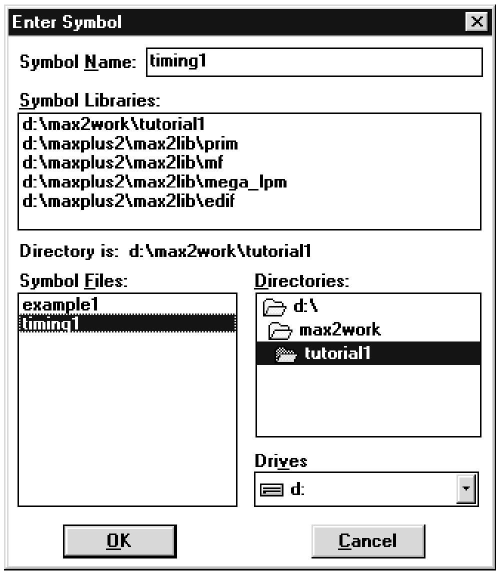
It is possible to design a logic circuit using a mixture of design-entry methods. As an example, in this section we will create a schematic that includes the circuit designed using the truth table in the previous section.

We need to specify a new project name for the mixed design. Select File | Project | Name and assign the name *mixed1* to the project. Use the same directory as for the projects designed in the previous sections.

B.5.1 Creating a Schematic That Includes a Truth Table

Open the Graphic Editor by selecting MAX+plusII | Graphic Editor. Select File | Save As and, if not already there, type the name *mixed1.gdf* in the File Name box. Make sure to use exactly this name.

Double-click the blank space in the Graphic Editor to open the Enter Symbol pop-up box, as shown in Figure B.29. In the box labeled Symbol Name, type the name *timing1*, which is the name of the circuit designed using a truth table in the previous section. Click OK to import a graphical symbol for the *timing1* circuit into the Graphic Editor. Once



Symbol Name: timing1

Symbol Libraries:
d:\max2work\tutorial1
d:\maxplus2\max2lib\prim
d:\maxplus2\max2lib\mf
d:\maxplus2\max2lib\mega_lpm
d:\maxplus2\max2lib\edif

Directory is: d:\max2work\tutorial1

Symbol Files:
example1
timing1

Directories:
d:\
max2work
tutorial1

Drives: d:

Figure B.29 Importing the truth-table design into the Graphic Editor.

the *timing1* symbol is imported into the Graphic Editor, double-clicking on the symbol automatically opens the Waveform Editor and displays the waveforms that were used to design the circuit. When the Waveform Editor is closed, the Graphic Editor is automatically reopened. This ability to move quickly from one design-entry tool to another is convenient when it is necessary to make changes to a schematic or the subcircuits in it.

Following the procedure described in section B.2.2, import a two-input AND-gate symbol and a NOT gate from the Primitives library into the Graphic Editor. Also from the Primitives library, import three input symbols and an output symbol. Arrange the symbols in the schematic as illustrated in Figure B.30. As described in section B.2.2, assign the names $x1$, $x2$, and $x3$ to the input symbols and assign the name f to the output symbol. The reader will observe that the name $x3$ is used twice in this design project: as an input to the *timing1* subcircuit and as an input to the mixed schematic. The MAX+plusII compiler treats these two nodes named $x3$ as separate nodes because they appear in different levels of the design project hierarchy. Connect the symbols in the schematic together as shown in Figure B.31. Because a wire drawn with the Graphic Editor can be either straight or have a single bend, it is necessary to draw more than one wire for the connection shown in the figure from the AND-gate output to the input labeled $x3$ on the *timing1* subcircuit. Start drawing each wire so that it touches the end of the previously drawn wire; wires that touch are automatically connected by the Graphic Editor. Save the schematic.

B.5.2 SYNTHESIZING AND SIMULATING A CIRCUIT FROM THE SCHEMATIC

Use the procedure described for the designs created in the previous sections to synthesize a circuit from the schematic. The synthesis tools will create a single logic circuit by merging the *timing1* subcircuit with the other logic gates in the schematic. Open the Compiler window, select Processing | Functional SNF Extractor, and then run the Compiler.

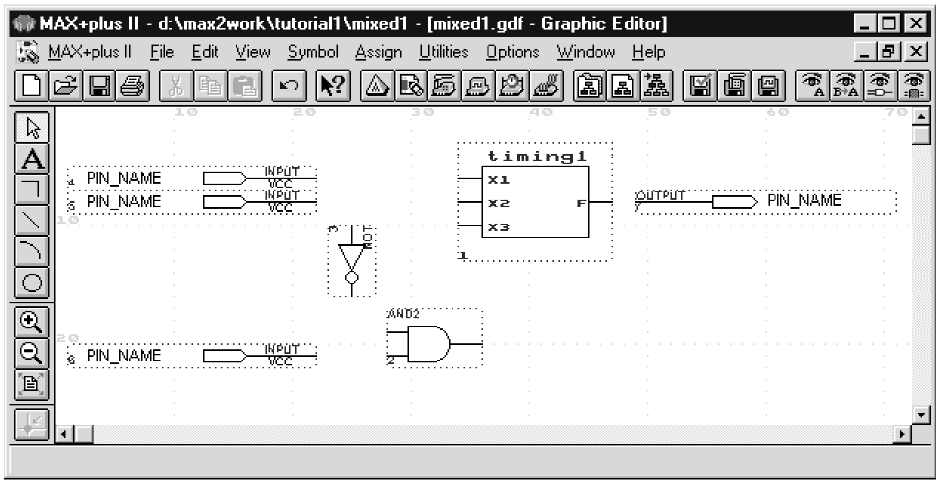

MAX+plus II - d:\max2work\tutorial1\mixed1 - [mixed1.gdf - Graphic Editor]
MAX+plus II File Edit View Symbol Assign Utilities Options Window Help
PIN_NAME
PIN_NAME
PIN_NAME
INPUT VCC
INPUT VCC
INPUT VCC
timing1
x1
x2
x3
F
OUTPUT
PIN_NAME
NOT
AND2

Figure B.30 A schematic including a truth table and logic gates.

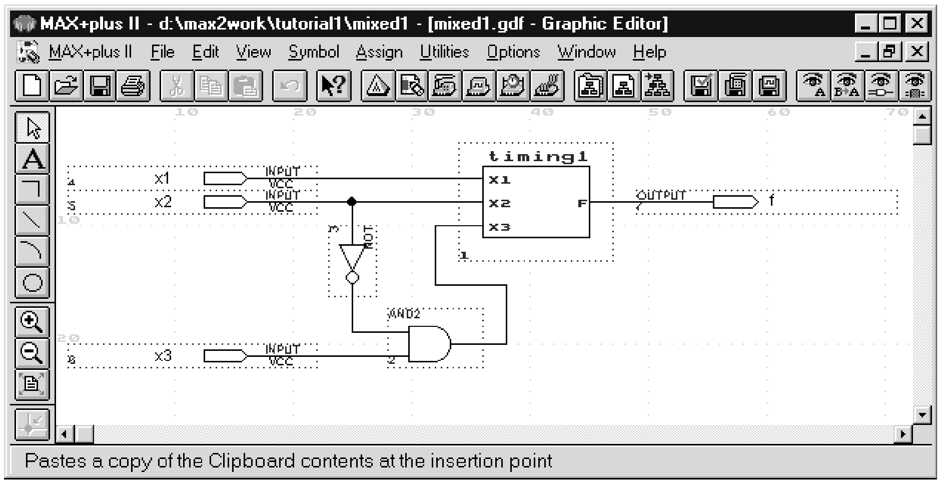

MAX+plus II - d:\max2work\tutorial1\mixed1 - [mixed1.gdf - Graphic Editor]
MAX+plus II File Edit View Symbol Assign Utilities Options Window Help
timing1
x1
x2
x3
F
INPUT
VCC
OUTPUT
NOT
AND2
f
Pastes a copy of the Clipboard contents at the insertion point

Figure B.31 The completed schematic corresponding to Figure B.30.

Simulation of the *mixed1* project is done in exactly the same way as for the other projects created in this tutorial. Open the Waveform Editor and select File | Save As to create a new file named *mixed1.scf*. Following the procedure given in section B.2.2, import the input and output nodes $x1$, $x2$, $x3$, and f into the Waveform Editor. Draw the waveforms for inputs $x1$, $x2$, and $x3$ that are shown in Figure B.17. Open the Simulator and click on the Start button; then select Open SCF to see the results of the simulation. The waveform generated by the Simulator for the output f should be exactly the same as the waveform shown in Figure B.19. The *mixed1* schematic represents the logic function $f = x_1x_2 + \overline{x}_2x_3$ that was designed using both schematic capture and VHDL code in this tutorial. Techniques that can be used to synthesize the expression for f from the *mixed1* schematic are covered in Chapter 4.

In practice a designer would not use a mixture of design-entry methods for a circuit as simple as our example. The reason that we have created the *mixed1* schematic is simply to illustrate that MAX+plusII allows design-entry methods to be combined in a hierarchical manner. It is also possible, although not shown here, to create a schematic that includes a subcircuit designed using VHDL code. MAX+plusII provides a convenient feature, called the Hierarchy Display, for working with hierarchical design projects.

B.5.3 USING THE HIERARCHY DISPLAY

Select MAX+PlusII | Hierarchy Display to open the Hierarchy Display window shown in Figure B.32. The display shows that the design project consists of two hierarchical levels, with *mixed1* at the higher level and *timing1* at the lower level. The *mixed1* design project has an icon next to it, labeled *gdf*. It can be double-clicked to automatically open the *mixed1.gdf* file in the Graphic Editor. Similarly, *timing1* has an icon next to it, labeled *wdf*. If this icon

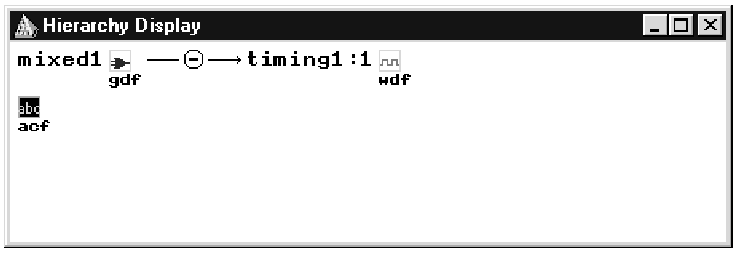


Figure B.32 The Hierarchy Display window for the *mixed1* design project.

is double-clicked, the file *timing1.wdf* is opened in the Waveform Editor. Experiment with this method of opening design files. Figure B.32 also shows a small icon labeled *acf*, which represents the *assignment & configuration file* for the project. The file contains settings for a large number of optional features of MAX+plusII that affect the way the design files are processed. These settings are saved automatically in the assignment & configuration file, and so we will not need to modify them manually. Although it is not necessary, the *acf* file can be opened in the Text Editor by double-clicking on its icon in the Hierarchy Display.

B.5.4 Concluding Remarks

This tutorial has introduced the basic use of the MAX+plusII CAD system. We have shown how to perform design entry by drawing a schematic, writing VHDL code, and drawing a timing diagram that represents a truth table. Each design was processed by the initial synthesis tools and then simulated with the functional simulator.

In the next tutorial we will show how the logic synthesis and physical design tools are used to implement circuits in PLDs. The timing characteristics of the implemented circuits will be examined using timing simulation.

chapter

6

Combinational-Circuit Building Blocks

Chapter Objectives

In this chapter you will learn about:

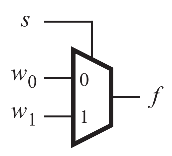
- Commonly used combinational subcircuits
- Multiplexers, which can be used for selection of signals and for implementation of general logic functions
- Circuits used for encoding, decoding, and code-conversion purposes
- Key Verilog constructs used to define combinational circuits

Previous chapters have introduced the basic techniques for design of logic circuits. In practice, a few types of logic circuits are often used as building blocks in larger designs. This chapter discusses a number of these blocks and gives examples of their use. The chapter also includes a major section on Verilog, which describes several key features of the language.

6.1 Multiplexers

Multiplexers were introduced briefly in Chapters 2 and 3. A multiplexer circuit has a number of data inputs, one or more select inputs, and one output. It passes the signal value on one of the data inputs to the output. The data input is selected by the values of the select inputs. Figure 6.1 shows a 2-to-1 multiplexer. Part (*a*) gives the symbol commonly used. The *select* input, s, chooses as the output of the multiplexer either input w_0 or w_1. The multiplexer's functionality can be described in the form of a truth table as shown in part (*b*) of the figure. Part (*c*) gives a sum-of-products implementation of the 2-to-1 multiplexer, and part (*d*) illustrates how it can be constructed with transmission gates.

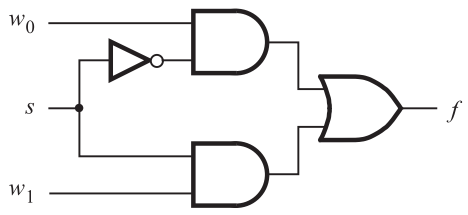
Figure 6.2*a* depicts a larger multiplexer with four data inputs, $w_0, \ldots, w_3$, and two select inputs, s_1 and s_0. As shown in the truth table in part (*b*) of the figure, the two-bit number represented by $s_1 s_0$ selects one of the data inputs as the output of the multiplexer. A sum-of-products implementation of the 4-to-1 multiplexer appears in Figure 6.2*c*. It



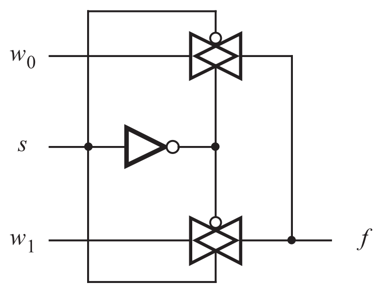
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

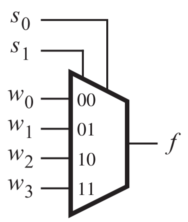


(c) Sum-of-products circuit



(d) Circuit with transmission gates

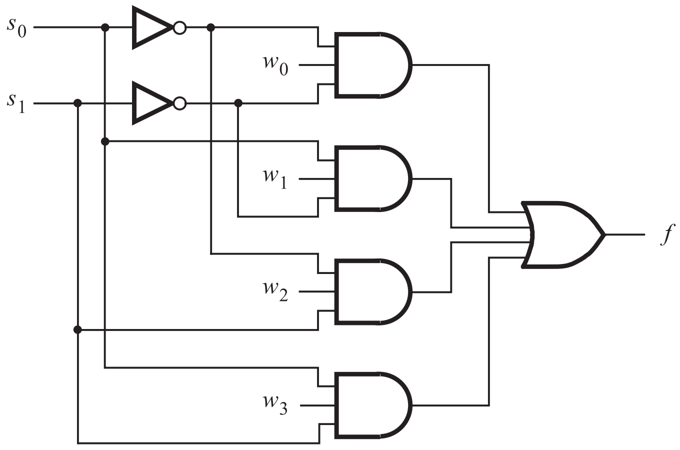
Figure 6.1 A 2-to-1 multiplexer.



(a) Graphical symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table



(c) Circuit

Figure 6.2 A 4-to-1 multiplexer.

realizes the multiplexer function

$$f = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1 + s_1\bar{s}_0w_2 + s_1s_0w_3$$

It is possible to build larger multiplexers using the same approach. Usually, the number of data inputs, n, is an integer power of two. A multiplexer that has n data inputs, $w_0, \ldots, w_{n-1}$, requires $\lceil \log_2 n \rceil$ select inputs. Larger multiplexers can also be constructed from smaller multiplexers. For example, the 4-to-1 multiplexer can be built using three 2-to-1 multiplexers as illustrated in Figure 6.3. If the 4-to-1 multiplexer is implemented using transmission gates, then the structure in this figure is always used. Figure 6.4 shows how a 16-to-1 multiplexer is constructed with five 4-to-1 multiplexers.

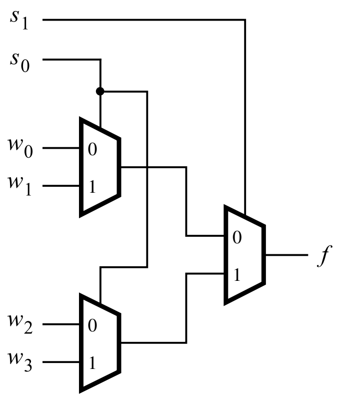


Figure 6.3 Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

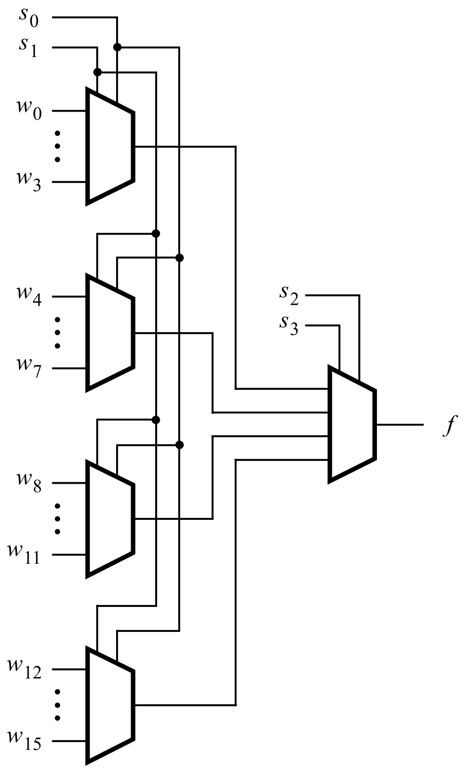


Figure 6.4 A 16-to-1 multiplexer.

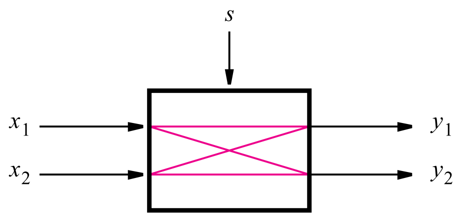
Example 6.1

Figure 6.5 shows a circuit that has two inputs, x_1 and x_2, and two outputs, y_1 and y_2. As indicated by the blue lines, the function of the circuit is to allow either of its inputs to be connected to either of its outputs, under the control of another input, s. A circuit that has n inputs and k outputs, whose sole function is to provide a capability to connect any input to any output, is usually referred to as an $n \times k$ crossbar switch. Crossbars of various sizes can be created, with different numbers of inputs and outputs. When there are two inputs and two outputs, it is called a 2×2 crossbar.

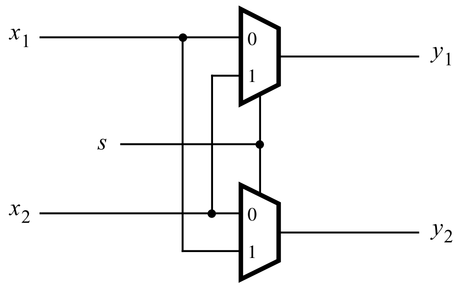
Figure 6.5*b* shows how the 2×2 crossbar can be implemented using 2-to-1 multiplexers. The multiplexer select inputs are controlled by the signal s. If $s = 0$, the crossbar connects x_1 to y_1 and x_2 to y_2, while if $s = 1$, the crossbar connects x_1 to y_2 and x_2 to y_1. Crossbar switches are useful in many practical applications in which it is necessary to be able to connect one set of wires to another set of wires, where the connection pattern changes from time to time.

Example 6.2

We introduced field-programmable gate array (FPGA) chips in section 3.6.5. Figure 3.39 depicts a small FPGA that is programmed to implement a particular circuit. The logic blocks in the FPGA have two inputs, and there are four tracks in each routing channel. Each of the programmable switches that connects a logic block input or output to an interconnection wire is shown as an X. A small part of Figure 3.39 is reproduced in Figure 6.6*a*. For clarity,

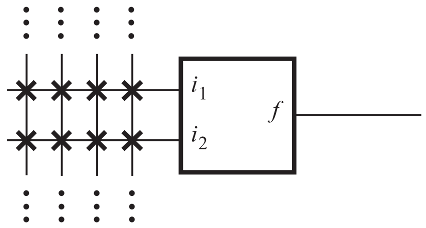


(a) A 2x2 crossbar switch

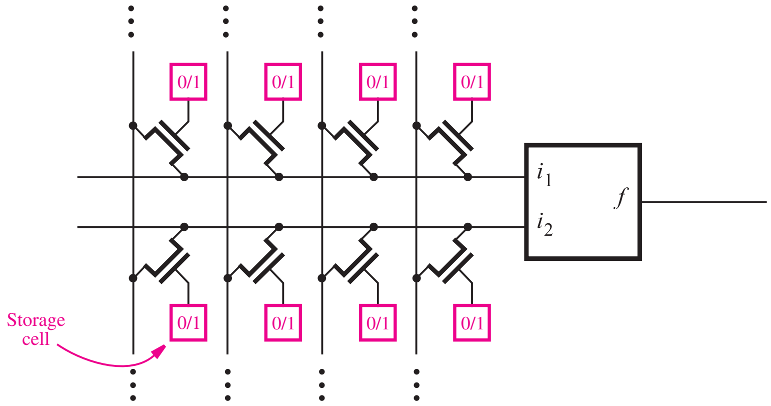


(b) Implementation using multiplexers

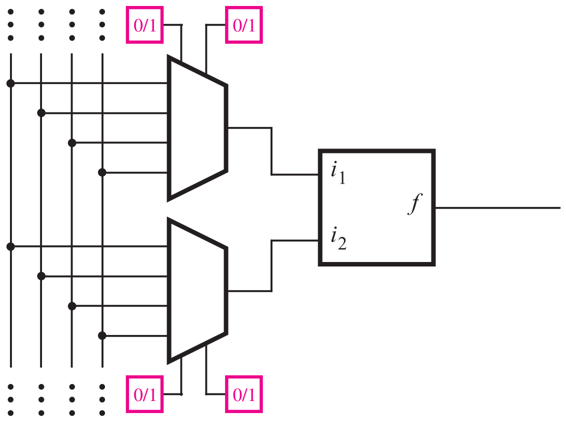
Figure 6.5 A practical application of multiplexers.



(a) Part of the FPGA in Figure 3.39



(b) Implementation using pass transistors



(c) Implementation using multiplexers

Figure 6.6 Implementing programmable switches in an FPGA.

the figure shows only a single logic block and the interconnection wires and switches associated with its input terminals.

One way in which the programmable switches can be implemented is illustrated in Figure 6.6*b*. Each X in part (*a*) of the figure is realized using an NMOS transistor controlled by a storage cell. This type of programmable switch was also shown in Figure 3.68. We described storage cells briefly in section 3.6.5 and will discuss them in more detail in section 10.1. Each cell stores a single logic value, either 0 or 1, and provides this value as the output of the cell. Each storage cell is built by using several transistors. Thus the eight cells shown in the figure use a significant amount of chip area.

The number of storage cells needed can be reduced by using multiplexers, as shown in Figure 6.6*c*. Each logic block input is fed by a 4-to-1 multiplexer, with the select inputs controlled by storage cells. This approach requires only four storage cells, instead of eight. In commercial FPGAs the multiplexer-based approach is usually adopted.

6.1.1 Synthesis of Logic Functions Using Multiplexers

Multiplexers are useful in many practical applications, such as those described above. They can also be used in a more general way to synthesize logic functions. Consider the example in Figure 6.7*a*. The truth table defines the function $f = w_1 \oplus w_2$. This function can be implemented by a 4-to-1 multiplexer in which the values of f in each row of the truth table are connected as constants to the multiplexer data inputs. The multiplexer select inputs are driven by w_1 and w_2. Thus for each valuation of $w_1 w_2$, the output f is equal to the function value in the corresponding row of the truth table.

The above implementation is straightforward, but it is not very efficient. A better implementation can be derived by manipulating the truth table as indicated in Figure 6.7*b*, which allows f to be implemented by a single 2-to-1 multiplexer. One of the input signals, w_1 in this example, is chosen as the select input of the 2-to-1 multiplexer. The truth table is redrawn to indicate the value of f for each value of w_1. When $w_1 = 0, f$ has the same value as input w_2, and when $w_1 = 1, f$ has the value of $\overline{w}_2$. The circuit that implements this truth table is given in Figure 6.7*c*. This procedure can be applied to synthesize a circuit that implements any logic function.

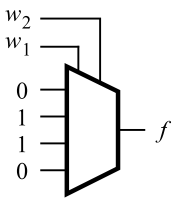
Example 6.3

Figure 6.8*a* gives the truth table for the three-input majority function, and it shows how the truth table can be modified to implement the function using a 4-to-1 multiplexer. Any two of the three inputs may be chosen as the multiplexer select inputs. We have chosen w_1 and w_2 for this purpose, resulting in the circuit in Figure 6.8*b*.

Example 6.4

Figure 6.9*a* indicates how the function $f = w_1 \oplus w_2 \oplus w_3$ can be implemented using 2-to-1 multiplexers. When $w_1 = 0, f$ is equal to the XOR of w_2 and w_3, and when $w_1 = 1, f$ is the

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

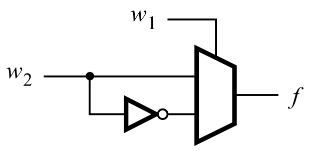


(a) Implementation using a 4-to-1 multiplexer

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\overline{w}_2$

(b) Modified truth table



(c) Circuit

Figure 6.7 Synthesis of a logic function using mutiplexers.

XNOR of w_2 and w_3. The left multiplexer in the circuit produces $w_2 \oplus w_3$, using the result from Figure 6.7, and the right multiplexer uses the value of w_1 to select either $w_2 \oplus w_3$ or its complement. Note that we could have derived this circuit directly by writing the function as $f = (w_2 \oplus w_3) \oplus w_1$.

Figure 6.10 gives an implementation of the three-input XOR function using a 4-to-1 multiplexer. Choosing w_1 and w_2 for the select inputs results in the circuit shown.

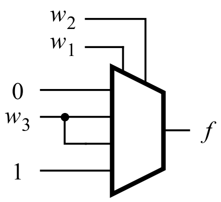
6.1.2 Multiplexer Synthesis Using Shannon's Expansion

Figures 6.8 through 6.10 illustrate how truth tables can be interpreted to implement logic functions using multiplexers. In each case the inputs to the multiplexers are the constants 0 and 1, or some variable or its complement. Besides using such simple inputs, it is

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

(a) Modified truth table

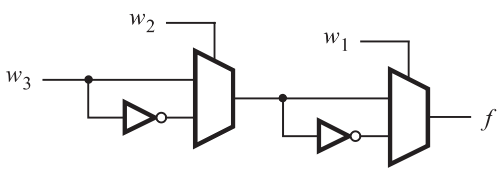


(b) Circuit

Figure 6.8 Implementation of the three-input majority function using a 4-to-1 multiplexer.

w_1	w_2	w_3	f	
0	0	0	0	$w_2 \oplus w_3$
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	$\overline{w_2 \oplus w_3}$
1	0	1	0	
1	1	0	0	
1	1	1	1	

(a) Truth table

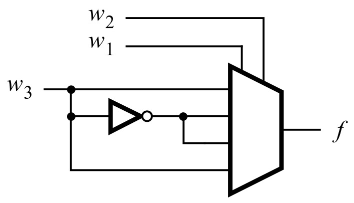


(b) Circuit

Figure 6.9 Three-input XOR implemented with 2-to-1 multiplexers.

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\overline{w}_3$
0	1	1	0	
1	0	0	1	$\overline{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

(a) Truth table



(b) Circuit

Figure 6.10 Three-input XOR implemented with a 4-to-1 multiplexer.

possible to connect more complex circuits as inputs to a multiplexer, allowing functions to be synthesized using a combination of multiplexers and other logic gates. Suppose that we want to implement the three-input majority function in Figure 6.8 using a 2-to-1 multiplexer in this way. Figure 6.11 shows an intuitive way of realizing this function. The truth table can be modified as shown on the right. If $w_1 = 0$, then $f = w_2w_3$, and if $w_1 = 1$, then $f = w_2 + w_3$. Using w_1 as the select input for a 2-to-1 multiplexer leads to the circuit in Figure 6.11*b*.

This implementation can be derived using algebraic manipulation as follows. The function in Figure 6.11*a* is expressed in sum-of-products form as

$$f = \overline{w}_1 w_2 w_3 + w_1 \overline{w}_2 w_3 + w_1 w_2 \overline{w}_3 + w_1 w_2 w_3$$

It can be manipulated into

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(\overline{w}_2 w_3 + w_2\overline{w}_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

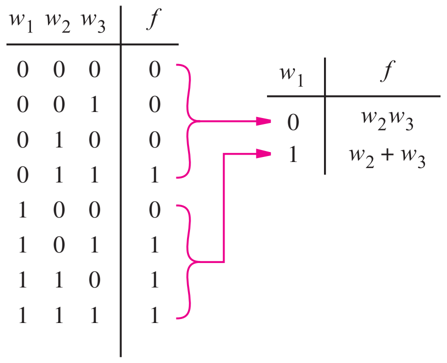
which corresponds to the circuit in Figure 6.11*b*.

Multiplexer implementations of logic functions require that a given function be decomposed in terms of the variables that are used as the select inputs. This can be accomplished by means of a theorem proposed by Claude Shannon [1].

Shannon's Expansion Theorem Any Boolean function $f(w_1, \ldots, w_n)$ can be written in the form

$$f(w_1, w_2, \ldots, w_n) = \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$

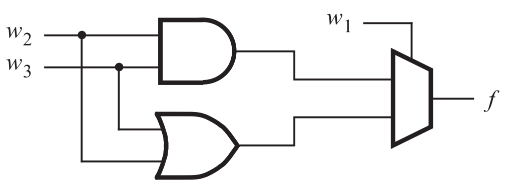
This expansion can be done in terms of any of the n variables. We will leave the proof of the theorem as an exercise for the reader (see problem 6.9).



w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	w_2w_3
1	$w_2 + w_3$

(a) Truth table



(b) Circuit

Figure 6.11 The three-input majority function implemented using a 2-to-1 multiplexer.

To illustrate its use, we can apply the theorem to the three-input majority function, which can be written as

$$f(w_1, w_2, w_3) = w_1w_2 + w_1w_3 + w_2w_3$$

Expanding this function in terms of w_1 gives

$$f = \overline{w}_1(w_2w_3) + w_1(w_2 + w_3)$$

which is the expression that we derived above.

For the three-input XOR function, we have

$$\begin{aligned} f &= w_1 \oplus w_2 \oplus w_3 \\ &= \overline{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot (\overline{w_2 \oplus w_3}) \end{aligned}$$

which gives the circuit in Figure 6.9*b*.

In Shannon's expansion the term $f(0, w_2, \ldots, w_n)$ is called the *cofactor* of f with respect to $\overline{w}_1$; it is denoted in shorthand notation as $f_{\overline{w}_1}$. Similarly, the term $f(1, w_2, \ldots, w_n)$ is called the cofactor of f with respect to w_1, written f_{w_1}. Hence we can write

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

In general, if the expansion is done with respect to variable w_i, then f_{w_i} denotes $f(w_1, \ldots, w_{i-1}, 1, w_{i+1}, \ldots, w_n)$ and

$$f(w_1, \ldots, w_n) = \overline{w}_i f_{\overline{w}_i} + w_i f_{w_i}$$

The complexity of the logic expression may vary, depending on which variable, w_i, is used, as illustrated in Example 6.5.

Example 6.5 For the function $f = \overline{w}_1 w_3 + w_2 \overline{w}_3$, decomposition using w_1 gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1 (w_3 + w_2) + w_1 (w_2 \overline{w}_3) \end{aligned}$$

Using w_2 instead of w_1 produces

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2 (\overline{w}_1 w_3) + w_2 (\overline{w}_1 + \overline{w}_3) \end{aligned}$$

Finally, using w_3 gives

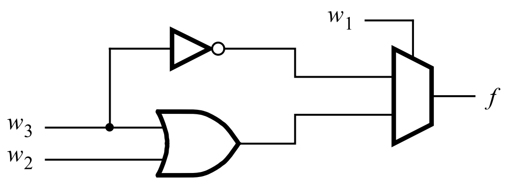
$$\begin{aligned} f &= \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3} \\ &= \overline{w}_3 (w_2) + w_3 (\overline{w}_1) \end{aligned}$$

The results generated using w_1 and w_2 have the same cost, but the expression produced using w_3 has a lower cost. In practice, the CAD tools that perform decompositions of this type try a number of alternatives and choose the one that produces the best result.

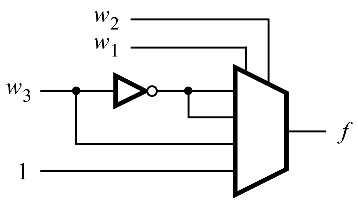
Shannon's expansion can be done in terms of more than one variable. For example, expanding a function in terms of w_1 and w_2 gives

$$\begin{aligned} f(w_1, \ldots, w_n) = {} & \overline{w}_1 \overline{w}_2 \cdot f(0, 0, w_3, \ldots, w_n) + \overline{w}_1 w_2 \cdot f(0, 1, w_3, \ldots, w_n) \\ & + w_1 \overline{w}_2 \cdot f(1, 0, w_3, \ldots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n) \end{aligned}$$

This expansion gives a form that can be implemented using a 4-to-1 multiplexer. If Shannon's expansion is done in terms of all n variables, then the result is the canonical sum-of-products form, which was defined in section 2.6.1.



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

Figure 6.12 The circuits synthesized in Example 6.6.

Example 6.6

Assume that we wish to implement the function

$$f = \overline{w}_1\overline{w}_3 + w_1w_2 + w_1w_3$$

using a 2-to-1 multiplexer and any other necessary gates. Shannon's expansion using w_1 gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3) \end{aligned}$$

The corresponding circuit is shown in Figure 6.12*a*. Assume now that we wish to use a 4-to-1 multiplexer instead. Further decomposition using w_2 gives

$$\begin{aligned} f &= \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1 w_2 f_{w_1 w_2} \\ &= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1 w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1 w_2(1) \end{aligned}$$

The circuit is shown in Figure 6.12*b*.

Example 6.7

Consider the three-input majority function

$$f = w_1w_2 + w_1w_3 + w_2w_3$$

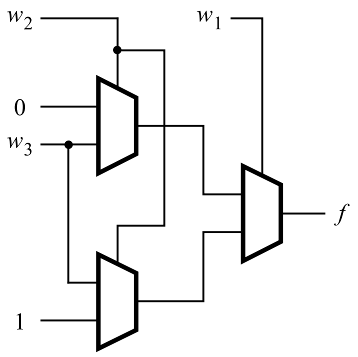


Figure 6.13 The circuit synthesized in Example 6.7.

We wish to implement this function using only 2-to-1 multiplexers. Shannon's expansion using w_1 yields

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

Let $g = w_2w_3$ and $h = w_2 + w_3$. Expansion of both g and h using w_2 gives

$$\begin{aligned} g &= \overline{w}_2(0) + w_2(w_3) \\ h &= \overline{w}_2(w_3) + w_2(1) \end{aligned}$$

The corresponding circuit is shown in Figure 6.13. It is equivalent to the 4-to-1 multiplexer circuit derived using a truth table in Figure 6.8.

Example 6.8 In section 3.6.5 we said that most FPGAs use lookup tables for their logic blocks. Assume that an FPGA exists in which each logic block is a three-input lookup table (3-LUT). Because it stores a truth table, a 3-LUT can realize any logic function of three variables. Using Shannon's expansion, any four-variable function can be realized with at most three 3-LUTs. Consider the function

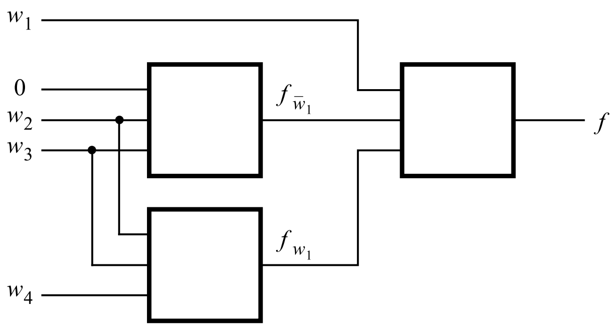
$$f = \overline{w}_2w_3 + \overline{w}_1w_2\overline{w}_3 + w_2\overline{w}_3w_4 + w_1\overline{w}_2\overline{w}_4$$

Expansion in terms of w_1 produces

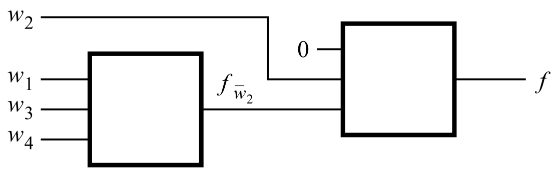
$$\begin{aligned} f &= \overline{w}_1f_{\overline{w}_1} + w_1f_{w_1} \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2\overline{w}_3w_4) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \end{aligned}$$

A circuit with three 3-LUTs that implements this expression is shown in Figure 6.14*a*. Decomposition of the function using w_2, instead of w_1, gives

$$\begin{aligned} f &= \overline{w}_2f_{\overline{w}_2} + w_2f_{w_2} \\ &= \overline{w}_2(w_3 + w_1\overline{w}_4) + w_2(\overline{w}_1\overline{w}_3 + \overline{w}_3w_4) \end{aligned}$$



(a) Using three 3-LUTs



(b) Using two 3-LUTs

Figure 6.14 Circuits synthesized in Example 6.8.

Observe that $\overline{f}_{\overline{w}_2} = f_{w_2}$; hence only two 3-LUTs are needed, as illustrated in Figure 6.14*b*. The LUT on the right implements the two-variable function $\overline{w}_2 f_{\overline{w}_2} + w_2 \overline{f}_{\overline{w}_2}$.

Since it is possible to implement any logic function using multiplexers, general-purpose chips exist that contain multiplexers as their basic logic resources. Both Actel Corporation [2] and QuickLogic Corporation [3] offer FPGAs in which the logic block comprises an arrangement of multiplexers. Texas Instruments offers gate array chips that have multiplexer-based logic blocks [4].

6.2 Decoders

Decoder circuits are used to decode encoded information. A binary decoder, depicted in Figure 6.15, is a logic circuit with n inputs and 2^n outputs. Only one output is asserted at a time, and each output corresponds to one valuation of the inputs. The decoder also has an enable input, En, that is used to disable the outputs; if $En = 0$, then none of the decoder outputs is asserted. If $En = 1$, the valuation of $w_{n-1} \cdots w_1 w_0$ determines which of the outputs is asserted. An n-bit binary code in which exactly one of the bits is set to 1 at a

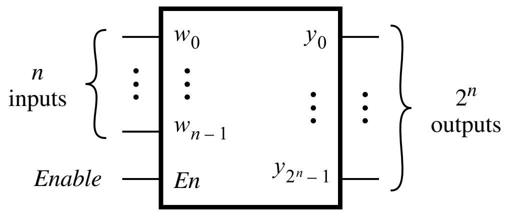


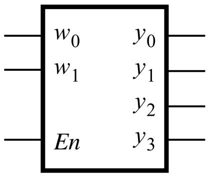
Figure 6.15 An n-to-2^n binary decoder.

time is referred to as *one-hot encoded*, meaning that the single bit that is set to 1 is deemed to be "hot." The outputs of a binary decoder are one-hot encoded.

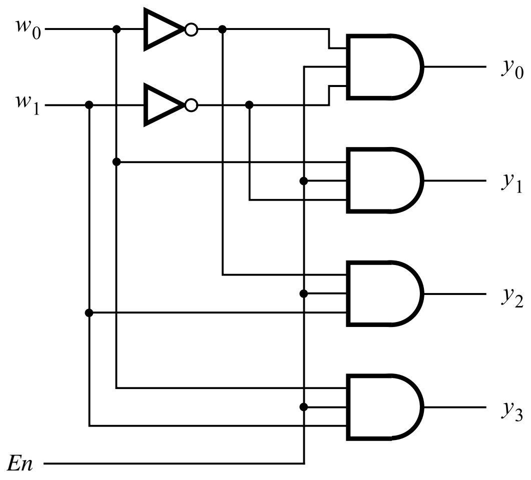
A 2-to-4 decoder is given in Figure 6.16. The two data inputs are w_1 and w_0. They represent a two-bit number that causes the decoder to assert one of the outputs $y_0, \ldots, y_3$. Although a decoder can be designed to have either active-high or active-low outputs, in

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



(b) Graphical symbol



(c) Logic circuit

Figure 6.16 A 2-to-4 decoder.

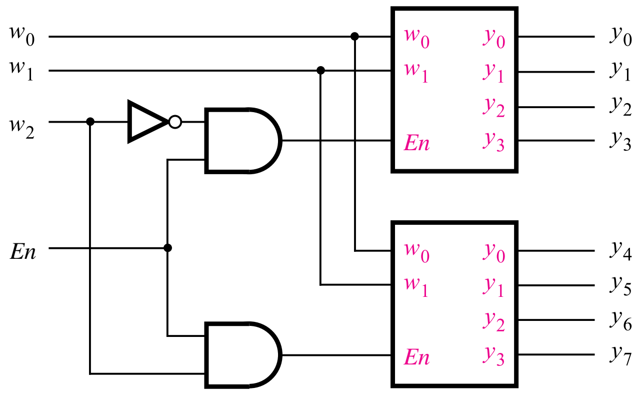


Figure 6.17 A 3-to-8 decoder using two 2-to-4 decoders.

Figure 6.16 active-high outputs are assumed. Setting the inputs w_1w_0 to 00, 01, 10, or 11 causes the output y_0, y_1, y_2, or y_3 to be set to 1, respectively. A graphical symbol for the decoder is given in part (*b*) of the figure, and a logic circuit is shown in part (*c*).

Larger decoders can be built using the sum-of-products structure in Figure 6.16*c*, or else they can be constructed from smaller decoders. Figure 6.17 shows how a 3-to-8 decoder is built with two 2-to-4 decoders. The w_2 input drives the enable inputs of the two decoders. The top decoder is enabled if $w_2 = 0$, and the bottom decoder is enabled if $w_2 = 1$. This concept can be applied for decoders of any size. Figure 6.18 shows how five 2-to-4 decoders can be used to construct a 4-to-16 decoder. Because of its treelike structure, this type of circuit is often referred to as a *decoder tree*.

Example 6.9

Decoders are useful for many practical purposes. In Figure 6.2*c* we showed the sum-of-products implementation of the 4-to-1 multiplexer, which requires AND gates to distinguish the four different valuations of the select inputs s_1 and s_0. Since a decoder evaluates the values on its inputs, it can be used to build a multiplexer as illustrated in Figure 6.19. The enable input of the decoder is not needed in this case, and it is set to 1. The four outputs of the decoder represent the four valuations of the select inputs.

Example 6.10

In Figure 3.59 we showed how a 2-to-1 multiplexer can be constructed using two tri-state buffers. This concept can be applied to any size of multiplexer, with the addition of a decoder. An example is shown in Figure 6.20. The decoder enables one of the tri-state buffers for each valuation of the select lines, and that tri-state buffer drives the output, f, with the selected data input. We have now seen that multiplexers can be implemented in various ways. The choice of whether to employ the sum-of-products form, transmission gates, or tri-state buffers depends on the resources available in the chip being used. For instance, most FPGAs that use lookup tables for their logic blocks do not contain tri-state

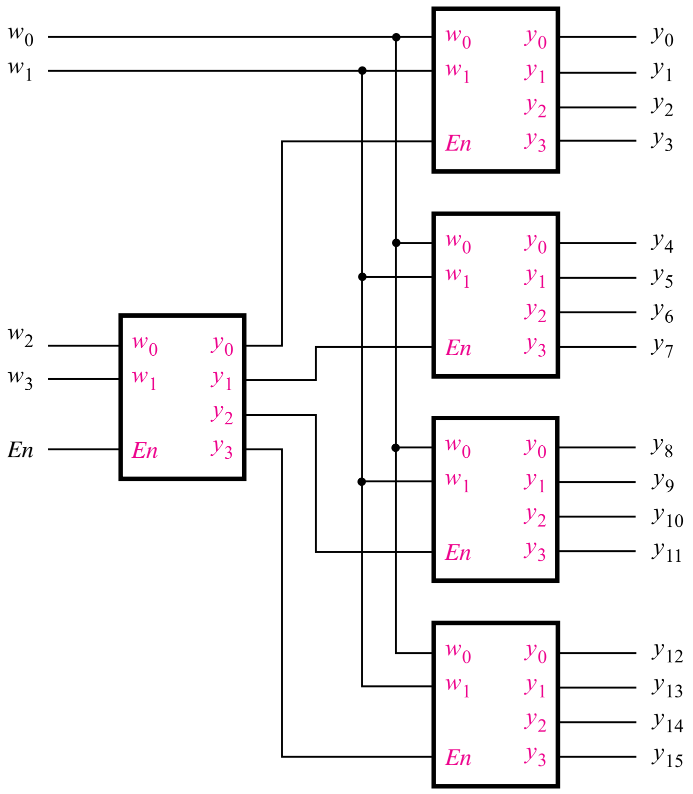


Figure 6.18 A 4-to-16 decoder built using a decoder tree.

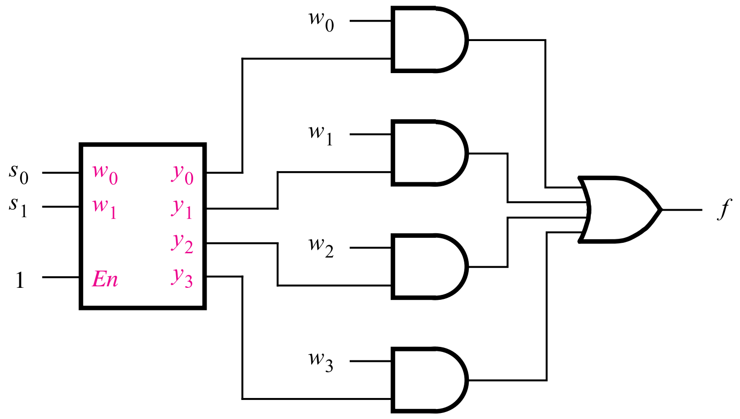


Figure 6.19 A 4-to-1 multiplexer built using a decoder.

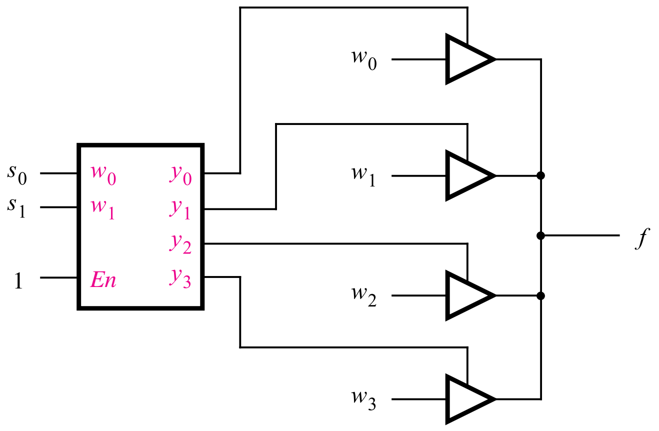


Figure 6.20 A 4-to-1 multiplexer built using a decoder and tri-state buffers.

buffers. Hence multiplexers must be implemented in the sum-of-products form using the lookup tables (see Example 6.33).

6.2.1 Demultiplexers

We showed in section 6.1 that a multiplexer has one output, n data inputs, and $\lceil \log_2 n \rceil$ select inputs. The purpose of the multiplexer circuit is to *multiplex* the n data inputs onto the single data output under control of the select inputs. A circuit that performs the opposite function, namely, placing the value of a single data input onto multiple data outputs, is called a *demultiplexer*. The demultiplexer can be implemented using a decoder circuit. For example, the 2-to-4 decoder in Figure 6.16 can be used as a 1-to-4 demultiplexer. In this case the En input serves as the data input for the demultiplexer, and the y_0 to y_3 outputs are the data outputs. The valuation of $w_1 w_0$ determines which of the outputs is set to the value of En. To see how the circuit works, consider the truth table in Figure 6.16*a*. When $En = 0$, all the outputs are set to 0, including the one selected by the valuation of $w_1 w_0$. When $En = 1$, the valuation of $w_1 w_0$ sets the appropriate output to 1.

In general, an n-to-2^n decoder circuit can be used as a 1-to-n demultiplexer. However, in practice decoder circuits are used much more often as decoders rather than as demultiplexers. In many applications the decoder's En input is not actually needed; hence it can be omitted. In this case the decoder always asserts one of its data outputs, $y_0, \ldots, y_{2^n-1}$, according to the valuation of the data inputs, $w_{n-1} \cdots w_0$. Example 6.11 uses a decoder that does not have the En input.

Example 6.11 One of the most important applications of decoders is in memory blocks, which are used to store information. Such memory blocks are included in digital systems, such as computers, where there is a need to store large amounts of information electronically. One type of memory block is called a *read-only memory* (ROM). A ROM consists of a collection of storage cells, where each cell permanently stores a single logic value, either 0 or 1. Figure 6.21 shows an example of a ROM block. The storage cells are arranged in 2^m rows with n cells per row. Thus each row stores n bits of information. The location of each row in the ROM is identified by its *address*. In the figure the row at the top of the ROM has address 0, and the row at the bottom has address $2^m - 1$. The information stored in the rows can be accessed by asserting the select lines, Sel_0 to Sel_{2^m-1}. As shown in the figure, a decoder with m inputs and 2^m outputs is used to generate the signals on the select lines. Since the inputs to the decoder choose the particular address (row) selected, they are called the *address* lines. The information stored in the row appears on the data outputs of the ROM, $d_{n-1}, \ldots, d_0$, which are called the *data* lines. Figure 6.21 shows that each data line has an associated tri-state buffer that is enabled by the ROM input named *Read*. To access, or *read*, data from the ROM, the address of the desired row is placed on the address lines and *Read* is set to 1.

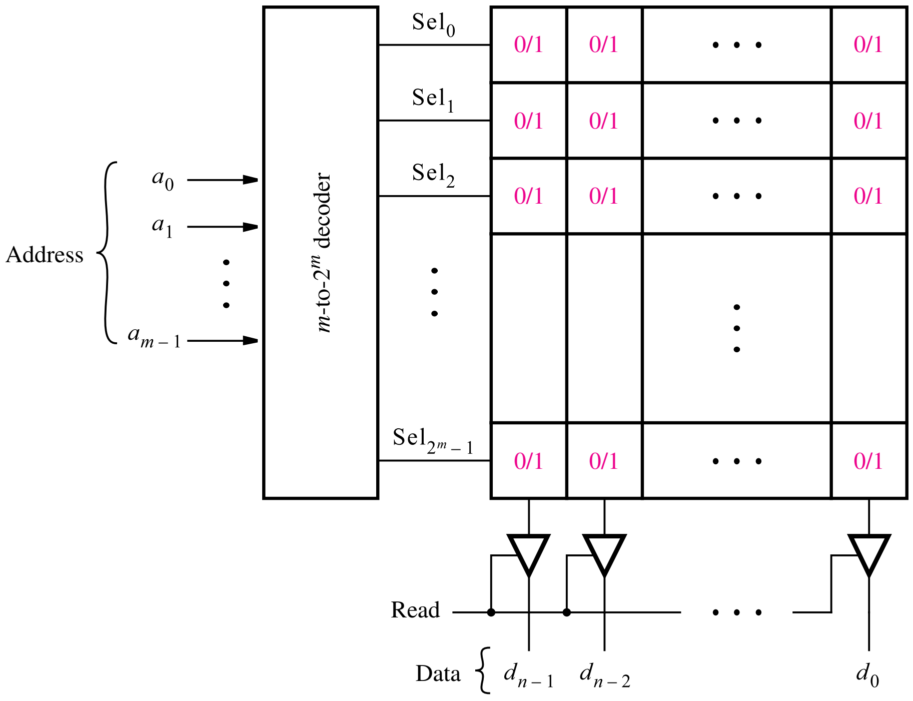


Figure 6.21 A $2^m \times n$ read-only memory (ROM) block.

Many different types of memory blocks exist. In a ROM the stored information can be read out of the storage cells, but it cannot be changed (see problem 6.31). Another type of ROM allows information to be both read out of the storage cells and stored, or *written*, into them. Reading its contents is the normal operation, whereas writing requires a special procedure. Such a memory block is called a programmable ROM (PROM). The storage cells in a PROM are usually implemented using EEPROM transistors. We discussed EEPROM transistors in section 3.10 to show how they are used in PLDs. Other types of memory blocks are discussed in section 10.1.

6.3 Encoders

An encoder performs the opposite function of a decoder. It encodes given information into a more compact form.

6.3.1 Binary Encoders

A *binary encoder* encodes information from 2^n inputs into an n-bit code, as indicated in Figure 6.22. Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1. The truth table for a 4-to-2 encoder is provided in Figure 6.23*a*. Observe that the output y_0 is 1 when either input w_1 or w_3 is 1, and output y_1 is 1 when input w_2 or w_3 is 1. Hence these outputs can be generated by the circuit in Figure 6.23*b*. Note that we assume that the inputs are one-hot encoded. All input patterns that have multiple inputs set to 1 are not shown in the truth table, and they are treated as don't-care conditions.

Encoders are used to reduce the number of bits needed to represent given information. A practical use of encoders is for transmitting information in a digital system. Encoding the information allows the transmission link to be built using fewer wires. Encoding is also useful if information is to be stored for later use because fewer bits need to be stored.

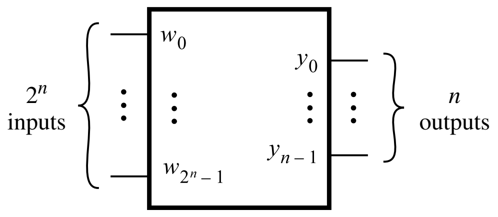
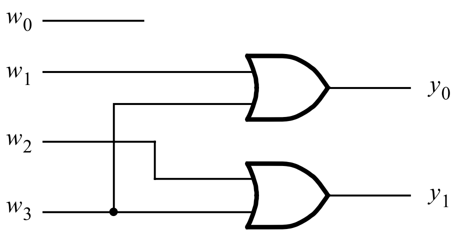


Figure 6.22 A 2^n-to-n binary encoder.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

Figure 6.23 A 4-to-2 binary encoder.

6.3.2 Priority Encoders

Another useful class of encoders is based on the priority of input signals. In a *priority encoder* each input has a priority level associated with it. The encoder outputs indicate the active input that has the highest priority. When an input with a high priority is asserted, the other inputs with lower priority are ignored. The truth table for a 4-to-2 priority encoder is shown in Figure 6.24. It assumes that w_0 has the lowest priority and w_3 the highest. The outputs y_1 and y_0 represent the binary number that identifies the highest priority input set to 1. Since it is possible that none of the inputs is equal to 1, an output, z, is provided to indicate this condition. It is set to 1 when at least one of the inputs is equal to 1. It is set to

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figure 6.24 Truth table for a 4-to-2 priority encoder.

0 when all inputs are equal to 0. The outputs y_1 and y_0 are not meaningful in this case, and hence the first row of the truth table can be treated as a don't-care condition for y_1 and y_0.

The behavior of the priority encoder is most easily understood by first considering the last row in the truth table. It specifies that if input w_3 is 1, then the outputs are set to $y_1y_0 = 11$. Because w_3 has the highest priority level, the values of inputs w_2, w_1, and w_0 do not matter. To reflect the fact that their values are irrelevant, w_2, w_1, and w_0 are denoted by the symbol x in the truth table. The second-last row in the truth table stipulates that if $w_2 = 1$, then the outputs are set to $y_1y_0 = 10$, but only if $w_3 = 0$. Similarly, input w_1 causes the outputs to be set to $y_1y_0 = 01$ only if both w_3 and w_2 are 0. Input w_0 produces the outputs $y_1y_0 = 00$ only if w_0 is the only input that is asserted.

A logic circuit that implements the truth table can be synthesized by using the techniques developed in Chapter 4. However, a more convenient way to derive the circuit is to define a set of intermediate signals, $i_0, \ldots, i_3$, based on the observations above. Each signal, i_k, is equal to 1 only if the input with the same index, w_k, represents the highest-priority input that is set to 1. The logic expressions for $i_0, \ldots, i_3$ are

$$\begin{aligned} i_0 &= \overline{w}_3\overline{w}_2\overline{w}_1w_0 \\ i_1 &= \overline{w}_3\overline{w}_2w_1 \\ i_2 &= \overline{w}_3w_2 \\ i_3 &= w_3 \end{aligned}$$

Using the intermediate signals, the rest of the circuit for the priority encoder has the same structure as the binary encoder in Figure 6.23, namely

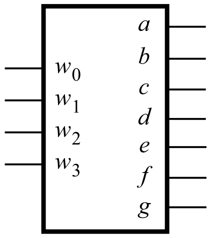
$$\begin{aligned} y_0 &= i_1 + i_3 \\ y_1 &= i_2 + i_3 \end{aligned}$$

The output z is given by

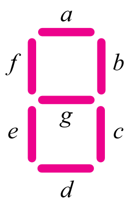
$$z = i_0 + i_1 + i_2 + i_3$$

6.4 Code Converters

The purpose of the decoder and encoder circuits is to convert from one type of input encoding to a different output encoding. For example, a 3-to-8 binary decoder converts from a binary number on the input to a one-hot encoding at the output. An 8-to-3 binary encoder performs the opposite conversion. There are many other possible types of code converters. One common example is a BCD-to-7-segment decoder, which converts one binary-coded decimal (BCD) digit into information suitable for driving a digit-oriented display. As illustrated in Figure 6.25*a*, the circuit converts the BCD digit into seven signals that are used to drive the segments in the display. Each segment is a small light-emitting diode (LED), which glows when driven by an electrical signal. The segments are labeled from *a* to *g* in the figure. The truth table for the BCD-to-7-segment decoder is given in Figure 6.25*c*. For each valuation of the inputs $w_3, \ldots, w_0$, the seven outputs are set to



(a) Code converter



(b) 7-segment display

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
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	1	0	1	1

(c) Truth table

Figure 6.25 A BCD-to-7-segment display code converter.

display the appropriate BCD digit. Note that the last 6 rows of a complete 16-row truth table are not shown. They represent don't-care conditions because they are not legal BCD codes and will never occur in a circuit that deals with BCD data. A circuit that implements the truth table can be derived using the synthesis techniques discussed in Chapter 4. Finally, we should note that although the word *decoder* is traditionally used for this circuit, a more appropriate term is *code converter*. The term *decoder* is more appropriate for circuits that produce one-hot encoded outputs.

6.5 Arithmetic Comparison Circuits

Chapter 5 presented arithmetic circuits that perform addition, subtraction, and multiplication of binary numbers. Another useful type of arithmetic circuit compares the relative sizes of two binary numbers. Such a circuit is called a *comparator*. This section considers the design of a comparator that has two n-bit inputs, A and B, which represent unsigned binary numbers. The comparator produces three outputs, called $AeqB$, $AgtB$, and $AltB$. The $AeqB$ output is set to 1 if A and B are equal. The $AgtB$ output is 1 if A is greater than B, and the $AltB$ output is 1 if A is less than B.

The desired comparator can be designed by creating a truth table that specifies the three outputs as functions of A and B. However, even for moderate values of n, the truth table is large. A better approach is to derive the comparator circuit by considering the bits of A and B in pairs. We can illustrate this by a small example, where $n = 4$.

Let $A = a_3a_2a_1a_0$ and $B = b_3b_2b_1b_0$. Define a set of intermediate signals called i_3, i_2, i_1, and i_0. Each signal, i_k, is 1 if the bits of A and B with the same index are equal. That is, $i_k = \overline{a_k \oplus b_k}$. The comparator's $AeqB$ output is then given by

$$AeqB = i_3i_2i_1i_0$$

An expression for the $AgtB$ output can be derived by considering the bits of A and B in the order from the most-significant bit to the least-significant bit. The first bit-position, k, at which a_k and b_k differ determines whether A is less than or greater than B. If $a_k = 0$ and $b_k = 1$, then $A < B$. But if $a_k = 1$ and $b_k = 0$, then $A > B$. The $AgtB$ output is defined by

$$AgtB = a_3\overline{b}_3 + i_3a_2\overline{b}_2 + i_3i_2a_1\overline{b}_1 + i_3i_2i_1a_0\overline{b}_0$$

The i_k signals ensure that only the first digits, considered from the left to the right, of A and B that differ determine the value of $AgtB$.

The $AltB$ output can be derived by using the other two outputs as

$$AltB = \overline{AeqB + AgtB}$$

A logic circuit that implements the four-bit comparator circuit is shown in Figure 6.26. This approach can be used to design a comparator for any value of n.

Comparator circuits, like most logic circuits, can be designed in different ways. Another approach for designing a comparator circuit is presented in Example 5.10 in Chapter 5.

6.6 Verilog for Combinational Circuits

Having presented a number of useful building block circuits, we will now consider how such circuits can be described in Verilog. Rather than using gates or logic expressions, we will specify the circuits in terms of their behavior. We will also give a more rigorous description of previously used behavioral Verilog constructs and introduce some new ones.

6.6.1 The Conditional Operator

In a logic circuit it is often necessary to choose between several possible signals or values based on the state of some condition. A typical example is a multiplexer circuit in which the output is equal to the data input signal chosen by the valuation of the select inputs. For simple implementation of such choices Verilog provides a *conditional* operator (?:) which

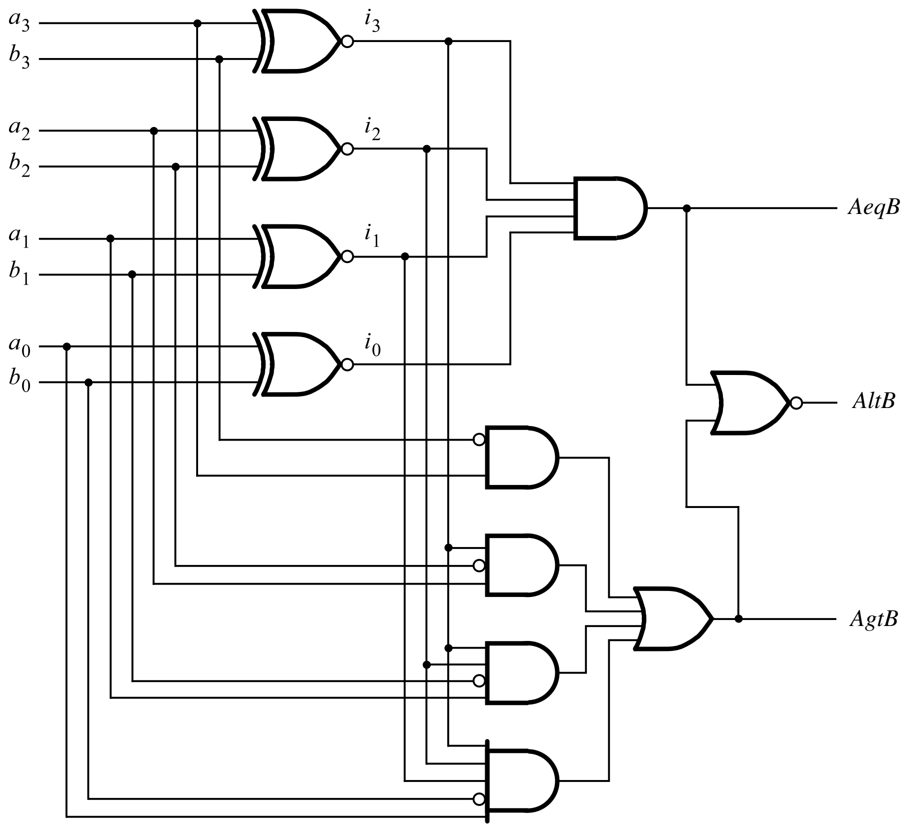


Figure 6.26 A four-bit comparator circuit.

assigns one of two values depending on a conditional expression. It involves three operands used in the syntax

conditional_expression ? true_expression : false_expression

If the conditional expression evaluates to 1 (true), then the value of true_expression is chosen; otherwise, the value of false_expression is chosen. For example, the statement

A = (B < C) ? (D + 5) : (D + 2);

means that if B is less than C, the value of A will be $D + 5$ or else A will have the value $D+2$. We used parentheses in the expression to improve readability; they are not necessary. The conditional operator can be used both in continuous assignment statements and in procedural statements inside an **always** block.

Example 6.12

A 2-to-1 multiplexer can be defined using the conditional operator in an **assign** statement as shown in Figure 6.27. The module, named *mux2to1*, has the inputs w_0, w_1, and s, and the output f. The signal s is used for the selection criterion. The output f is equal to w_1 if the select input s has the value 1; otherwise, f is equal to w_0. Figure 6.28 shows how the same multiplexer can be defined by using the conditional operator inside an **always** block.

The same approach can be used to define a 4-to-1 multiplexer by nesting the conditional operators as indicated in Figure 6.29. The module is named *mux4to1*. Its two select inputs, which are called s_1 and s_0 in Figure 6.2, are represented by the two-bit vector S. The first conditional expression tests the value of bit s_1. If $s_1 = 1$, then s_0 is tested and f is set to w_3

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output f;

   assign f = s ? w1 : w0;

endmodule
```

Figure 6.27 A 2-to-1 multiplexer specified using the conditional operator.

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output reg f;

   always @(w0, w1, s)
      f = s ? w1 : w0;

endmodule
```

Figure 6.28 An alternative specification of a 2-to-1 multiplexer using the conditional operator.

```
module mux4to1 (w0, w1, w2, w3, S, f);
   input w0, w1, w2, w3;
   input [1:0] S;
   output f;

   assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule
```

Figure 6.29 A 4-to-1 multiplexer specified using the conditional operator.

if $s_0 = 1$ and f is set to w_2 if $s_0 = 0$. This corresponds to the third and fourth rows of the truth table in Figure 6.2*b*. Similarly, if $s_1 = 0$ the conditional operator on the right chooses $f = w_1$ if $s_0 = 1$ and $f = w_0$ if $s_0 = 0$, thus realizing the first two rows of the truth table.

6.6.2 The If-Else Statement

We have already used the **if-else** statement in previous chapters. It has the syntax

```
if (conditional_expression) statement;
else statement;
```

The conditional expression may use the operators given in Table A.1. If the expression is evaluated to true then the first statement (or a block of statements delineated by **begin** and **end** keywords) is executed or else the second statement (or a block of statements) is executed.

Example 6.13 Figure 6.30 shows how the **if-else** statement can be used to describe a 2-to-1 multiplexer. The **if** clause states that f is assigned the value of w_0 when $s = 0$. Else, f is assigned the value of w_1.

The **if-else** statement can be used to implement larger multiplexers. A 4-to-1 multiplexer is shown in Figure 6.31. The **if-else** clauses set f to the value of one of the inputs $w_0, \ldots, w_3$, depending on the valuation of S. Compiling the code results in the circuit shown in Figure 6.2*c*.

Another way of defining the same circuit is presented in Figure 6.32. In this case, a four-bit vector W is defined instead of single-bit signals w_0, w_1, w_2, and w_3. Also, the four different values of S are specified as decimal rather than binary numbers.

Example 6.14 Figure 6.4 shows how a 16-to-1 multiplexer is built using five 4-to-1 multiplexers. Figure 6.33 presents Verilog code for this circuit using five instantiations of the *mux4to1* module.

```
module  mux2to1 (w0, w1, s, f);
   input  w0, w1, s;
   output  reg  f;

   always @(w0, w1, s)
      if (s == 0)
         f = w0;
      else
         f = w1;

endmodule
```

Figure 6.30 Code for a 2-to-1 multiplexer using the **if-else** statement.

```
module mux4to1 (w0, w1, w2, w3, S, f);
   input w0, w1, w2, w3;
   input [1:0] S;
   output reg f;

   always @(*)
      if (S == 2'b00)
         f = w0;
      else if (S == 2'b01)
         f = w1;
      else if (S == 2'b10)
         f = w2;
      else
         f = w3;

endmodule
```

Figure 6.31 Code for a 4-to-1 multiplexer using the **if-else** statement.

```
module mux4to1 (W, S, f);
   input [0:3] W;
   input [1:0] S;
   output reg f;

   always @(W, S)
      if (S == 0)
         f = W[0];
      else if (S == 1)
         f = W[1];
      else if (S == 2)
         f = W[2];
      else
         f = W[3];

endmodule
```

Figure 6.32 Alternative specification of a 4-to-1 multiplexer.

The data inputs to the *mux16to1* module are the 16-bit vector W, and the select inputs are the four-bit vector $S16$. In the Verilog code signal names are needed for the outputs of the four 4-to-1 multiplexers on the left of Figure 6.4. A four-bit signal named M is used for this purpose. The first multiplexer instantiated, *Mux1*, corresponds to the multiplexer at the top left of Figure 6.4. Its first four ports, which correspond to $w_0, \ldots, w_3$ in Figure

```
module mux16to1 (W, S16, f);
    input [0:15] W;
    input [3:0] S16;
    output f;
    wire [0:3] M;

    mux4to1 Mux1 (W[0:3], S16[1:0], M[0]);
    mux4to1 Mux2 (W[4:7], S16[1:0], M[1]);
    mux4to1 Mux3 (W[8:11], S16[1:0], M[2]);
    mux4to1 Mux4 (W[12:15], S16[1:0], M[3]);
    mux4to1 Mux5 (M[0:3], S16[3:2], f);

endmodule
```

Figure 6.33 Hierarchical code for a 16-to-1 multiplexer.

6.31, are driven by the signals $W[0], \ldots, W[3]$. The syntax $S16[1{:}0]$ is used to attach the signals $S16[1]$ and $S16[0]$ to the two-bit S port of the *mux4to1* module. The $M[0]$ signal is connected to the multiplexer's output port. Similarly, *Mux2, Mux3*, and *Mux4* are instantiations of the next three multiplexers on the left. The multiplexer on the right of Figure 6.4 is instantiated as *Mux5*. The signals $M[0], \ldots, M[3]$ are connected to its data inputs, and bits $S16[3]$ and $S16[2]$, which are specified by the syntax $S16[3{:}2]$, are attached to the select inputs. The output port generates the *mux16to1* output f. Compiling the code results in the multiplexer function

$$f = \bar{s}_3\bar{s}_2\bar{s}_1\bar{s}_0 w_0 + \bar{s}_3\bar{s}_2\bar{s}_1 s_0 w_1 + \bar{s}_3\bar{s}_2 s_1\bar{s}_0 w_2 + \cdots + s_3 s_2 s_1 \bar{s}_0 w_{14} + s_3 s_2 s_1 s_0 w_{15}$$

Since the *mux4to1* module is being instantiated in the code of Figure 6.33, it is necessary to either include the code of Figure 6.32 in the same file as the *mux16to1* module or place the *mux4to1* module in a separate file in the same directory, or a directory with a specified path so that the Verilog compiler can find it. Observe that if the code in Figure 6.31 were used as the required *mux4to1* module, then we would have to list the ports separately, as in $W[0]$, $W[1]$, $W[2]$, $W[3]$, rather than as the vector $W[0{:}3]$.

6.6.3 The Case Statement

The **if-else** statement provides the means for choosing an alternative based on the value of an expression. When there are many possible alternatives, the code based on this statement may become awkward to read. Instead, it is often possible to use the Verilog **case** statement which is defined as

```
case (expression)
    alternative1: statement;
    alternative2: statement;
    .
    .
    .
    alternativej: statement;
    [default: statement;]
endcase
```

The controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included. Otherwise, the Verilog compiler will synthesize memory elements to deal with the unspecified possibilities; we will discuss this issue in Chapter 7.

Example 6.15

The **case** statement can be used to define a 4-to-1 multiplexer as shown in Figure 6.34. The four values that the select vector S can have are given as decimal numbers, but they could also be given as binary numbers.

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output reg f;

    always @(W, S)
        case (S)
            0: f = W[0];
            1: f = W[1];
            2: f = W[2];
            3: f = W[3];
        endcase

endmodule
```

Figure 6.34 A 4-to-1 multiplexer defined using the **case** statement.

Example 6.16 Figure 6.35 shows how a **case** statement can be used to describe the truth table for a 2-to-4 binary decoder. The module is called *dec2to4*. The data inputs are the two-bit vector W, and the enable input is *En*. The four outputs are represented by the four-bit vector Y.

In the truth table for the decoder in Figure 6.16*a*, the inputs are listed in the order *En* w_1 w_0. To represent these three signals in the controlling expression, the Verilog code uses the concatenate operator to combine the *En* and *W* signals into a three-bit vector. The four alternatives in the **case** statement correspond to the truth table in Figure 6.16*a* where $En = 1$, and the decoder outputs have the same patterns as in the first four rows of the truth table. The last clause uses the **default** keyword and sets the decoder outputs to 0000, because it represents all other cases, namely those where $En = 0$.

Example 6.17 The 2-to-4 decoder can be specified using a combination of **if-else** and **case** statements as given in Figure 6.36. The **case** alternatives are evaluated if $En = 1$; otherwise, all four bits of the output Y are set to the value 0.

Example 6.18 The tree structure of the 4-to-16 decoder in Figure 6.18 can be defined as shown in Figure 6.37. The inputs are a four-bit vector W and an enable signal *En*. The outputs are represented by the 16-bit vector Y. The circuit uses five instances of the 2-to-4 decoder defined in either Figure 6.35 or 6.36. The outputs of the leftmost decoder in Figure 6.18 are denoted as the four-bit vector M in Figure 6.37.

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;

   always @(W, En)
      case ({En, W})
         3'b100: Y = 4'b1000;
         3'b101: Y = 4'b0100;
         3'b110: Y = 4'b0010;
         3'b111: Y = 4'b0001;
         default: Y = 4'b0000;
      endcase

endmodule
```

Figure 6.35 Verilog code for a 2-to-4 binary decoder.

```
module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output reg [0:3] Y;

  always @(W, En)
  begin
    if (En == 0)
      Y = 4'b0000;
    else
      case (W)
        0: Y = 4'b1000;
        1: Y = 4'b0100;
        2: Y = 4'b0010;
        3: Y = 4'b0001;
      endcase
  end

endmodule
```

Figure 6.36 Alternative code for a 2-to-4 binary decoder.

```
module dec4to16 (W, Y, En);
  input [3:0] W;
  input En;
  output [0:15] Y;
  wire [0:3] M;

  dec2to4 Dec1 (W[3:2], M[0:3], En);
  dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
  dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
  dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
  dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

endmodule
```

Figure 6.37 Verilog code for a 4-to-16 decoder.

Example 6.19

Another example of a **case** statement is given in Figure 6.38. The module, *seg7*, represents the BCD-to-7-segment decoder in Figure 6.25. The BCD input is the four-bit vector named *bcd*, and the seven outputs are the seven-bit vector named *leds*. The **case** alternatives are listed so that they resemble the truth table in Figure 6.25*c*. Note that there is a comment to the right of the **case** statement, which labels the seven outputs with the letters from *a*

```
module seg7 (bcd, leds);
   input [3:0] bcd;
   output reg [1:7] leds;

   always @(bcd)
      case (bcd)    //abcdefg
         0: leds = 7'b1111110;
         1: leds = 7'b0110000;
         2: leds = 7'b1101101;
         3: leds = 7'b1111001;
         4: leds = 7'b0110011;
         5: leds = 7'b1011011;
         6: leds = 7'b1011111;
         7: leds = 7'b1110000;
         8: leds = 7'b1111111;
         9: leds = 7'b1111011;
         default: leds = 7'bx;
      endcase

endmodule
```

Figure 6.38 Code for a BCD-to-7-segment decoder.

to g. These labels indicate to the reader the correlation between the bits of the *leds* vector in the Verilog code and the seven segments in Figure 6.25*b*. The final **case** alternative sets all seven bits of *leds* to x. Recall that x is used in Verilog to denote a don't-care condition. This alternative represents the don't-care conditions discussed for Figure 6.25, which are the cases where the *bcd* input does not represent a valid BCD digit.

Example 6.20 An arithmetic logic unit (ALU) is a logic circuit that performs various Boolean and arithmetic operations on n-bit operands. In section 3.5 we discussed a family of standard chips called the 7400-series chips. We said that some of these chips contain basic logic gates, and others provide commonly used logic circuits. One example of an ALU is the chip called the 74381. Table 6.1 specifies the functionality of this chip. It has 2 four-bit data inputs, A and B, a three-bit select input, S, and a four-bit output, F. As the table shows, F is defined by various arithmetic or Boolean operations on the inputs A and B. In this table $+$ means arithmetic addition, and $-$ means arithmetic subtraction. To avoid confusion, the table uses the words XOR, OR, and AND for the Boolean operations. Each Boolean operation is done in a bitwise fashion. For example, $F = A$ AND B produces the four-bit result $f_0 = a_0b_0$, $f_1 = a_1b_1, f_2 = a_2b_2$, and $f_3 = a_3b_3$.

Figure 6.39 shows how the functionality of the 74381 ALU can be described in Verilog code. The case statement shown corresponds directly to Table 6.1. To check the functionality of the code, we synthesized a circuit for implementation in a PLD, and show a timing simulation in Figure 6.40. For each valuation of s, the circuit generates the appropriate Boolean or arithmetic operation.

Table 6.1 The functionality of the 74381 ALU.

Operation	Inputs $s_2\ s_1\ s_0$	Outputs F
Clear	0 0 0	0 0 0 0
B−A	0 0 1	$B - A$
A−B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

```
// 74381 ALU
module alu (s, A, B, F);
   input [2:0] s;
   input [3:0] A, B;
   output reg [3:0] F;

   always @(s, A, B)
      case (s)
         0: F = 4'b0000;
         1: F = B − A;
         2: F = A − B;
         3: F = A + B;
         4: F = A ^ B;
         5: F = A | B;
         6: F = A & B;
         7: F = 4'b1111;
      endcase

endmodule
```

Figure 6.39 Code that represents the functionality of the 74381 ALU chip.

The Casex and Casez Statements

In the **case** statement it is possible to use the logic values 0, 1, *z*, and *x* in the **case** alternatives. A bit-by-bit comparison is used to determine the match between the expression and one of the alternatives.

Verilog provides two variants of the **case** statement that treat the *z* and *x* values in a different way. The **casez** statement treats all *z* values in the case alternatives and the

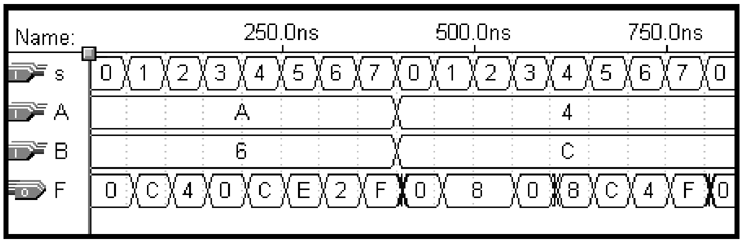


Figure 6.40 Timing simulation for the code in Figure 6.39.

controlling expression as don't cares. The **casex** statement treats all z and x values as don't cares.

Example 6.21 Figure 6.41 gives Verilog code for the priority encoder defined in Figure 6.24. The desired priority scheme is realized by using a **casex** statement. The first alternative specifies that, if the input w_3 is 1, then the output is set to $y_1y_0 = 3$. This assignment does not depend on the values of inputs w_2, w_1, or w_0; hence their values do not matter. The other alternatives in the **casex** statement are evaluated only if $w_3 = 0$. The second alternative states that if w_2 is 1, then $y_1y_0 = 2$. If $w_2 = 0$, then the next alternative results in $y_1y_0 = 1$ if $w_1 = 1$. If $w_3 = w_2 = w_1 = 0$ and $w_0 = 1$, then the fourth alternative results in $y_1y_0 = 0$.

```
module priority (W, Y, z);
   input [3:0] W;
   output reg [1:0] Y;
   output reg z;

   always @(W)
   begin
      z = 1;
      casex (W)
         4'b1xxx: Y = 3;
         4'b01xx: Y = 2;
         4'b001x: Y = 1;
         4'b0001: Y = 0;
         default: begin
                     z = 0;
                     Y = 2'bx;
                  end
      endcase
   end

endmodule
```

Figure 6.41 Verilog code for a priority encoder.

The priority encoder's output z must be set to 1 whenever at least one of the data inputs is 1. This output is set to 1 at the start of the **always** block. If none of the four alternatives matches the value of W, then the **default** clause is executed. It consists of a two-statement block that resets z to 0 and indicates that the Y output can be set to any pattern because it will be ignored.

6.6.4 The For Loop

If the structure of a desired circuit exhibits a certain regularity, it may be possible to define the circuit using a **for** loop. We introduced the **for** loop in section 5.5.4, where it was useful in a generic specification of a ripple-carry adder. The **for** loop has the syntax

for (initial_index; terminal_index; increment) statement;

A loop control variable, which has to be of type **integer**, is set to the value given as the initial index. It is used in the statement or a block of statements delineated by **begin** and **end** keywords. After each iteration, the control variable is changed as defined in the increment. The iterations end after the control variable has reached the terminal index.

Unlike **for** loops in high-level programming languages, the Verilog **for** loop does not specify changes that take place in time through successive loop iterations. Instead, during each iteration it specifies a different subcircuit. In Figure 5.28 the **for** loop was used to define a cascade of full-adder subcircuits to form an n-bit ripple-carry adder. The **for** loop can be used to define many other structures as illustrated by the next two examples.

Example 6.22

Figure 6.42 shows how the **for** loop can be used to specify a 2-to-4 decoder circuit. The effect of the loop is to repeat the **if-else** statement four times, for $k = 0, \ldots, 3$. The first

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;
   integer k;

   always @(W, En)
      for (k = 0; k < = 3; k = k+1)
         if ((W == k) && (En == 1))
            Y[k] = 1;
         else
            Y[k] = 0;

endmodule
```

Figure 6.42 A 2-to-4 binary decoder specified using the **for** loop.

loop iteration sets $y_0 = 1$ if $W = 0$ and $En = 1$. Similarly, the other three iterations set the values of y_1, y_2, and y_3 according to the values of W and En.

This arrangement can be used to specify a large n-to-2^n decoder simply by increasing the sizes of vectors W and Y accordingly, and making $n - 1$ be the terminal index value of k.

Example 6.23 The priority encoder of Figure 6.24 can be defined by the Verilog code in Figure 6.43. In the **always** block, the output bits y_1 and y_0 are first set to the don't-care state and z is cleared to 0. Then, if one or more of the four inputs $w_3, \ldots, w_0$ is equal to 1, the **for** loop will set the valuation of y_1y_0 to match the index of the highest priority input that has the value 1. Note that each successive iteration through the loop corresponds to a higher priority. Verilog semantics specify that a signal that receives multiple assignments in an **always** block retains the last assignment. Thus the iteration that corresponds to the highest priority input that is equal to 1 will override any setting of Y established during the previous iterations.

6.6.5 Verilog Operators

In this section we discuss the Verilog operators that are useful for synthesizing logic circuits. Table 6.2 lists these operators in groups that reflect the type of operation performed. A more complete listing of the operators is given in Table A.1.

```
module priority (W, Y, z);
   input [3:0] W;
   output reg [1:0] Y;
   output reg z;
   integer k;

   always @(W)
   begin
      Y = 2'bx;
      z = 0;
      for (k = 0; k < 4; k = k+1)
         if (W[k])
         begin
            Y = k;
            z = 1;
         end
   end

endmodule
```

Figure 6.43 A priority encoder specified using the **for** loop.

Table 6.2 Verilog operators.

Operator type	Operator symbols	Operation performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
	||	OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1
Arithmetic	+	Addition	2
	-	Subtraction	2
	-	2's complement	1
	*	Multiplication	2
	/	Division	2
Relational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal to	2
	<=	Less than or equal to	2
Equality	==	Logical equality	2
	!=	Logical inequality	2
Shift	>>	Right shift	2
	<<	Left shift	2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

To illustrate the results produced by the various operators, we will use three-bit vectors $A[2{:}0]$, $B[2{:}0]$ and $C[2{:}0]$, as well as scalars f and w.

Bitwise Operators

Bitwise operators operate on individual bits of operands. The ~ operator forms the 1's complement of the operand such that the statement

$$C = \sim A;$$

produces the result $c_2 = \overline{a}_2$, $c_1 = \overline{a}_1$, and $c_0 = \overline{a}_0$, where a_i and c_i are the bits of the vectors A and C.

Other bitwise operators operate on pairs of bits. The statement

C = A & B;

generates $c_2 = a_2 \cdot b_2$, $c_1 = a_1 \cdot b_1$, and $c_0 = a_0 \cdot b_0$. Similarly, the | and ^ operators perform bitwise OR and XOR operations. The ^~ operator, which can also be written as ~^, produces the XNOR such that

C = A ~^ B;

gives $c_2 = \overline{a_2 \oplus b_2}$, $c_1 = \overline{a_1 \oplus b_1}$, and $c_0 = \overline{a_0 \oplus b_0}$. If the operands are of unequal size, then the shorter operand is extended by padding 0s to the left.

A scalar function may be assigned a value as a result of a bitwise operation on two vector operands. In this case, it is only the least-significant bits of the operands that are involved in the operation. Hence the statement

f = A ^ B;

yields $f = a_0 \oplus b_0$.

The bitwise operations may involve operands that include the unknown logic value x. Then the operations are performed according to the truth tables in Figure 6.44. For example, if P = 4'b101x and Q = 4'b1001, then P & Q = 4'b100x while P | Q = 4'b1011.

Logical Operators

The ! operator has the same effect on a scalar operand as the ~ operator. Thus, f = !w = ~w. But the effect on a vector operand is different, namely if

f = !A;

then $f = \overline{a_2 + a_1 + a_0}$.

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

~^	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Figure 6.44 Truth tables for bitwise operators.

The && operator implements the AND operation such that

$$f = A \ \&\& \ B;$$

produces $f = (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0)$. Similarly, using the || operator in

$$f = A \ || \ B;$$

gives $f = (a_2 + a_1 + a_0) + (b_2 + b_1 + b_0)$.

Reduction Operators

The reduction operators perform an operation on the bits of a single vector operand and produce a one-bit result. Using the & operator in

$$f = \&A;$$

produces $f = a_2 \cdot a_1 \cdot a_0$. Similarly,

$$f = \hat{}A;$$

gives $f = a_2 \oplus a_1 \oplus a_0$, and so on. As an example of reduction operator use, consider the parity function discussed in section 5.8. The XOR circuit that computes the parity bit, p, of an n-bit vector X can be defined with the statement

$$p = \hat{}X;$$

Arithmetic Operators

We have already encountered the arithmetic operators in Chapter 5. They perform standard arithmetic operations. Thus

$$C = A + B;$$

puts the three-bit sum of A plus B into C, while

$$C = A - B;$$

puts the difference of A and B into C. The operation

$$C = -A;$$

places the 2's complement of A into C.

The addition, subtraction, and multiplication operations are supported by most CAD synthesis tools. However, the division operation is often not supported. When the Verilog compiler encounters an arithmetic operator, it usually synthesizes it by using an appropriate module from a library.

Relational Operators

The relational operators are typically used as conditions in **if-else** and **for** statements. These operators function in the same way as the corresponding operators in the C programming language. An expression that uses the relational operators returns the value 1 if it is evaluated as true, and the value 0 if evaluated as false. If there are any x (unknown) or z bits in the operands, then the expression takes the value x.

```
module compare (A, B, AeqB, AgtB, AltB);
   input [3:0] A, B;
   output reg AeqB, AgtB, AltB;

   always @(A, B)
   begin
      AeqB = 0;
      AgtB = 0;
      AltB = 0;
      if (A == B)
         AeqB = 1;
      else if (A > B)
         AgtB = 1;
      else
         AltB = 1;
   end

endmodule
```

Figure 6.45 Verilog code for a four-bit comparator.

Example 6.24 The use of relational operators in the **if-else** statement is illustrated in Figure 6.45. The defined circuit is the four-bit comparator described in section 6.5.

Equality Operators

The expression (A == B) is evaluated as true if *A* is equal to *B* and false otherwise. The != operator has the opposite effect. The result is ambiguous (*x*) if either operand contains *x* or *z* values.

Shift Operators

A vector operand can be shifted to the right or left by a number of bits specified as a constant. When bits are shifted, the vacant bit positions are filled with 0s. For example,

$$\mathrm{B} = \mathrm{A} << 1;$$

results in $b_2 = a_1$, $b_1 = a_0$, and $b_0 = 0$. Similarly,

$$\mathrm{B} = \mathrm{A} >> 2;$$

yields $b_2 = b_1 = 0$ and $b_0 = a_2$.

Concatenate Operator

This operator concatenates two or more vectors to create a larger vector. For example,

$$\mathrm{D} = \{\mathrm{A}, \mathrm{B}\};$$

defines the six-bit vector $D = a_2a_1a_0b_2b_1b_0$. Similarly, the concatenation

E = {3'b111, A, 2'b00};

produces the eight-bit vector $E = 111a_2a_1a_000$.

Replication Operator

This operator allows repetitive concatenation of the same vector, which is replicated the number of times indicated in the replication constant. For example, {3{A}} is equivalent to writing {A, A, A}. The specification {4{2'b10}} produces the eight-bit vector 10101010.

The replication operator may be used in conjunction with the concatenate operator. For instance, {2{A}, 3{B}} is equivalent to {A, A, B, B, B}. We introduced the concatenate and replication operators in section 5.5.6 and illustrated their use in specifying the adder circuits.

Conditional Operator

The conditional operator is discussed fully in section 6.6.1.

Operator Precedence

The Verilog operators are assumed to have the precedence indicated in Table 6.3. The order of precedence is from top to bottom; operators in the top row have the highest precedence and those in the bottom row have the lowest precedence. The operators listed in the same row have the same precedence.

Table 6.3 Precedence of Verilog operators.

Operator type	Operator symbols	Precedence
Complement	! ~ −	Highest procedence
Arithmetic	∗ / + −	
Shift	<< >>	
Relational	< <= > >=	
Equality	== !=	
Reduction	& ~& ^ ~^ | ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

The designer can use parentheses to change the precedence of operators in Verilog code or remove any possible misinterpretation. It is a good practice to use parentheses to make the code unambiguous and easy to read.

6.6.6 THE GENERATE CONSTRUCT

In section 5.5.4 we introduced the **generate** loop capability which can be used to create multiple instances of subcircuits. A subcircuit may be defined in a block of statements delineated by the **generate** and **endgenerate** keywords. The subcircuit is instantiated multiple times using a generate-index variable. This variable is defined using the **genvar** keyword and it can have only positive integer values. It is not possible to use an index declared as a normal **integer** variable.

Example 6.25 Figure 6.46 shows how the **generate** construct can be used to specify an *n*-bit ripple-carry adder. The subcircuit is a full-adder defined structurally in terms of primitive gates as introduced in Figure 5.22. The **for** loop causes the full-adder block to be instantiated *n* times.

In this example, the **for** statement is used in the **generate** block to control the selection of the generated objects. The **generate** block can also contain **if-else** and **case** statements to determine which objects are generated.

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n–1:0] X, Y;
   output [n–1:0] S;
   output carryout;
   wire [n:0] C;

   genvar k;
   assign C[0] = carryin;
   assign carryout = C[n];
   generate
      for (k = 0; k < n; k = k+1)
      begin: fulladd_stage
         wire z1, z2, z3;   //wires within full-adder
         xor (S[k], X[k], Y[k], C[k]);
         and (z1, X[k], Y[k]);
         and (z2, X[k], C[k]);
         and (z3, Y[k], C[k]);
         or (C[k+1], z1, z2, z3);
      end
   endgenerate

endmodule
```

Figure 6.46 Using the **generate** loop to define an *n*-bit ripple-carry adder.

6.6.7 Tasks and Functions

In high-level programming languages it is possible to use subroutines and functions to avoid replicating specific routines that may be needed in several places of a given program. Verilog provides similar capabilities, known as tasks and functions. They can be used to modularize large designs and make the Verilog code easier to understand.

Verilog Task

A task is declared by the keyword **task** and it comprises a block of statements that ends with the keyword **endtask**. The task must be included in the module that calls it. It may have input and output ports. These are not the ports of the module that contains the task, which are used to make external connections to the module. The task ports are used only to pass values between the module and the task.

Example 6.26

In Figure 6.33 we showed the Verilog code for a 16-to-1 multiplexer that instantiates five copies of a 4-to-1 multiplexer circuit given in a separate module named *mux4to1*. The same circuit can be specified using the task approach as shown in Figure 6.47. Observe the key differences. The task *mux4to1* is included in the module *mux16to1*. It is called from an **always** block by means of an appropriate **case** statement. The output of a task must be a variable, hence *g* is of **reg** type.

Verilog Function

A function is declared by the keyword **function** and it comprises a block of statements that ends with the keyword **endfunction**. The function must have at least one input and it returns a single value that is placed where the function is invoked.

Example 6.27

Figure 6.48 shows how the code in Figure 6.47 can be written to use a function. The Verilog compiler essentially inserts the body of the function at each place where it is called. Hence the clause

```
0: f = mux4to1 (W[0:3], S16[1:0]);
```

becomes

```
0: case (S16[1:0])
      0: f = W[0];
      1: f = W[1];
      2: f = W[2];
      3: f = W[3];
   endcase
```

```
module mux16to1 (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output reg f;

  always @(W, S16)
    case (S16[3:2])
      0: mux4to1 (W[0:3], S16[1:0], f);
      1: mux4to1 (W[4:7], S16[1:0], f);
      2: mux4to1 (W[8:11], S16[1:0], f);
      3: mux4to1 (W[12:15], S16[1:0], f);
    endcase

  // Task that specifies a 4-to-1 multiplexer
  task mux4to1;
    input [0:3] X;
    input [1:0] S4;
    output reg g;

    case (S4)
      0: g = X[0];
      1: g = X[1];
      2: g = X[2];
      3: g = X[3];
    endcase
  endtask

endmodule
```

Figure 6.47 Use of a task in Verilog code.

The function serves as a convenience that makes the mux16to1 module compact and easier to read.

A Verilog function can invoke another function but it cannot call a Verilog task. A task may call another task and it may invoke a function. In Figure 6.47 we defined the task after the **always** block that calls it. In contrast, in Figure 6.48 we defined the function before the **always** block that invokes it. Both possibilities are allowed in the Verilog standard for both tasks and functions. However, some tools require functions to be defined before the statements that invoke them.

```
module mux16to1 (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output reg f;

  // Function that specifies a 4-to-1 multiplexer
  function mux4to1;
    input [0:3] X;
    input [1:0] S4;

    case (S4)
      0: mux4to1 = X[0];
      1: mux4to1 = X[1];
      2: mux4to1 = X[2];
      3: mux4to1 = X[3];
    endcase
  endfunction

  always @(W, S16)
    case (S16[3:2])
      0: f = mux4to1 (W[0:3], S16[1:0]);
      1: f = mux4to1 (W[4:7], S16[1:0]);
      2: f = mux4to1 (W[8:11], S16[1:0]);
      3: f = mux4to1 (W[12:15], S16[1:0]);
    endcase

endmodule
```

Figure 6.48 The code from Figure 6.47 using a function.

6.7 CONCLUDING REMARKS

This chapter has introduced a number of circuit building blocks. Examples using these blocks to construct larger circuits will be presented in Chapters 7 and 10. To describe the building block circuits efficiently, several Verilog constructs have been introduced. In many cases a given circuit can be described in various ways, using different constructs. A circuit that can be described using an **if-else** statement can also be described using a **case** statement or perhaps a **for** loop. In general, there are no strict rules that dictate when one style should be preferred over another. With experience the user develops a sense for which types of statements work well in a particular design situation. Personal preference also influences how the code is written.

Verilog is not a programming language, and Verilog code should not be written as if it were a computer program. The statements discussed in this chapter can be used to create large, complex circuits. A good way to design such circuits is to construct them using well-defined modules, in the manner that we illustrated for the multiplexers, decoders, encoders, and so on. Additional examples using the Verilog statements introduced in this chapter are given in Chapters 7 and 8. In Chapter 10 we provide a number of examples of using Verilog code to describe larger digital systems. For more information on Verilog, the reader can consult more specialized books [5–11].

In the next chapter we introduce logic circuits that include the ability to store logic signal values in memory elements.

6.8 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example 6.28 **Problem:** Implement the function $f(w_1, w_2, w_3) = \sum m(0, 1, 3, 4, 6, 7)$ by using a 3-to-8 binary decoder and an OR gate.

Solution: The decoder generates a separate output for each minterm of the required function. These outputs are then combined in the OR gate, giving the circuit in Figure 6.49.

Example 6.29 **Problem:** Derive a circuit that implements an 8-to-3 binary encoder.

Solution: The truth table for the encoder is shown in Figure 6.50. Only those rows for which a single input variable is equal to 1 are shown; the other rows can be treated as don't care cases. From the truth table it is seen that the desired circuit is defined by the equations

$$y_2 = w_4 + w_5 + w_6 + w_7$$
$$y_1 = w_2 + w_3 + w_6 + w_7$$
$$y_0 = w_1 + w_3 + w_5 + w_7$$

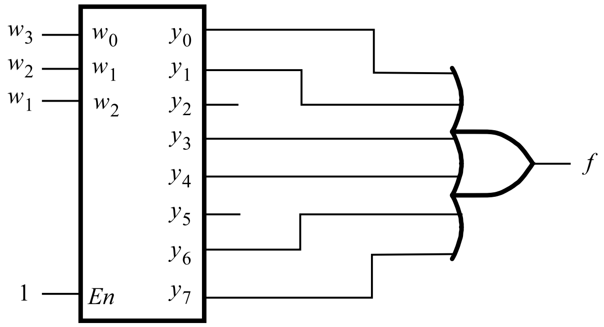


Figure 6.49 Circuit for Example 6.28.

w_7	w_6	w_5	w_4	w_3	w_2	w_1	w_0	y_2	y_1	y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Figure 6.50 Truth table for an 8-to-3 binary encoder.

Example 6.30

Problem: Implement the function

$$f(w_1, w_2, w_3, w_4) = \overline{w}_1\overline{w}_2\overline{w}_4\overline{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5$$

by using a 4-to-1 multiplexer and as few other gates as possible. Assume that only the uncomplemented inputs w_1, w_2, w_3, and w_4 are available.

Solution: Since variables w_1 and w_4 appear in more product terms in the expression for f than the other three variables, let us perform Shannon's expansion with respect to these two variables. The expansion gives

$$\begin{aligned} f &= \overline{w}_1\overline{w}_4 f_{\overline{w}_1\overline{w}_4} + \overline{w}_1 w_4 f_{\overline{w}_1 w_4} + w_1\overline{w}_4 f_{w_1\overline{w}_4} + w_1w_4 f_{w_1w_4} \\ &= \overline{w}_1\overline{w}_4(\overline{w}_2\overline{w}_5) + \overline{w}_1w_4(w_3w_5) + w_1\overline{w}_4(w_2 + w_3) + w_1w_2(1) \end{aligned}$$

We can use a NOR gate to implement $\overline{w}_2\overline{w}_5 = \overline{w_2 + w_5}$. We also need an AND gate and an OR gate. The complete circuit is presented in Figure 6.51.

Example 6.31

Problem: In Chapter 4 we pointed out that the rows and columns of a Karnaugh map are labeled using Gray code. This is a code in which consecutive valuations differ in one variable only. Figure 6.52 depicts the conversion between three-bit binary and Gray codes. Design a circuit that can convert a binary code into Gray code according to the figure.

Solution: From the figure it follows that

$$\begin{aligned} g_2 &= b_2 \\ g_1 &= b_1\overline{b}_2 + \overline{b}_1b_2 \\ &= b_1 \oplus b_2 \\ g_0 &= b_0\overline{b}_1 + \overline{b}_0b_1 \\ &= b_0 \oplus b_1 \end{aligned}$$

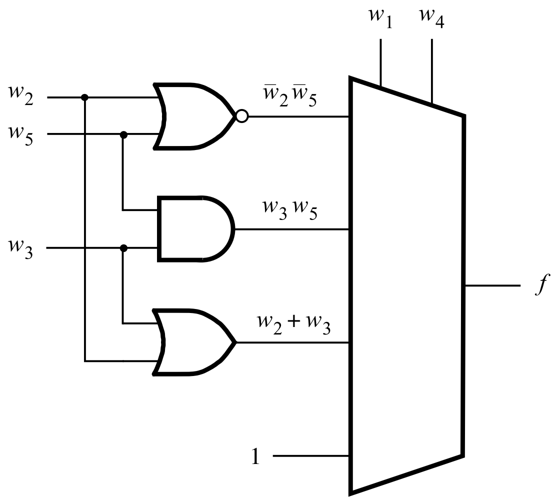


Figure 6.51 Circuit for Example 6.30.

b_2	b_1	b_0	g_2	g_1	g_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

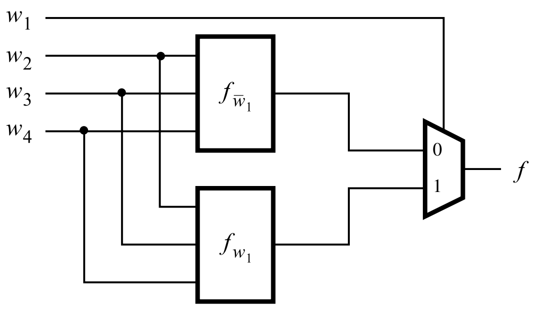
Figure 6.52 Binary to Gray code coversion.

Example 6.32 **Problem:** In section 6.1.2 we showed that any logic function can be decomposed using Shannon's expansion theorem. For a four-variable function, $f(w_1, \ldots, w_4)$, the expansion with respect to w_1 is

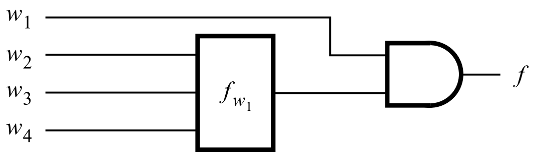
$$f(w_1, \ldots, w_4) = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

A circuit that implements this expression is given in Figure 6.53*a*.
(a) If the decomposition yields $f_{\overline{w}_1} = 0$, then the multiplexer in the figure can be replaced by a single logic gate. Show this circuit.
(b) Repeat part (*a*) for the case where $f_{w_1} = 1$.

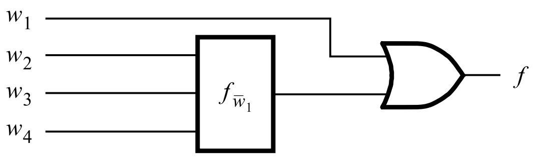
Solution: The desired circuits are shown in parts (*b*) and (*c*) of Figure 6.53.



(a) Shannon's expansion of the function *f*.



(b) Solution for part *a*



(c) Solution for part *b*

Figure 6.53 Circuits for Example 6.32.

Example 6.33

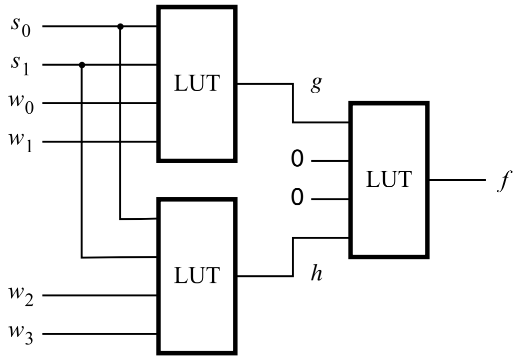
Problem: In several commercial FPGAs the logic blocks are 4-LUTs. What is the minimum number of 4-LUTs needed to construct a 4-to-1 multiplexer with select inputs s_1 and s_0 and data inputs w_3, w_2, w_1, and w_0?

Solution: A straightforward attempt is to use directly the expression that defines the 4-to-1 multiplexer

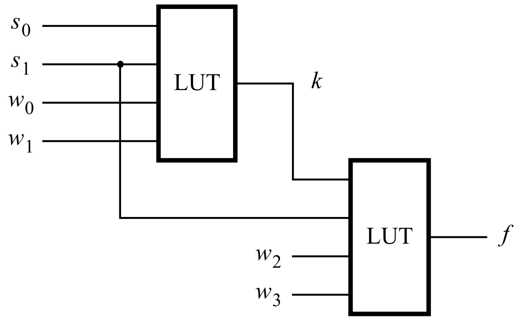
$$f = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1\overline{s}_0w_2 + s_1s_0w_3$$

Let $g = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1$ and $h = s_1\overline{s}_0w_2 + s_1s_0w_3$, so that $f = g + h$. This decomposition leads to the circuit in Figure 6.54*a*, which requires three LUTs.

When designing logic circuits, one can sometimes come up with a clever idea which leads to a superior implementation. Figure 6.54*b* shows how it is possible to implement



(a) Using three LUTs



(b) Using two LUTs

Figure 6.54 Circuits for Example 6.33.

the multiplexer with just two LUTs, based on the following observation. The truth table in Figure 6.2*b* indicates that when $s_1 = 0$ the output must be either w_0 or w_1, as determined by the value of s_0. This can be generated by the first LUT. The second LUT must make the choice between w_2 and w_3 when $s_1 = 1$. But, the choice can be made only by knowing the value of s_0. Since it is impossible to have five inputs in the LUT, more information has to be passed from the first to the second LUT. Observe that when $s_1 = 1$ the output f will be equal to either w_2 or w_3, in which case it is not necessary to know the values of w_0 and w_1. Hence, in this case we can pass on the value of s_0 through the first LUT, rather than w_0 or w_1. This can be done by making the function of this LUT

$$k = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1s_0$$

Then, the second LUT performs the function

$$f = \overline{s}_1k + s_1\overline{k}w_3 + s_1kw_4$$

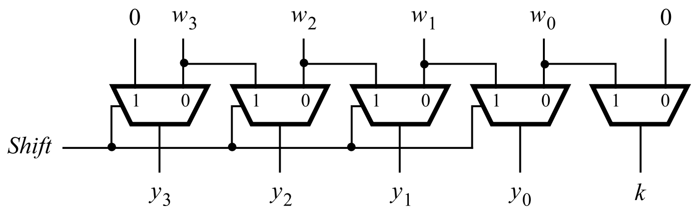


Figure 6.55 A shifter circuit.

Example 6.34

Problem: In digital systems it is often necessary to have circuits that can shift the bits of a vector by one or more bit positions to the left or right. Design a circuit that can shift a four-bit vector $W = w_3w_2w_1w_0$ one bit position to the right when a control signal *Shift* is equal to 1. Let the outputs of the circuit be a four-bit vector $Y = y_3y_2y_1y_0$ and a signal k, such that if $Shift = 1$ then $y_3 = 0$, $y_2 = w_3$, $y_1 = w_2$, $y_0 = w_1$, and $k = w_0$. If $Shift = 0$ then $Y = W$ and $k = 0$.

Solution: The required circuit can be implemented with five 2-to-1 multiplexers as shown in Figure 6.55. The *Shift* signal is used as the select input to each multiplexer.

Example 6.35

Problem: The shifter circuit in Example 6.34 shifts the bits of an input vector by one bit position to the right. It fills the vacated bit on the left side with 0. A more versatile shifter circuit may be able to shift by more bit positions at a time. If the bits that are shifted out are placed into the vacated positions on the left, then the circuit effectively rotates the bits of the input vector by a specified number of bit positions. Such a circuit is often called a *barrel shifter*. Design a four-bit barrel shifter that rotates the bits by 0, 1, 2, or 3 bit positions as determined by the valuation of two control signals s_1 and s_0.

Solution: The required action is given in Figure 6.56*a*. The barrel shifter can be implemented with four 4-to-1 multiplexers as shown in Figure 6.56*b*. The control signals s_1 and s_0 are used as the select inputs to the multiplexers.

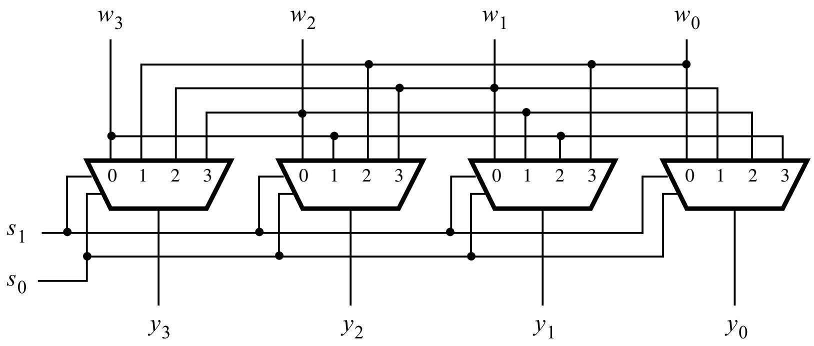
Example 6.36

Problem: Write Verilog code that represents the circuit in Figure 6.19. Use the *dec2to4* module in Figure 6.35 as a subcircuit in your code.

Solution: The code is shown in Figure 6.57. Note that the *dec2to4* module can be included in the same file as we have done in the figure, but it can also be in a separate file in the project directory.

s_1	s_0	y_3	y_2	y_1	y_0
0	0	w_3	w_2	w_1	w_0
0	1	w_0	w_3	w_2	w_1
1	0	w_1	w_0	w_3	w_2
1	1	w_2	w_1	w_0	w_3

(a) Truth table



(b) Circuit

Figure 6.56 A barrel shifter circuit.

Example 6.37 **Problem:** Write Verilog code that represents the shifter circuit in Figure 6.55.

Solution: One possibility is to specify the structure of this circuit as shown in Figure 6.58. The **if-else** construct is used to define the desired shifting of individual bits. A typical Verilog compiler will implement this code with 2-to-1 multiplexers as depicted in Figure 6.55.

An alternative is to make use of the shift operator defined in section 6.6.5, as indicated in Figure 6.59.

Example 6.38 **Problem:** Write Verilog code that defines the barrel shifter in Figure 6.56.

Solution: The code in Figure 6.60 is a possible solution. The rotate function is accomplished by concatenating two copies of the input vector W and shifting the obtained 8-bit vector to the right by the number of bit positions specified as the input S. The four least-significant bits of the resulting 8-bit vector are the desired output Y.

```
module mux4to1 (W, S, f);
   input [0:3] W;
   input [1:0] S;
   output f;
   wire  [0:3] Y;

   dec2to4 decoder (S, Y, 1);
   assign f = |(W & Y);

endmodule

module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;

   always @(W, En)
      case ({En, W})
         3'b100: Y = 4'b1000;
         3'b101: Y = 4'b0100;
         3'b110: Y = 4'b0010;
         3'b111: Y = 4'b0001;
         default: Y = 4'b0000;
      endcase

endmodule
```

Figure 6.57 Verilog code for Example 6.36.

Problems

Answers to problems marked by an asterisk are given at the back of the book.

6.1 Show how the function $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 4, 5, 7)$ can be implemented using a 3-to-8 binary decoder and an OR gate.

6.2 Show how the function $f(w_1, w_2, w_3) = \sum m(1, 2, 3, 5, 6)$ can be implemented using a 3-to-8 binary decoder and an OR gate.

***6.3** Consider the function $f = \overline{w}_1\overline{w}_3 + w_2\overline{w}_3 + \overline{w}_1 w_2$. Use the truth table to derive a circuit for f that uses a 2-to-1 multiplexer.

6.4 Repeat problem 6.3 for the function $f = \overline{w}_2\overline{w}_3 + w_1 w_2$.

***6.5** For the function $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 6)$, use Shannon's expansion to derive an implementation using a 2-to-1 multiplexer and any other necessary gates.

```
module shifter (W, Shift, Y, k);
    input [3:0] W;
    input Shift;
    output reg [3:0] Y;
    output reg k;

    always @(W, Shift)
    begin
        if (Shift)
        begin
            Y[3] = 0;
            Y[2:0] = W[3:1];
            k = W[0];
        end
        else
        begin
            Y = W;
            k = 0;
        end
    end

endmodule
```

Figure 6.58 Verilog code for the circuit in Figure 6.55.

6.6 Repeat problem 6.5 for the function $f(w_1, w_2, w_3) = \sum m(0, 4, 6, 7)$.

6.7 Consider the function $f = \overline{w}_2 + \overline{w}_1\overline{w}_3 + w_1 w_3$. Show how repeated application of Shannon's expansion can be used to derive the minterms of f.

6.8 Repeat problem 6.7 for $f = w_2 + \overline{w}_1\overline{w}_3$.

6.9 Prove Shannon's expansion theorem presented in section 6.1.2.

***6.10** Section 6.1.2 shows Shannon's expansion in sum-of-products form. Using the principle of duality, derive the equivalent expression in product-of-sums form.

6.11 Consider the function $f = \overline{w}_1\overline{w}_2 + \overline{w}_2\overline{w}_3 + w_1 w_2 w_3$. Give a circuit that implements f using the minimal number of two-input LUTs. Show the truth table implemented inside each LUT.

***6.12** For the function in problem 6.11, the cost of the minimal sum-of-products expression is 14, which includes four gates and 10 inputs to the gates. Use Shannon's expansion to derive a multilevel circuit that has a lower cost and give the cost of your circuit.

6.13 Consider the function $f(w_1, w_2, w_3, w_4) = \sum m(0, 1, 3, 6, 8, 9, 14, 15)$. Derive an implementation using the minimum possible number of three-input LUTs.

***6.14** Give two examples of logic functions with five inputs, $w_1, \ldots, w_5$, that can be realized using 2 four-input LUTs.

```
module shifter (W, Shift, Y, k);
   input [3:0] W;
   input Shift;
   output reg [3:0] Y;
   output reg k;

   always @(W, Shift)
   begin
      if (Shift)
      begin
         Y = W >> 1;
         k = W[0];
      end
      else
      begin
         Y = W;
         k = 0;
      end
   end

endmodule
```

Figure 6.59 Alternative Verilog code for the circuit in Figure 6.55.

```
module barrel (W, S, Y);
   input [3:0] W;
   input [1:0] S;
   output [3:0] Y;
   wire [3:0] T;

   assign {T, Y} = {W, W} >> S;

endmodule
```

Figure 6.60 Verilog code for the barrel shifter.

6.15 For the function, f, in Example 6.30 perform Shannon's expansion with respect to variables w_1 and w_2, rather than w_1 and w_4. How does the resulting circuit compare with the circuit in Figure 6.51?

6.16 Actel Corporation manufactures an FPGA family called Act 1, which has the multiplexer-based logic block illustrated in Figure P6.1. Show how the function $f = w_2\overline{w}_3 + w_1w_3 + \overline{w}_2w_3$ can be implemented using only one Act 1 logic block.

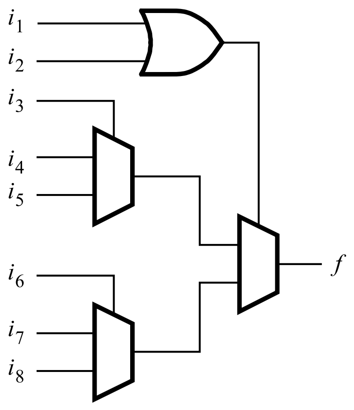


Figure P6.1 The Actel Act 1 logic block.

6.17 Show how the function $f = w_1\overline{w}_3 + \overline{w}_1 w_3 + w_2\overline{w}_3 + w_1\overline{w}_2$ can be realized using Act 1 logic blocks. Note that there are no NOT gates in the chip; hence complements of signals have to be generated using the multiplexers in the logic block.

***6.18** Consider the Verilog code in Figure P6.2. What type of circuit does the code represent? Comment on whether or not the style of code used is a good choice for the circuit that it represents.

```
module problem6_18 (W, En, y0, y1, y2, y3);
   input [1:0] W;
   input En;
   output reg y0, y1, y2, y3;

   always @(W, En)
   begin
      y0 = 0;
      y1 = 0;
      y2 = 0;
      y3 = 0;
      if (En)
         if (W == 0) y0 = 1;
         else if (W == 1) y1 = 1;
         else if (W == 2) y2 = 1;
         else y3 = 1;
   end

endmodule
```

Figure P6.2 Code for problem 6.18.

6.19 Write Verilog code that represents the function in problem 6.2, using a **case** statement.

6.20 Write Verilog code for a 4-to-2 binary encoder.

6.21 Write Verilog code for an 8-to-3 binary encoder.

6.22 Figure P6.3 shows a modified version of the code for a 2-to-4 decoder in Figure 6.42. This code is almost correct but contains one error. What is the error?

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;
   integer k;

   always @(W, En)
      for (k = 0; k < = 3; k = k+1)
         if (W == k)
            Y[k] = En;

endmodule
```

Figure P6.3 Code for problem 6.22.

6.23 Derive the circuit for an 8-to-3 priority encoder.

6.24 Using a **casex** statement, write Verilog code for an 8-to-3 priority encoder.

6.25 Repeat problem 6.24, using a **for** loop.

6.26 Create a Verilog module named *if2to4* that represents a 2-to-4 binary decoder using an **if-else** statement. Create a second module named *h3to8* that represents the 3-to-8 binary decoder in Figure 6.17 using two instances of the *if2to4* module.

6.27 Create a Verilog module named *h6to64* that represents a 6-to-64 binary decoder. Use the treelike structure in Figure 6.18, in which the 6-to-64 decoder is built using nine instances of the *h3to8* decoder created in problem 6.26.

6.28 Write Verilog code that represents the circuit in Figure 6.19. Use the *dec2to4* module in Figure 6.35 as a subcircuit in your code.

***6.29** Derive minimal sum-of-products expressions for the outputs a, b, and c of the 7-segment display in Figure 6.25.

6.30 Derive minimal sum-of-products expressions for the outputs d, e, f, and g of the 7-segment display in Figure 6.25.

6.31 Figure 6.21 shows a block diagram of a ROM. A circuit that implements a small ROM, with four rows and four columns, is depicted in Figure P6.4. Each X in the figure represents a switch that determines whether the ROM produces a 1 or 0 when that location is read.

(a) Show how a switch (X) can be realized using a single NMOS transistor.
(b) Draw the complete 4×4 ROM circuit, using your switches from part (*a*). The ROM should be programmed to store the bits 0101 in row 0 (the top row), 1010 in row 1, 1100 in row 2, and 0011 in row 3 (the bottom row).
(c) Show how each (X) can be implemented as a programmable switch (as opposed to providing either a 1 or 0 permanently), using an EEPROM cell as shown in Figure 3.64. Briefly describe how the storage cell is used.

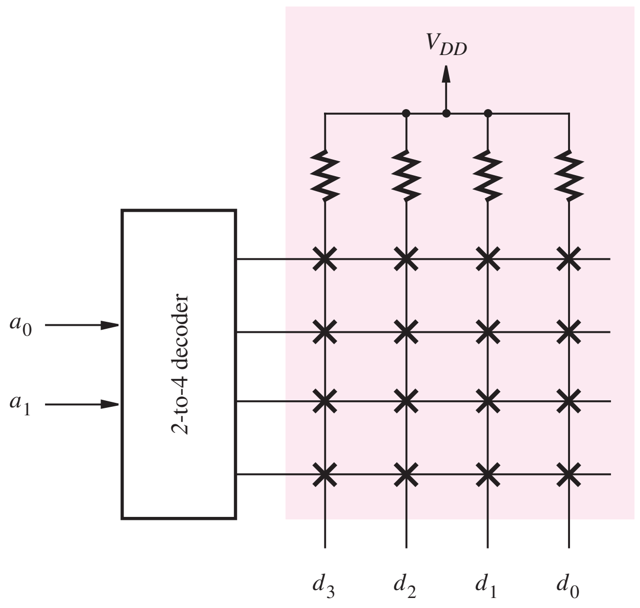


Figure P6.4 A 4 × 4 ROM circuit.

6.32 Show the complete circuit for a ROM using the storage cells designed in Part (*a*) of problem 6.31 that realizes the logic functions

$$d_3 = a_0 \oplus a_1$$

$$d_2 = \overline{a_0 \oplus a_1}$$

$$d_1 = a_0 a_1$$

$$d_0 = a_0 + a_1$$

References

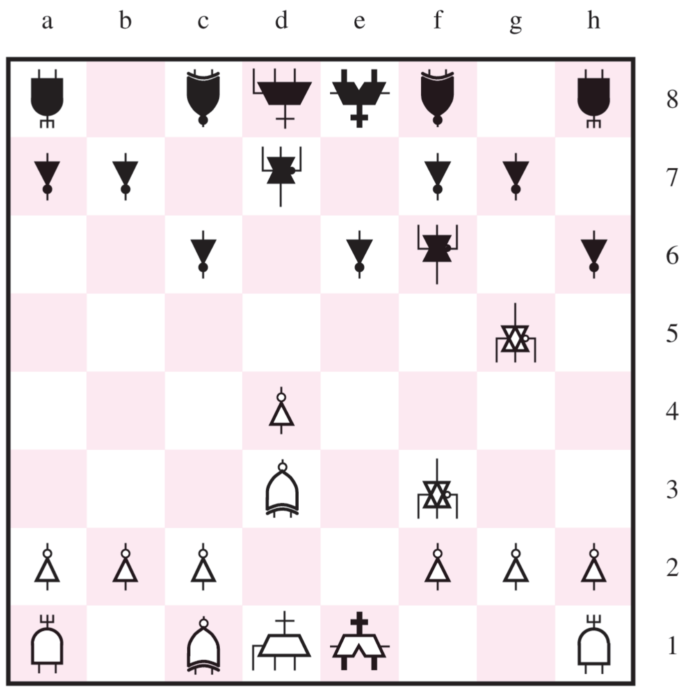
1. C. E. Shannon, "Symbolic Analysis of Relay and Switching Circuits," *Transactions AIEE* 57 (1938), pp. 713–723.

2. Actel Corporation, “MX FPGA Data Sheet,” *http://www.actel.com.*
3. QuickLogic Corporation, “pASIC 3 FPGA Data Sheet,” *http://www.quicklogic.com.*
4. R. Landers, S. Mahant-Shetti, and C. Lemonds, “A Multiplexer-Based Architecture for High-Density, Low Power Gate Arrays,” *IEEE Journal of Solid-State Circuits* 30, no. 4 (April 1995).
5. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
6. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
7. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed., (Prentice-Hall: Upper Saddle River, NJ, 2003).
8. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
9. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
10. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
11. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

chapter

7

Flip-Flops, Registers, Counters, and a Simple Processor



7. Ng1–f3, h7–h6

In previous chapters we considered combinational circuits where the value of each output depends solely on the values of signals applied to the inputs. There exists another class of logic circuits in which the values of the outputs depend not only on the present values of the inputs but also on the past behavior of the circuit. Such circuits include storage elements that store the values of logic signals. The contents of the storage elements are said to represent the *state* of the circuit. When the circuit's inputs change values, the new input values either leave the circuit in the same state or cause it to change into a new state. Over time the circuit changes through a sequence of states as a result of changes in the inputs. Circuits that behave in this way are referred to as *sequential circuits*.

In this chapter we will introduce circuits that can be used as storage elements. But first, we will motivate the need for such circuits by means of a simple example. Suppose that we wish to control an alarm system, as shown in Figure 7.1. The alarm mechanism responds to the control input $On/\overline{Off}$. It is turned on when $On/\overline{Off} = 1$, and it is off when $On/\overline{Off} = 0$. The desired operation is that the alarm turns on when the sensor generates a positive voltage signal, *Set*, in response to some undesirable event. Once the alarm is triggered, it must remain active even if the sensor output goes back to zero. The alarm is turned off manually by means of a *Reset* input. The circuit requires a memory element to remember that the alarm has to be active until the *Reset* signal arrives.

Figure 7.2 gives a rudimentary memory element, consisting of a loop that has two inverters. If we assume that $A = 0$, then $B = 1$. The circuit will maintain these values indefinitely. We say that the circuit is in the *state* defined by these values. If we assume that $A = 1$, then $B = 0$, and the circuit will remain in this second state indefinitely. Thus the circuit has two possible states. This circuit is not useful, because it lacks some practical means for changing its state.

A more useful circuit is shown in Figure 7.3. It includes a mechanism for changing the state of the circuit in Figure 7.2, using two transmission gates of the type discussed in section 3.9. One transmission gate, *TG*1, is used to connect the *Data* input terminal to point

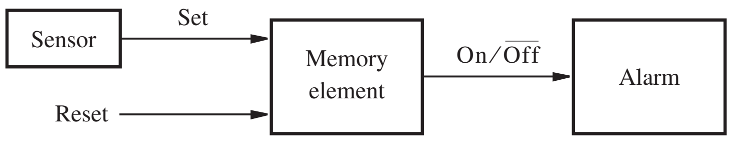


Figure 7.1 Control of an alarm system.

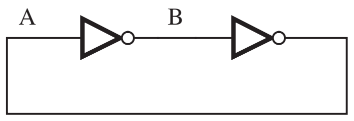


Figure 7.2 A simple memory element.

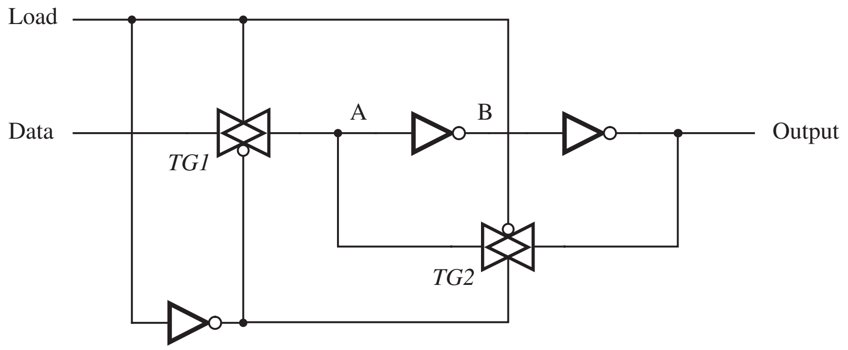


Figure 7.3 A controlled memory element.

A in the circuit. The second, *TG*2, is used as a switch in the *feedback loop* that maintains the state of the circuit. The transmission gates are controlled by the *Load* signal. If *Load* = 1, then *TG*1 is on and the point *A* will have the same value as the *Data* input. Since the value presently stored at *Output* may not be the same value as *Data*, the feedback loop is broken by having *TG*2 turned off when *Load* = 1. When *Load* changes to zero, then *TG*1 turns off and *TG*2 turns on. The feedback path is closed and the memory element will retain its state as long as *Load* = 0. This memory element cannot be applied directly to the system in Figure 7.1, but it is useful for many other applications, as we will see later.

7.1 BASIC LATCH

Instead of using the transmission gates, we can construct a similar circuit using ordinary logic gates. Figure 7.4 presents a memory element built with NOR gates. Its inputs, *Set* and *Reset*, provide the means for changing the state, Q, of the circuit. A more usual way of drawing this circuit is given in Figure 7.5*a*, where the two NOR gates are said to be connected in cross-coupled style. The circuit is referred to as a *basic latch*. Its behavior is described by the truth table in Figure 7.5*b*. When both inputs, *R* and *S*, are equal to 0 the latch maintains its existing state. This state may be either $Q_a = 0$ and $Q_b = 1$, or $Q_a = 1$ and $Q_b = 0$, which is indicated in the truth table by stating that the Q_a and Q_b outputs have values 0/1 and 1/0, respectively. Observe that Q_a and Q_b are complements of each other in this case. When $R = 0$ and $S = 1$, the latch is *set* into a state where $Q_a = 1$ and $Q_b = 0$. When $R = 1$ and $S = 0$, the latch is *reset* into a state where $Q_a = 0$ and $Q_b = 1$. The fourth possibility is to have $R = S = 1$. In this case both Q_a and Q_b will be 0.

Figure 7.5*c* gives a timing diagram for the latch, assuming that the propagation delay through the NOR gates is negligible. Of course, in a real circuit the changes in the waveforms would be delayed according to the propagation delays of the gates. We assume that initially $Q_a = 0$ and $Q_b = 1$. The state of the latch remains unchanged until time t_2,

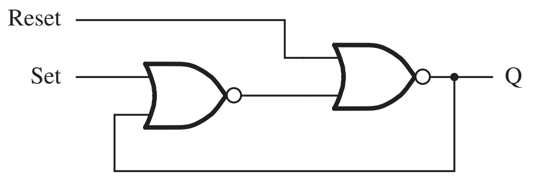
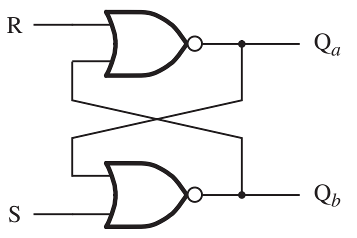


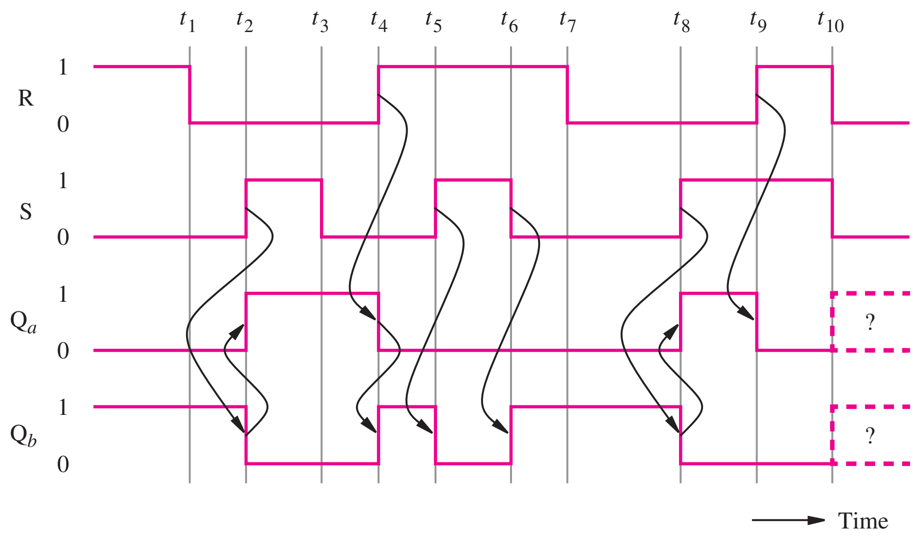
Figure 7.4 A memory element with NOR gates.



(a) Circuit

S	R	Q_a	Q_b	
0	0	0/1	1/0	(No change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Truth table



(c) Timing diagram

Figure 7.5 A basic latch built with NOR gates.

when S becomes equal to 1, causing Q_b to change to 0, which in turn causes Q_a to change to 1. The causality relationship is indicated by the arrows in the diagram. When S goes to 0 at t_3, there is no change in the state because both S and R are then equal to 0. At t_4 we have $R = 1$, which causes Q_a to go to 0, which in turn causes Q_b to go to 1. At t_5 both S and R are equal to 1, which forces both Q_a and Q_b to be equal to 0. As soon as S returns to 0, at t_6, Q_b becomes equal to 1 again. At t_8 we have $S = 1$ and $R = 0$, which causes $Q_b = 0$ and $Q_a = 1$. An interesting situation occurs at t_{10}. From t_9 to t_{10} we have $Q_a = Q_b = 0$ because $R = S = 1$. Now if both R and S change to 0 at t_{10}, both Q_a and Q_b will go to 1. But having both Q_a and Q_b equal to 1 will immediately force $Q_a = Q_b = 0$. There will be an oscillation between $Q_a = Q_b = 0$ and $Q_a = Q_b = 1$. If the delays through the two NOR gates are exactly the same, the oscillation will continue indefinitely. In a real circuit there will invariably be some difference in the delays through these gates, and the latch will eventually settle into one of its two stable states, but we don't know which state it will be. This uncertainty is indicated in the waveforms by dashed lines.

The oscillations discussed above illustrate that even though the basic latch is a simple circuit, careful analysis has to be done to fully appreciate its behavior. In general, any circuit that contains one or more feedback paths, such that the state of the circuit depends on the propagation delays through logic gates, has to be designed carefully. We discuss timing issues in detail in Chapter 9.

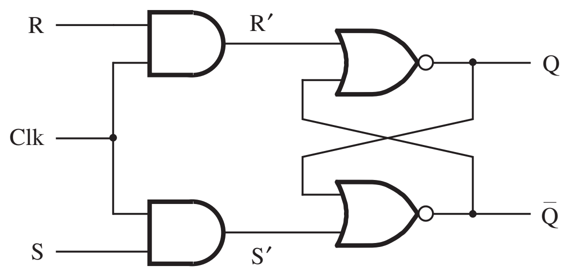
The latch in Figure 7.5*a* can perform the functions needed for the memory element in Figure 7.1, by connecting the *Set* signal to the S input and *Reset* to the R input. The Q_a output provides the desired $On/\overline{Off}$ signal. To initialize the operation of the alarm system, the latch is reset. Thus the alarm is off. When the sensor generates the logic value 1, the latch is set and Q_a becomes equal to 1. This turns on the alarm mechanism. If the sensor output returns to 0, the latch retains its state where $Q_a = 1$; hence the alarm remains turned on. The only way to turn off the alarm is by resetting the latch, which is accomplished by making the *Reset* input equal to 1.

7.2 Gated SR Latch

In section 7.1 we saw that the basic SR latch can serve as a useful memory element. It remembers its state when both the S and R inputs are 0. It changes its state in response to changes in the signals on these inputs. The state changes occur at the time when the changes in the signals occur. If we cannot control the time of such changes, then we don't know when the latch may change its state.

In the alarm system of Figure 7.1, it may be desirable to be able to enable or disable the entire system by means of a control input, *Enable*. Thus when enabled, the system would function as described above. In the disabled mode, changing the *Set* input from 0 to 1 would not cause the alarm to turn on. The latch in Figure 7.5*a* cannot provide the desired operation. But the latch circuit can be modified to respond to the input signals S and R only when $Enable = 1$. Otherwise, it would maintain its state.

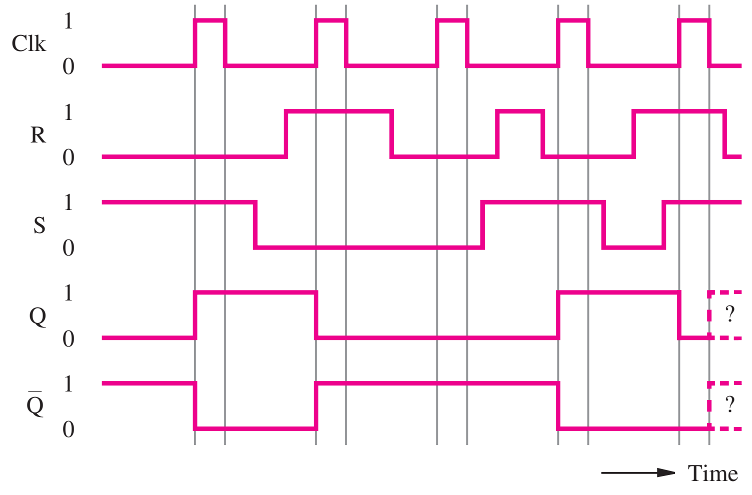
The modified circuit is depicted in Figure 7.6*a*. It includes two AND gates that provide the desired control. When the control signal *Clk* is equal to 0, the S' and R' inputs to the



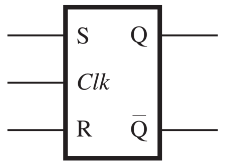
(a) Circuit

Clk	S	R	Q(*t* + 1)
0	x	x	Q(*t*) (No change)
1	0	0	Q(*t*) (No change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Truth table



(c) Timing diagram



(d) Graphical symbol

Figure 7.6 Gated SR latch.

latch will be 0, regardless of the values of signals *S* and *R*. Hence the latch will maintain its existing state as long as $Clk = 0$. When *Clk* changes to 1, the S' and R' signals will be the same as the *S* and *R* signals, respectively. Therefore, in this mode the latch will behave as we described in section 7.1. Note that we have used the name *Clk* for the control signal that allows the latch to be set or reset, rather than call it the *Enable* signal. The reason is that such circuits are often used in digital systems where it is desirable to allow the changes in the states of memory elements to occur only at well-defined time intervals, as if they were controlled by a clock. The control signal that defines these time intervals is usually called the *clock* signal. The name *Clk* is meant to reflect this nature of the signal.

Circuits of this type, which use a control signal, are called *gated latches*. Because our circuit exhibits set and reset capability, it is called a *gated SR latch*. Figure 7.6*b* describes its behavior. It defines the state of the Q output at time $t + 1$, namely, Q($t + 1$), as a function of the inputs S, R, and *Clk*. When $Clk = 0$, the latch will remain in the state it is in at time t, that is, Q(t), regardless of the values of inputs S and R. This is indicated by specifying S = x and R = x, where x means that the signal value can be either 0 or 1. (Recall that we already used this notation in Chapter 4.) When $Clk = 1$, the circuit behaves as the basic latch in Figure 7.5. It is set by $S = 1$ and reset by $R = 1$. The last row of the truth table, where $S = R = 1$, shows that the state Q($t + 1$) is undefined because we don't know whether it will be 0 or 1. This corresponds to the situation described in section 7.1 in conjunction with the timing diagram in Figure 7.5 at time t_{10}. At this time both S and R inputs go from 1 to 0, which causes the oscillatory behavior that we discussed. If $S = R = 1$, this situation will occur as soon as *Clk* goes from 1 to 0. To ensure a meaningful operation of the gated SR latch, it is essential to avoid the possibility of having both the S and R inputs equal to 1 when *Clk* changes from 1 to 0.

A timing diagram for the gated SR latch is given in Figure 7.6*c*. It shows *Clk* as a periodic signal that is equal to 1 at regular time intervals to suggest that this is how the clock signal usually appears in a real system. The diagram presents the effect of several combinations of signal values. Observe that we have labeled one output as Q and the other as its complement $\overline{\text{Q}}$, rather than Q_a and Q_b as in Figure 7.5. Since the undefined mode, where $S = R = 1$, must be avoided in practice, the normal operation of the latch will have the outputs as complements of each other. Moreover, we will often say that the latch is *set* when Q = 1, and it is *reset* when Q = 0. A graphical symbol for the gated SR latch is given in Figure 7.6*d*.

7.2.1 Gated SR Latch with NAND Gates

So far we have implemented the basic latch with cross-coupled NOR gates. We can also construct the latch with NAND gates. Using this approach, we can implement the gated SR latch as depicted in Figure 7.7. The behavior of this circuit is described by the truth table

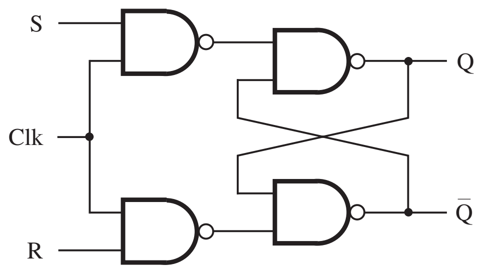


Figure 7.7 Gated SR latch with NAND gates.

in Figure 7.6*b*. Note that in this circuit, the clock is gated by NAND gates, rather than by AND gates. Note also that the *S* and *R* inputs are reversed in comparison with the circuit in Figure 7.6*a*. The circuit with NAND gates requires fewer transistors than the circuit with AND gates. We will use the circuit in Figure 7.7, in preference to the circuit in Figure 7.6*a*.

7.3 GATED D LATCH

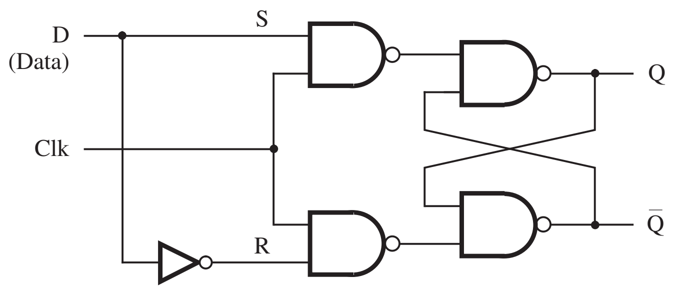
In section 7.2 we presented the gated SR latch and showed how it can be used as the memory element in the alarm system of Figure 7.1. This latch is useful for many other applications. In this section we describe another gated latch that is even more useful in practice. It has a single data input, called *D*, and it stores the value on this input, under the control of a clock signal. It is called a *gated D latch*.

To motivate the need for a gated D latch, consider the adder/subtractor unit discussed in Chapter 5 (Figure 5.13). When we described how that circuit is used to add numbers, we did not discuss what is likely to happen with the sum bits that are produced by the adder. Adder/subtractor units are often used as part of a computer. The result of an addition or subtraction operation is often used as an operand in a subsequent operation. Therefore, it is necessary to be able to remember the values of the sum bits generated by the adder until they are needed again. We might think of using the basic latches to remember these bits, one bit per latch. In this context, instead of saying that a latch remembers the value of a bit, it is more illuminating to say that the latch *stores* the value of the bit or simply "stores the bit." We should think of the latch as a storage element.

But can we obtain the desired operation using the basic latches? We can certainly reset all latches before the addition operation begins. Then we would expect that by connecting a sum bit to the *S* input of a latch, the latch would be set to 1 if the sum bit has the value 1; otherwise, the latch would remain in the 0 state. This would work fine if all sum bits are 0 at the start of the addition operation and, after some propagation delay through the adder, some of these bits become equal to 1 to give the desired sum. Unfortunately, the propagation delays that exist in the adder circuit cause a big problem in this arrangement. Suppose that we use a ripple-carry adder. When the *X* and *Y* inputs are applied to the adder, the sum outputs may alternate between 0 and 1 a number of times as the carries ripple through the circuit. This situation was illustrated in the timing diagram in Figure 5.21. The problem is that if we connect a sum bit to the *S* input of a latch, then if the sum bit is temporarily a 1 and then settles to 0 in the final result, the latch will remain set to 1 erroneously.

The problem caused by the alternating values of the sum bits in the adder could be solved by using the gated SR latches, instead of the basic latches. Then we could arrange that the clock signal is 0 during the time needed by the adder to produce a correct sum. After allowing for the maximum propagation delay in the adder circuit, the clock should go to 1 to store the values of the sum bits in the gated latches. As soon as the values have been stored, the clock can return to 0, which ensures that the stored values will be retained until the next time the clock goes to 1. To achieve the desired operation, we would also have to reset all latches to 0 prior to loading the sum-bit values into these latches. This is an awkward way of dealing with the problem, and it is preferable to use the gated D latches instead.

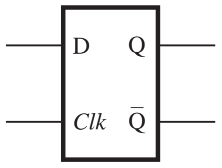
Figure 7.8*a* shows the circuit for a gated D latch. It is based on the gated SR latch, but instead of using the S and R inputs separately, it has just one data input, D. For convenience we have labeled the points in the circuit that are equivalent to the S and R inputs. If $D = 1$, then $S = 1$ and $R = 0$, which forces the latch into the state Q = 1. If $D = 0$, then $S = 0$ and $R = 1$, which causes Q = 0. Of course, the changes in state occur only when $Clk = 1$.



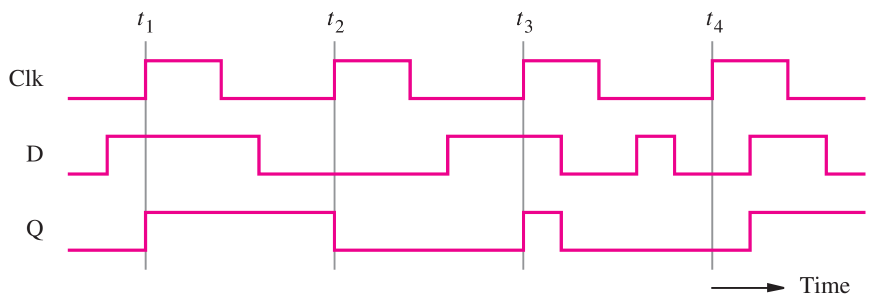
(a) Circuit

Clk	D	Q(*t* + 1)
0	x	Q(*t*)
1	0	0
1	1	1

(b) Truth table



(c) Graphical symbol



(d) Timing diagram

Figure 7.8 Gated D latch.

It is important to observe that in this circuit it is impossible to have the troublesome situation where $S = R = 1$. In the gated D latch, the output Q merely tracks the value of the input D while $Clk = 1$. As soon as Clk goes to 0, the state of the latch is frozen until the next time the clock signal goes to 1. Therefore, the gated D latch stores the value of the D input seen at the time the clock changes from 1 to 0. Figure 7.8 also gives the truth table, the graphical symbol, and the timing diagram for the gated D latch.

The timing diagram illustrates what happens if the D signal changes while $Clk = 1$. During the third clock pulse, starting at t_3, the output Q changes to 1 because $D = 1$. But midway through the pulse D goes to 0, which causes Q to go to 0. This value of Q is stored when Clk changes to 0. Now no further change in the state of the latch occurs until the next clock pulse, at t_4. The key point to observe is that as long as the clock has the value 1, the Q output follows the D input. But when the clock has the value 0, the Q output cannot change. In Chapter 3 we saw that the logic values are implemented as low and high voltage levels. Since the output of the gated D latch is controlled by the level of the clock input, the latch is said to be *level sensitive*. The circuits in Figures 7.6 through 7.8 are level sensitive. We will show in section 7.4 that it is possible to design storage elements for which the output changes only at the point in time when the clock changes from one value to the other. Such circuits are said to be *edge triggered*.

At this point we should reconsider the circuit in Figure 7.3. Careful examination of that circuit shows that it behaves in exactly the same way as the circuit in Figure 7.8*a*. The *Data* and *Load* inputs correspond to the D and Clk inputs, respectively. The *Output*, which has the same signal value as point A, corresponds to the Q output. Point B corresponds to $\overline{\text{Q}}$. Therefore, the circuit in Figure 7.3 is also a gated D latch. An advantage of this circuit is that it can be implemented using fewer transistors than the circuit in Figure 7.8*a*.

7.3.1 EFFECTS OF PROPAGATION DELAYS

In the previous discussion we ignored the effects of propagation delays. In practical circuits it is essential to take these delays into account. Consider the gated D latch in Figure 7.8*a*. It stores the value of the D input that is present at the time the clock signal changes from 1 to 0. It operates properly if the D signal is stable (that is, not changing) at the time Clk goes from 1 to 0. But it may lead to unpredictable results if the D signal also changes at this time. Therefore, the designer of a logic circuit that generates the D signal must ensure that this signal is stable when the critical change in the clock signal takes place.

Figure 7.9 illustrates the critical timing region. The minimum time that the D signal must be stable prior to the negative edge of the Clk signal is called the *setup time*, t_{su}, of the latch. The minimum time that the D signal must remain stable after the negative edge of the Clk signal is called the *hold time*, t_h, of the latch. The values of t_{su} and t_h depend on the technology used. Manufacturers of integrated circuit chips provide this information on the data sheets that describe their chips. Typical values for CMOS technology are $t_{su} = 3$ ns and $t_h = 2$ ns. We will give examples of how setup and hold times affect the speed of operation of circuits in section 7.13. The behavior of storage elements when setup or hold times are violated is discussed in section 10.3.3.

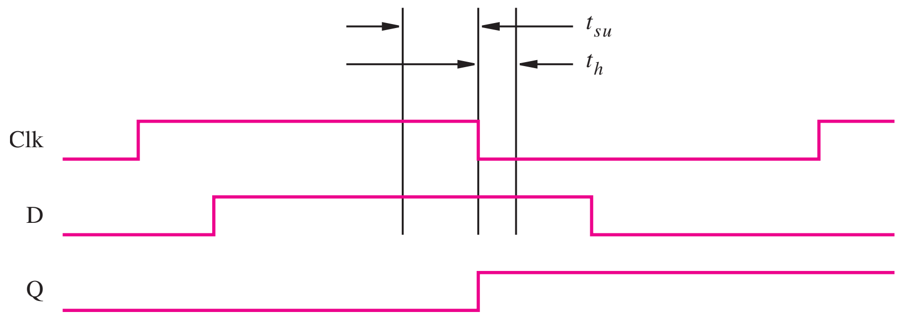


Figure 7.9 Setup and hold times.

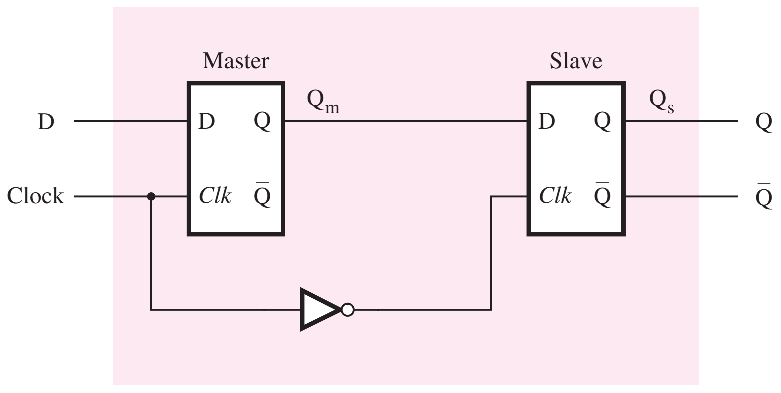
7.4 Master-Slave and Edge-Triggered D Flip-Flops

In the level-sensitive latches, the state of the latch keeps changing according to the values of input signals during the period when the clock signal is active (equal to 1 in our examples). As we will see in sections 7.8 and 7.9, there is also a need for storage elements that can change their states no more than once during one clock cycle. We will discuss two types of circuits that exhibit such behavior.

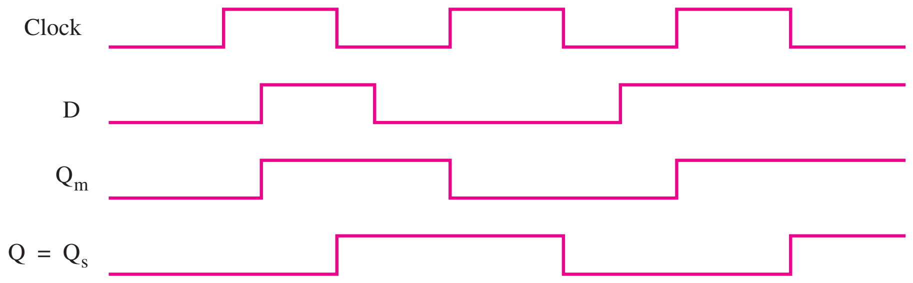
7.4.1 Master-Slave D Flip-Flop

Consider the circuit given in Figure 7.10*a*, which consists of two gated D latches. The first, called *master*, changes its state while *Clock* = 1. The second, called *slave*, changes its state while *Clock* = 0. The operation of the circuit is such that when the clock is high, the master tracks the value of the *D* input signal and the slave does not change. Thus the value of Q_m follows any changes in *D*, and the value of Q_s remains constant. When the clock signal changes to 0, the master stage stops following the changes in the *D* input. At the same time, the slave stage responds to the value of the signal Q_m and changes state accordingly. Since Q_m does not change while *Clock* = 0, the slave stage can undergo at most one change of state during a clock cycle. From the external observer's point of view, namely, the circuit connected to the output of the slave stage, the master-slave circuit changes its state at the negative-going edge of the clock. The *negative edge* is the edge where the clock signal changes from 1 to 0. Regardless of the number of changes in the *D* input to the master stage during one clock cycle, the observer of the Q_s signal will see only the change that corresponds to the *D* input at the negative edge of the clock.

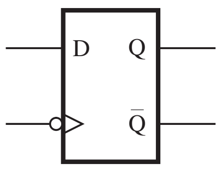
The circuit in Figure 7.10 is called a *master-slave D flip-flop*. The term *flip-flop* denotes a storage element that changes its output state at the edge of a controlling clock signal. The timing diagram for this flip-flop is shown in Figure 7.10*b*. A graphical symbol is given in Figure 7.10*c*. In the symbol we use the > mark to denote that the flip-flop responds to the "active edge" of the clock. We place a bubble on the clock input to indicate that the active edge for this particular circuit is the negative edge.



(a) Circuit



(b) Timing diagram

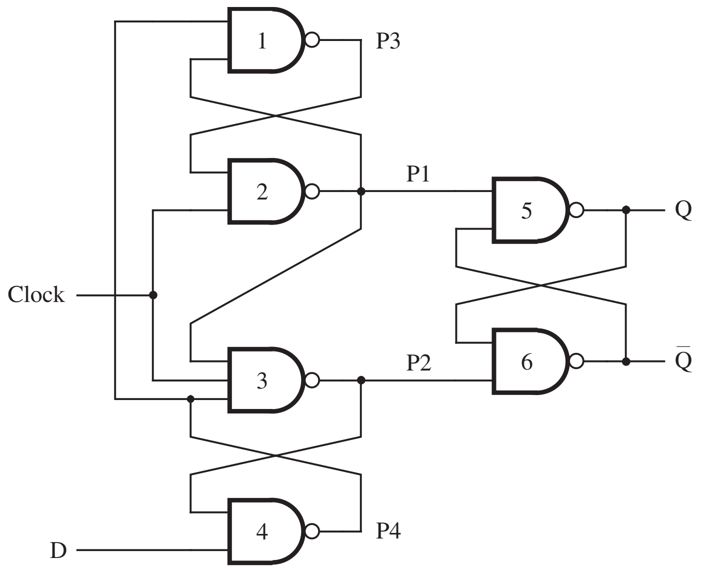


(c) Graphical symbol

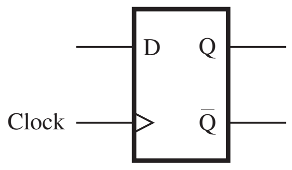
Figure 7.10 Master-slave D flip-flop.

7.4.2 Edge-Triggered D Flip-Flop

The output of the master-slave D flip-flop in Figure 7.10*a* responds on the negative edge of the clock signal. The circuit can be changed to respond to the positive clock edge by connecting the slave stage directly to the clock and the master stage to the complement of the clock. A different circuit that accomplishes the same task is presented in Figure 7.11*a*.



(a) Circuit



(b) Graphical symbol

Figure 7.11 A positive-edge-triggered D flip-flop.

It requires only six NAND gates and, hence, fewer transistors. The operation of the circuit is as follows. When *Clock* $= 0$, the outputs of gates 2 and 3 are high. Thus $P1 = P2 = 1$, which maintains the output latch, comprising gates 5 and 6, in its present state. At the same time, the signal $P3$ is equal to D, and $P4$ is equal to its complement $\overline{D}$. When *Clock* changes to 1, the following changes take place. The values of $P3$ and $P4$ are transmitted through gates 2 and 3 to cause $P1 = \overline{D}$ and $P2 = D$, which sets $\mathrm{Q} = D$ and $\overline{\mathrm{Q}} = \overline{D}$. To operate reliably, $P3$ and $P4$ must be stable when *Clock* changes from 0 to 1. Hence the setup time of the flip-flop is equal to the delay from the D input through gates 4 and 1 to $P3$. The hold time is given by the delay through gate 3 because once $P2$ is stable, the changes in D no longer matter.

For proper operation it is necessary to show that, after *Clock* changes to 1, any further changes in D will not affect the output latch as long as *Clock* $= 1$. We have to consider two cases. Suppose first that $D = 0$ at the positive edge of the clock. Then $P2 = 0$, which will keep the output of gate 4 equal to 1 as long as $Clock = 1$, regardless of the value of the D input. The second case is if $D = 1$ at the positive edge of the clock. Then $P1 = 0$, which forces the outputs of gates 1 and 3 to be equal to 1, regardless of the D input. Therefore, the flip-flop ignores changes in the D input while $Clock = 1$.

Figure 7.11*b* gives a graphical symbol for this flip-flop. The clock input indicates that the positive edge of the clock is the active edge. A similar circuit, constructed with NOR gates, can be used as a negative-edge-triggered flip-flop.

Level-Sensitive versus Edge-Triggered Storage Elements

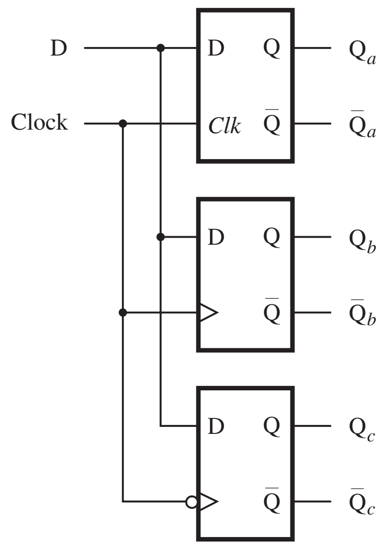
Figure 7.12 shows three different types of storage elements that are driven by the same data and clock inputs. The first element is a gated D latch, which is level sensitive. The second one is a positive-edge-triggered D flip-flop, and the third one is a negative-edge-triggered D flip-flop. To accentuate the differences between these storage elements, the D input changes its values more than once during each half of the clock cycle. Observe that the gated D latch follows the D input as long as the clock is high. The positive-edge-triggered flip-flop responds only to the value of D when the clock changes from 0 to 1. The negative-edge-triggered flip-flop responds only to the value of D when the clock changes from 1 to 0.

7.4.3 D FLIP-FLOPS WITH CLEAR AND PRESET

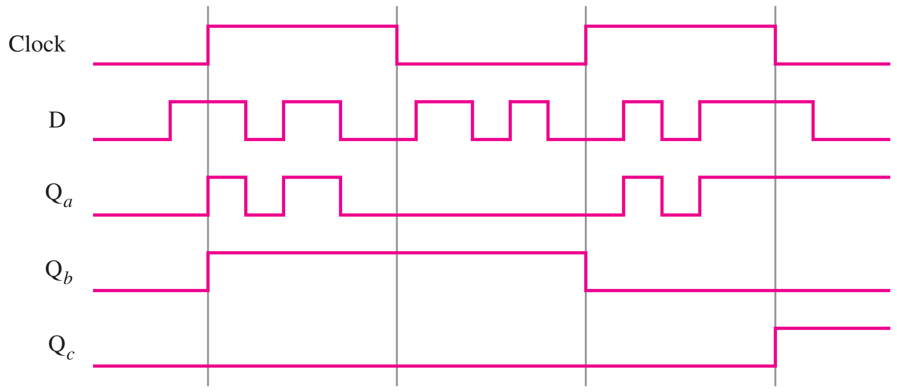
Flip-flops are often used for implementation of circuits that can have many possible states, where the response of the circuit depends not only on the present values of the circuit's inputs but also on the particular state that the circuit is in at that time. We will discuss a general form of such circuits in Chapter 8. A simple example is a counter circuit that counts the number of occurrences of some event, perhaps passage of time. We will discuss counters in detail in section 7.9. A counter comprises a number of flip-flops, whose outputs are interpreted as a number. The counter circuit has to be able to increment or decrement the number. It is also important to be able to force the counter into a known initial state (count). Obviously, it must be possible to clear the count to zero, which means that all flip-flops must have Q = 0. It is equally useful to be able to preset each flip-flop to Q = 1, to insert some specific count as the initial value in the counter. These features can be incorporated into the circuits of Figures 7.10 and 7.11 as follows.

Figure 7.13*a* shows an implementation of the circuit in Figure 7.10*a* using NAND gates. The master stage is just the gated D latch of Figure 7.8*a*. Instead of using another latch of the same type for the slave stage, we can use the slightly simpler gated SR latch of Figure 7.7. This eliminates one NOT gate from the circuit.

A simple way of providing the clear and preset capability is to add an extra input to each NAND gate in the cross-coupled latches, as indicated in blue. Placing a 0 on the *Clear* input will force the flip-flop into the state Q = 0. If $Clear = 1$, then this input will have no effect on the NAND gates. Similarly, $Preset = 0$ forces the flip-flop into the state Q = 1, while $Preset = 1$ has no effect. To denote that the *Clear* and *Preset* inputs are active when their value is 0, we placed an overbar on the names in the figure. We should note that the circuit that uses this flip-flop should not try to force both *Clear* and *Preset* to 0 at the same time. A graphical symbol for this flip-flop is shown in Figure 7.13*b*.



(a) Circuit

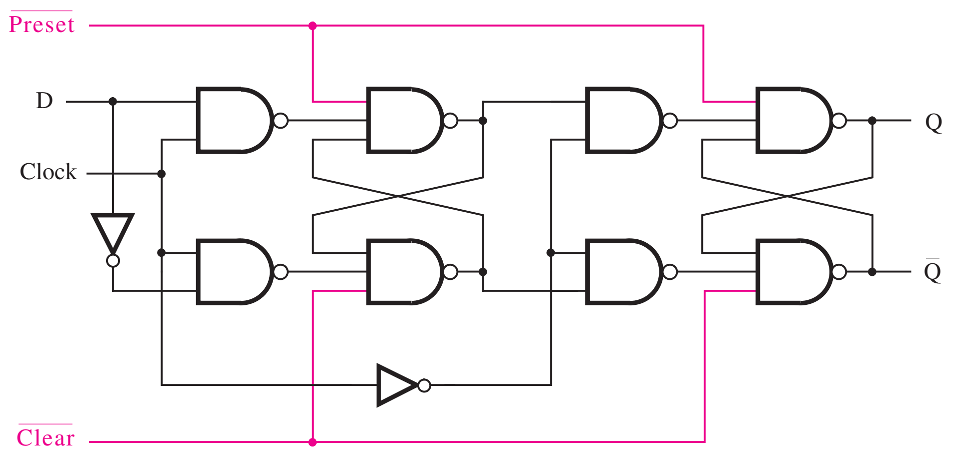


(b) Timing diagram

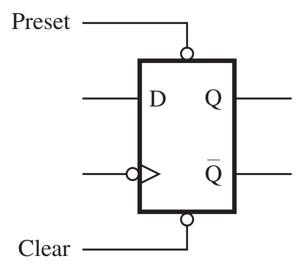
Figure 7.12 Comparison of level-sensitive and edge-triggered D storage elements.

A similar modification can be done on the edge-triggered flip-flop of Figure 7.11*a*, as indicated in Figure 7.14*a*. Again, both *Clear* and *Preset* inputs are active low. They do not disturb the flip-flop when they are equal to 1.

In the circuits in Figures 7.13*a* and 7.14*a*, the effect of a low signal on either the *Clear* or *Preset* input is immediate. For example, if *Clear* = 0 then the flip-flop goes into the state Q = 0 immediately, regardless of the value of the clock signal. In such a circuit, where the *Clear* signal is used to clear a flip-flop without regard to the clock signal, we say that the



(a) Circuit



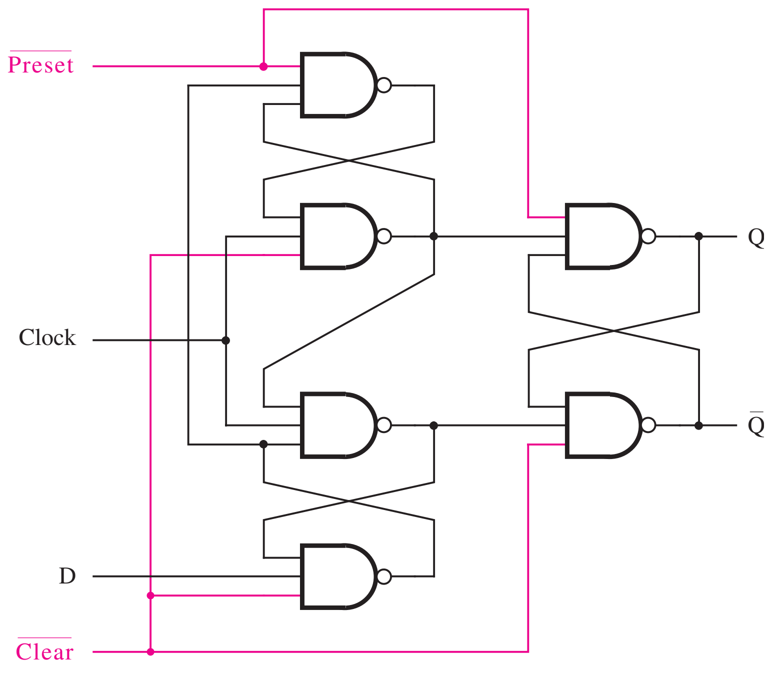
(b) Graphical symbol

Figure 7.13 Master-slave D flip-flop with *Clear* and *Preset*.

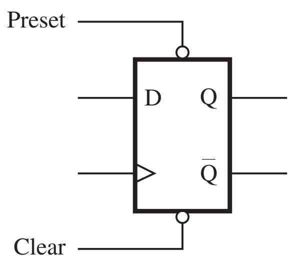
flip-flop has an *asynchronous clear*. In practice, it is often preferable to clear the flip-flops on the active edge of the clock. Such *synchronous clear* can be accomplished as shown in Figure 7.15. The flip-flop operates normally when the *Clear* input is equal to 1. But if *Clear* goes to 0, then on the next positive edge of the clock the flip-flop will be cleared to 0. We will examine the clearing of flip-flops in more detail in section 7.10.

7.5 T Flip-Flop

The D flip-flop is a versatile storage element that can be used for many purposes. By including some simple logic circuitry to drive its input, the D flip-flop may appear to be a different type of storage element. An interesting modification is presented in Figure 7.16*a*. This circuit uses a positive-edge-triggered D flip-flop. The *feedback* connections make the input signal *D* equal to either the value of Q or $\overline{\text{Q}}$ under the control of the signal that is labeled *T*. On each positive edge of the clock, the flip-flop may change its state Q(t). If



(a) Circuit



(b) Graphical symbol

Figure 7.14 Positive-edge-triggered D flip-flop with *Clear* and *Preset*.

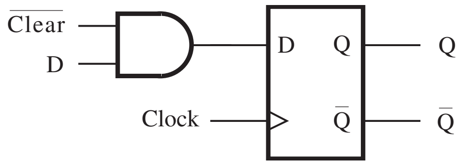
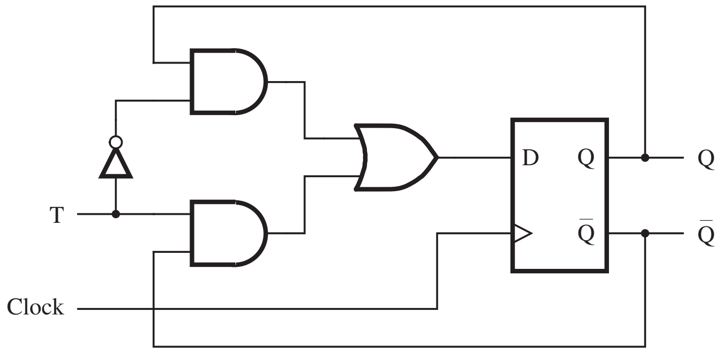


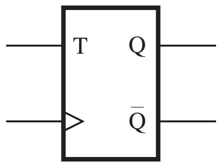
Figure 7.15 Synchronous reset for a D flip-flop.



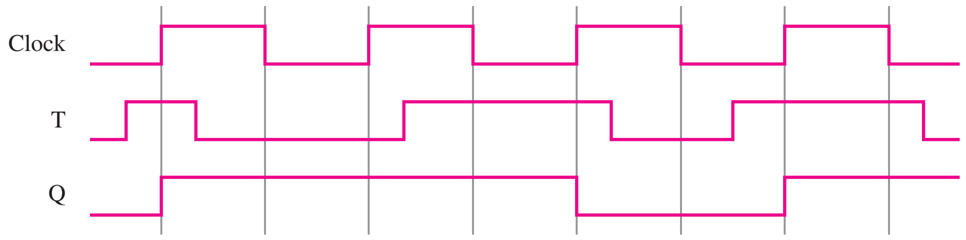
(a) Circuit

T	Q(*t* + 1)
0	Q(*t*)
1	$\overline{Q}(t)$

(b) Truth table



(c) Graphical symbol



(d) Timing diagram

Figure 7.16 T flip-flop.

$T = 0$, then $D = Q$ and the state will remain the same, that is, $Q(t+1) = Q(t)$. But if $T = 1$, then $D = \overline{Q}$ and the new state will be $Q(t+1) = \overline{Q}(t)$. Therefore, the overall operation of the circuit is that it retains its present state if $T = 0$, and it reverses its present state if $T = 1$.

The operation of the circuit is specified in the form of a truth table in Figure 7.16*b*. Any circuit that implements this truth table is called a *T flip-flop*. The name T flip-flop derives from the behavior of the circuit, which "toggles" its state when $T = 1$. The toggle feature makes the T flip-flop a useful element for building counter circuits, as we will see in section 7.9.

7.5.1 Configurable Flip-Flops

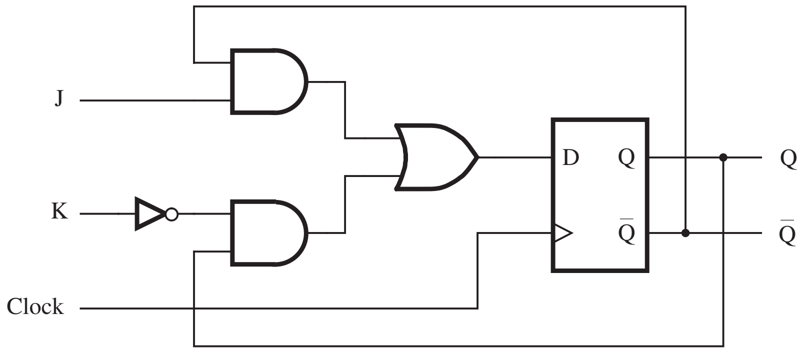
For some circuits one type of flip-flop may lead to a more efficient implementation than a different type of flip-flop. In general purpose chips like PLDs, the flip-flops that are provided are sometimes *configurable*, which means that a flip-flop circuit can be configured to be either D, T, or some other type. For example, in some PLDs the flip-flops can be configured as either D or T types (see problems 7.6 and 7.8).

7.6 JK Flip-Flop

Another interesting circuit can be derived from Figure 7.16*a*. Instead of using a single control input, T, we can use two inputs, J and K, as indicated in Figure 7.17*a*. For this circuit the input D is defined as

$$D = J\overline{Q} + \overline{K}Q$$

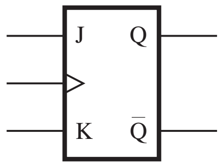
A corresponding truth table is given in Figure 7.17*b*. The circuit is called a *JK flip-flop*. It combines the behaviors of SR and T flip-flops in a useful way. It behaves as the SR flip-flop,



(a) Circuit

J	K	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	$\overline{Q}(t)$

(b) Truth table



(c) Graphical symbol

Figure 7.17 JK flip-flop.

where $J = S$ and $K = R$, for all input values except $J = K = 1$. For the latter case, which has to be avoided in the SR flip-flop, the JK flip-flop toggles its state like the T flip-flop.

The JK flip-flop is a versatile circuit. It can be used for straight storage purposes, just like the D and SR flip-flops. But it can also serve as a T flip-flop by connecting the J and K inputs together.

7.7 Summary of Terminology

We have used the terminology that is quite common. But the reader should be aware that different interpretations of the terms *latch* and *flip-flop* can be found in the literature. Our terminology can be summarized as follows:

Basic latch is a feedback connection of two NOR gates or two NAND gates, which can store one bit of information. It can be set to 1 using the S input and reset to 0 using the R input.

Gated latch is a basic latch that includes input gating and a control input signal. The latch retains its existing state when the control input is equal to 0. Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock. We considered two types of gated latches:

- **Gated SR latch** uses the S and R inputs to set the latch to 1 or reset it to 0, respectively.
- **Gated D latch** uses the D input to force the latch into a state that has the same logic value as the D input.

A **flip-flop** is a storage element based on the gated latch principle, which can have its output state changed only on the edge of the controlling clock signal. We considered two types:

- **Edge-triggered flip-flop** is affected only by the input values present when the active edge of the clock occurs.
- **Master-slave flip-flop** is built with two gated latches. The master stage is active during half of the clock cycle, and the slave stage is active during the other half. The output value of the flip-flop changes on the edge of the clock that activates the transfer into the slave stage. Master-slave flip-flops can be edge-triggered or level sensitive. If the master stage is a gated D latch, then it behaves as an edge-triggered flip-flop. If the master stage is a gated SR latch, then the flip-flop is level sensitive (see problem 7.19).

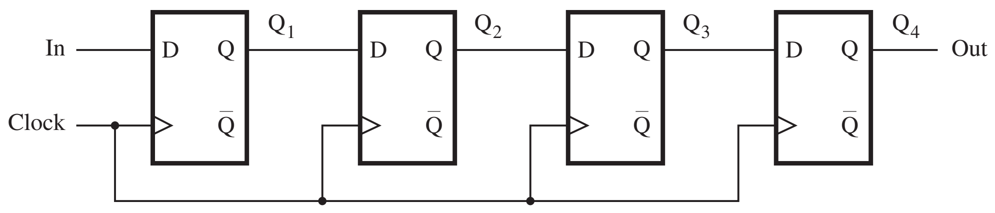
7.8 Registers

A flip-flop stores one bit of information. When a set of n flip-flops is used to store n bits of information, such as an n-bit number, we refer to these flip-flops as a *register*. A common clock is used for each flip-flop in a register, and each flip-flop operates as described in the previous sections. The term register is merely a convenience for referring to *n*-bit structures consisting of flip-flops.

7.8.1 Shift Register

In section 5.6 we explained that a given number is multiplied by 2 if its bits are shifted one bit position to the left and a 0 is inserted as the new least-significant bit. Similarly, the number is divided by 2 if the bits are shifted one bit-position to the right. A register that provides the ability to shift its contents is called a *shift register*.

Figure 7.18*a* shows a four-bit shift register that is used to shift its contents one bit-position to the right. The data bits are loaded into the shift register in a serial fashion using the *In* input. The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock. An illustration of the transfer is given in Figure 7.18*b*, which shows what happens when the signal values at *In* during eight consecutive clock cycles are 1, 0, 1, 1, 1, 0, 0, and 0, assuming that the initial state of all flip-flops is 0.



(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

(b) A sample sequence

Figure 7.18 A simple shift register.

To implement a shift register, it is necessary to use either edge-triggered or master-slave flip-flops. The level-sensitive gated latches are not suitable, because a change in the value of *In* would propagate through more than one latch during the time when the clock is equal to 1.

7.8.2 Parallel-Access Shift Register

In computer systems it is often necessary to transfer *n*-bit data items. This may be done by transmitting all bits at once using *n* separate wires, in which case we say that the transfer is performed in *parallel*. But it is also possible to transfer all bits using a single wire, by performing the transfer one bit at a time, in *n* consecutive clock cycles. We refer to this scheme as *serial* transfer. To transfer an *n*-bit data item serially, we can use a shift register that can be loaded with all *n* bits in parallel (in one clock cycle). Then during the next *n* clock cycles, the contents of the register can be shifted out for serial transfer. The reverse operation is also needed. If bits are received serially, then after *n* clock cycles the contents of the register can be accessed in parallel as an *n*-bit item.

Figure 7.19 shows a four-bit shift register that allows the parallel access. Instead of using the normal shift register connection, the *D* input of each flip-flop is connected to

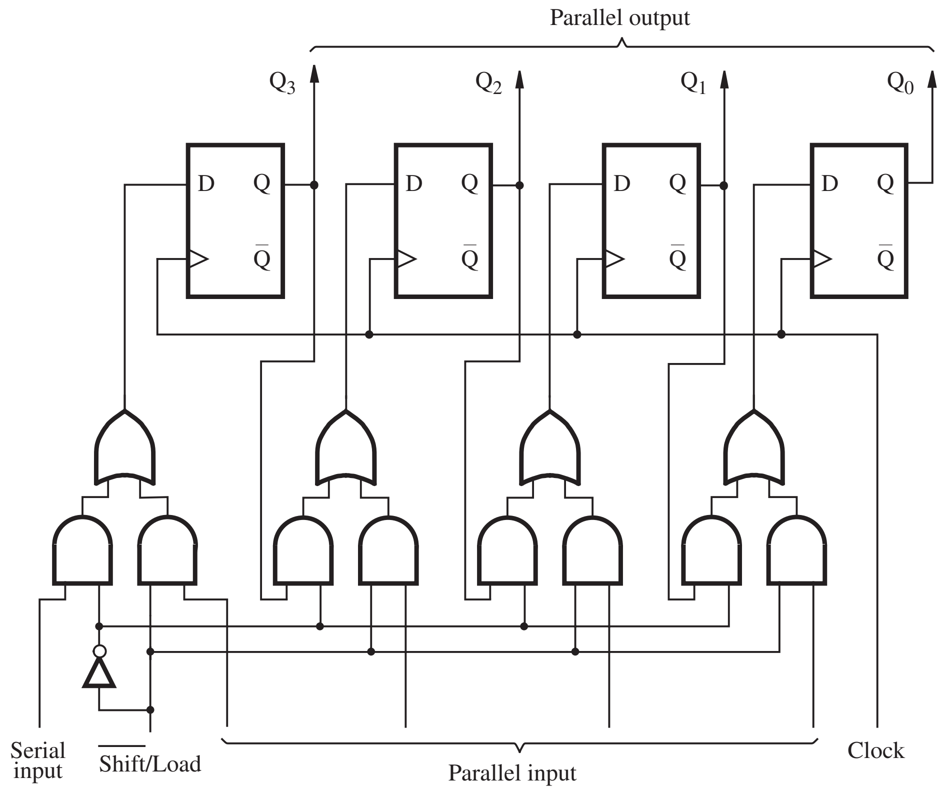


Figure 7.19 Parallel-access shift register.

two different sources. One source is the preceding flip-flop, which is needed for the shift-register operation. The other source is the external input that corresponds to the bit that is to be loaded into the flip-flop as a part of the parallel-load operation. The control signal $\overline{Shift}/Load$ is used to select the mode of operation. If $\overline{Shift}/Load = 0$, then the circuit operates as a shift register. If $\overline{Shift}/Load = 1$, then the parallel input data are loaded into the register. In both cases the action takes place on the positive edge of the clock.

In Figure 7.19 we have chosen to label the flip-flops outputs as $Q_3, \ldots, Q_0$ because shift registers are often used to hold binary numbers. The contents of the register can be accessed in parallel by observing the outputs of all flip-flops. The flip-flops can also be accessed serially, by observing the values of Q_0 during consecutive clock cycles while the contents are being shifted. A circuit in which data can be loaded in series and then accessed in parallel is called a series-to-parallel converter. Similarly, the opposite type of circuit is a parallel-to-series converter. The circuit in Figure 7.19 can perform both of these functions.

7.9 COUNTERS

In Chapter 5 we dealt with circuits that perform arithmetic operations. We showed how adder/subtractor circuits can be designed, either using a simple cascaded (ripple-carry) structure that is inexpensive but slow or using a more complex carry-lookahead structure that is both more expensive and faster. In this section we examine special types of addition and subtraction operations, which are used for the purpose of counting. In particular, we want to design circuits that can increment or decrement a count by 1. Counter circuits are used in digital systems for many purposes. They may count the number of occurrences of certain events, generate timing intervals for control of various tasks in a system, keep track of time elapsed between specific events, and so on.

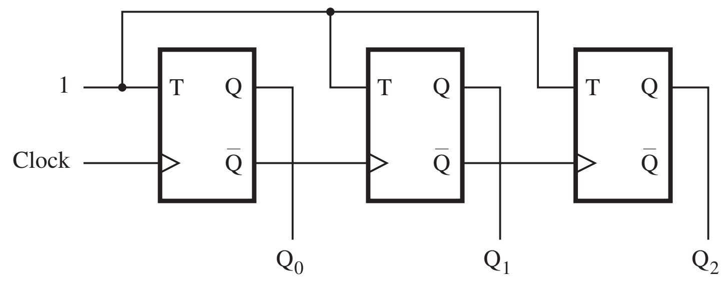
Counters can be implemented using the adder/subtractor circuits discussed in Chapter 5 and the registers discussed in section 7.8. However, since we only need to change the contents of a counter by 1, it is not necessary to use such elaborate circuits. Instead, we can use much simpler circuits that have a significantly lower cost. We will show how the counter circuits can be designed using T and D flip-flops.

7.9.1 ASYNCHRONOUS COUNTERS

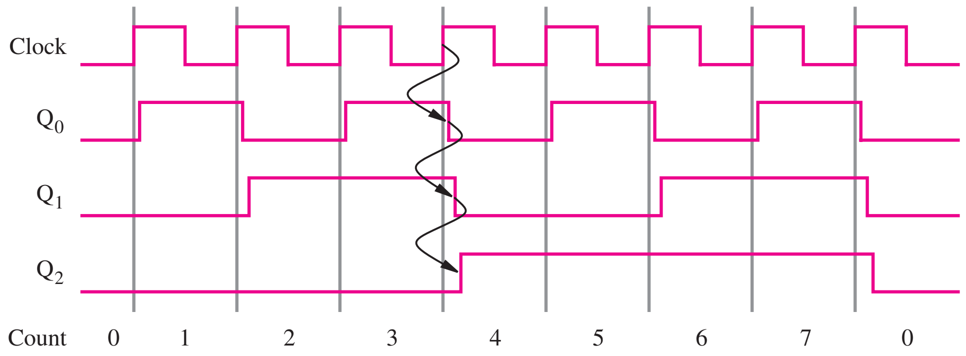
The simplest counter circuits can be built using T flip-flops because the toggle feature is naturally suited for the implementation of the counting operation.

Up-Counter with T Flip-Flops

Figure 7.20*a* gives a three-bit counter capable of counting from 0 to 7. The clock inputs of the three flip-flops are connected in cascade. The *T* input of each flip-flop is connected to a constant 1, which means that the state of the flip-flop will be reversed (toggled) at each positive edge of its clock. We are assuming that the purpose of this circuit is to count the number of pulses that occur on the primary input called *Clock*. Thus the clock input of the first flip-flop is connected to the *Clock* line. The other two flip-flops have their clock inputs driven by the $\overline{Q}$ output of the preceding flip-flop. Therefore, they toggle their state



(a) Circuit



(b) Timing diagram

Figure 7.20 A three-bit up-counter.

whenever the preceding flip-flop changes its state from Q = 1 to Q = 0, which results in a positive edge of the $\overline{Q}$ signal.

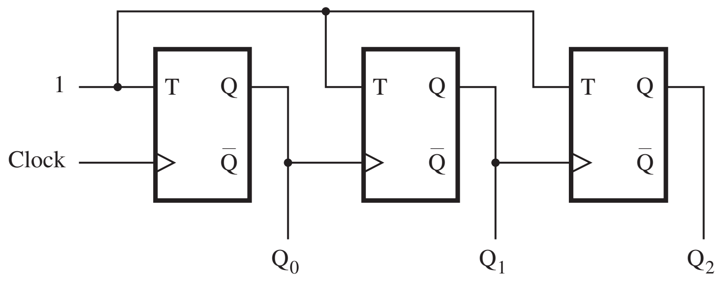
Figure 7.20*b* shows a timing diagram for the counter. The value of Q_0 toggles once each clock cycle. The change takes place shortly after the positive edge of the *Clock* signal. The delay is caused by the propagation delay through the flip-flop. Since the second flip-flop is clocked by $\overline{Q}_0$, the value of Q_1 changes shortly after the negative edge of the Q_0 signal. Similarly, the value of Q_2 changes shortly after the negative edge of the Q_1 signal. If we look at the values $Q_2Q_1Q_0$ as the count, then the timing diagram indicates that the counting sequence is 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, and so on. This circuit is a modulo-8 counter. Because it counts in the upward direction, we call it an *up-counter*.

The counter in Figure 7.20*a* has three *stages*, each comprising a single flip-flop. Only the first stage responds directly to the *Clock* signal; we say that this stage is *synchronized* to the clock. The other two stages respond after an additional delay. For example, when *Count* = 3, the next clock pulse will cause the *Count* to go to 4. As indicated by the arrows in the timing diagram in Figure 7.20*b*, this change requires the toggling of the states of all three flip-flops. The change in Q_0 is observed only after a propagation delay from the positive edge of *Clock*. The Q_1 and Q_2 flip-flops have not yet changed; hence for a brief time the count is $Q_2Q_1Q_0 = 010$. The change in Q_1 appears after a second propagation delay, at which point the count is 000. Finally, the change in Q_2 occurs after a third delay, at which point the stable state of the circuit is reached and the count is 100. This behavior is similar to the rippling of carries in the ripple-carry adder circuit of Figure 5.6. The circuit in Figure 7.20*a* is an *asynchronous counter*, or a *ripple counter*.

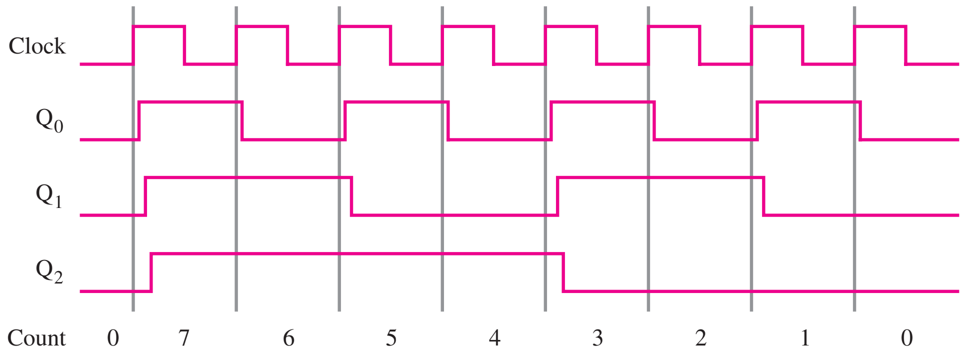
Down-Counter with T Flip-Flops

A slight modification of the circuit in Figure 7.20*a* is presented in Figure 7.21*a*. The only difference is that in Figure 7.21*a* the clock inputs of the second and third flip-flops are driven by the Q outputs of the preceding stages, rather than by the $\overline{Q}$ outputs. The timing diagram, given in Figure 7.21*b*, shows that this circuit counts in the sequence 0, 7, 6, 5, 4, 3, 2, 1, 0, 7, and so on. Because it counts in the downward direction, we say that it is a *down-counter*.

It is possible to combine the functionality of the circuits in Figures 7.20*a* and 7.21*a* to form a counter that can count either up or down. Such a counter is called an *up/down-counter*. We leave the derivation of this counter as an exercise for the reader (problem 7.16).



(a) Circuit



(b) Timing diagram

Figure 7.21 A three-bit down-counter.

7.9.2 SYNCHRONOUS COUNTERS

The asynchronous counters in Figures 7.20*a* and 7.21*a* are simple, but not very fast. If a counter with a larger number of bits is constructed in this manner, then the delays caused by the cascaded clocking scheme may become too long to meet the desired performance requirements. We can build a faster counter by clocking all flip-flops at the same time, using the approach described below.

Synchronous Counter with T Flip-Flops

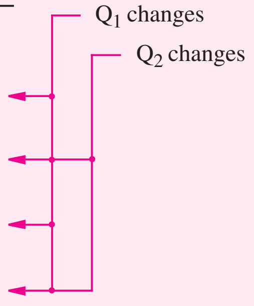
Table 7.1 shows the contents of a three-bit up-counter for eight consecutive clock cycles, assuming that the count is initially 0. Observing the pattern of bits in each row of the table, it is apparent that bit Q_0 changes on each clock cycle. Bit Q_1 changes only when $Q_0 = 1$. Bit Q_2 changes only when both Q_1 and Q_0 are equal to 1. In general, for an *n*-bit up-counter, a given flip-flop changes its state only when all the preceding flip-flops are in the state Q = 1. Therefore, if we use T flip-flops to realize the counter, then the *T* inputs are defined as

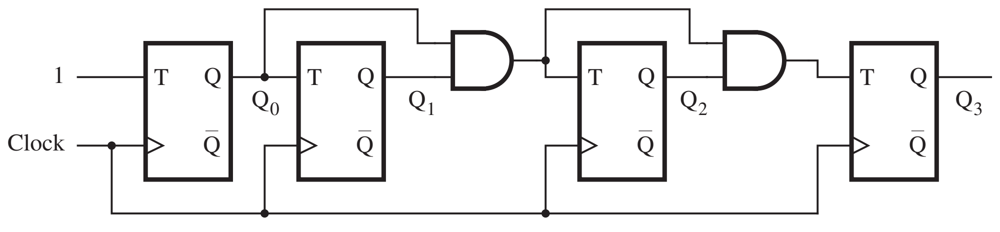
$$
\begin{aligned}
T_0 &= 1\\
T_1 &= Q_0\\
T_2 &= Q_0Q_1\\
T_3 &= Q_0Q_1Q_2\\
&\;\;\vdots\\
T_n &= Q_0Q_1\cdots Q_{n-1}
\end{aligned}
$$

An example of a four-bit counter based on these expressions is given in Figure 7.22*a*. Instead of using AND gates of increased size for each stage, which may lead to fan-in problems, we use a factored arrangement, as shown in the figure. This arrangement does not slow down the response of the counter, because all flip-flops change their states after a

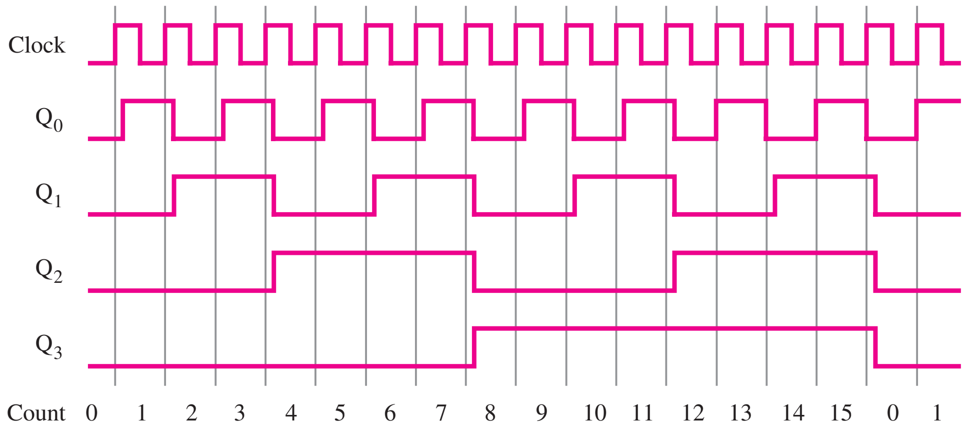
Table 7.1 Derivation of the synchronous up-counter.

Clock cycle	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0





(a) Circuit



(b) Timing diagram

Figure 7.22 A four-bit synchronous up-counter.

propagation delay from the positive edge of the clock. Note that a change in the value of Q_0 may have to propagate through several AND gates to reach the flip-flops in the higher stages of the counter, which requires a certain amount of time. This time must not exceed the clock period. Actually, it must be less than the clock period minus the setup time for the flip-flops.

Figure 7.22*b* gives a timing diagram. It shows that the circuit behaves as a modulo-16 up-counter. Because all changes take place with the same delay after the active edge of the *Clock* signal, the circuit is called a *synchronous counter*.

Enable and Clear Capability

The counters in Figures 7.20 through 7.22 change their contents in response to each clock pulse. Often it is desirable to be able to inhibit counting, so that the count remains in its present state. This may be accomplished by including an *Enable* control signal, as indicated in Figure 7.23. The circuit is the counter of Figure 7.22, where the *Enable* signal controls directly the *T* input of the first flip-flop. Connecting the *Enable* also to the AND-

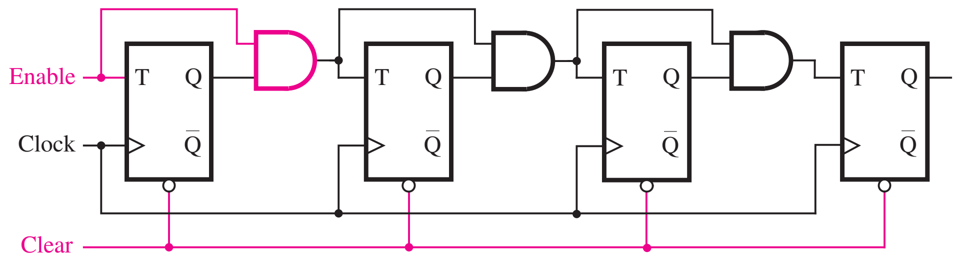


Figure 7.23 Inclusion of Enable and Clear capability.

gate chain means that if *Enable* = 0, then all *T* inputs will be equal to 0. If *Enable* = 1, then the counter operates as explained previously.

In many applications it is necessary to start with the count equal to zero. This is easily achieved if the flip-flops can be cleared, as explained in section 7.4.3. The clear inputs on all flip-flops can be tied together and driven by a *Clear* control input.

Synchronous Counter with D Flip-Flops

While the toggle feature makes T flip-flops a natural choice for the implementation of counters, it is also possible to build counters using other types of flip-flops. The JK flip-flops can be used in exactly the same way as the T flip-flops because if the *J* and *K* inputs are tied together, a JK flip-flop becomes a T flip-flop. We will now consider using D flip-flops for this purpose.

It is not obvious how D flip-flops can be used to implement a counter. We will present a formal method for deriving such circuits in Chapter 8. Here we will present a circuit structure that meets the requirements but will leave the derivation for Chapter 8. Figure 7.24 gives a four-bit up-counter that counts in the sequence 0, 1, 2, . . . , 14, 15, 0, 1, and so on. The count is indicated by the flip-flop outputs $Q_3Q_2Q_1Q_0$. If we assume that *Enable* = 1, then the *D* inputs of the flip-flops are defined by the expressions

$$\begin{aligned} D_0 &= \overline{Q}_0 = 1 \oplus Q_0 \\ D_1 &= Q_1 \oplus Q_0 \\ D_2 &= Q_2 \oplus Q_1Q_0 \\ D_3 &= Q_3 \oplus Q_2Q_1Q_0 \end{aligned}$$

For a larger counter the *i*th stage is defined by

$$D_i = Q_i \oplus Q_{i-1}Q_{i-2} \cdots Q_1Q_0$$

We will show how to derive these equations in Chapter 8.

We have included the *Enable* control signal so that the counter counts the clock pulses only if *Enable* = 1. In effect, the above equations are modified to implement the circuit in the figure as follows

$$\begin{aligned} D_0 &= Q_0 \oplus Enable \\ D_1 &= Q_1 \oplus Q_0 \cdot Enable \end{aligned}$$

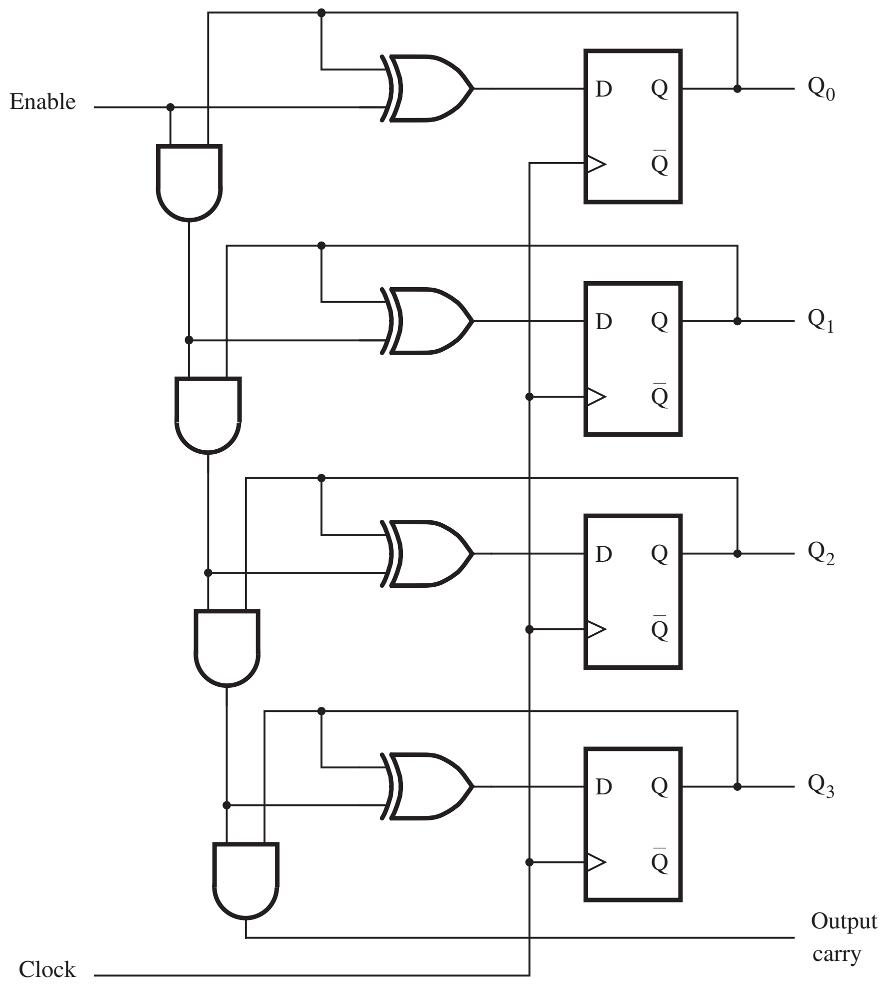


Figure 7.24 A four-bit counter with D flip-flops.

$$D_2 = \mathrm{Q}_2 \oplus \mathrm{Q}_1 \cdot \mathrm{Q}_0 \cdot Enable$$
$$D_3 = \mathrm{Q}_3 \oplus \mathrm{Q}_2 \cdot \mathrm{Q}_1 \cdot \mathrm{Q}_0 \cdot Enable$$

The operation of the counter is based on our observation for Table 7.1 that the state of the flip-flop in stage *i* changes only if all preceding flip-flops are in the state $\mathrm{Q} = 1$. This makes the output of the AND gate that feeds stage *i* equal to 1, which causes the output of the XOR gate connected to D_i to be equal to $\overline{\mathrm{Q}}_i$. Otherwise, the output of the XOR gate provides $D_i = \mathrm{Q}_i$, and the flip-flop remains in the same state. This resembles the carry propagation in a carry-lookahead adder circuit (see section 5.4); hence the AND-gate chain can be thought of as the *carry chain*. Even though the circuit is only a four-bit counter, we have included an extra AND that produces the "output carry." This signal makes it easy to concatenate two such four-bit counters to create an eight-bit counter.

Finally, the reader should note that the counter in Figure 7.24 is essentially the same as the circuit in Figure 7.23. We showed in Figure 7.16*a* that a T flip-flop can be formed from a D flip-flop by providing the extra gating that gives

$$\begin{aligned} D &= \mathrm{Q}\overline{T} + \overline{\mathrm{Q}}T \\ &= \mathrm{Q} \oplus T \end{aligned}$$

Thus in each stage in Figure 7.24, the D flip-flop and the associated XOR gate implement the functionality of a T flip-flop.

7.9.3 COUNTERS WITH PARALLEL LOAD

Often it is necessary to start counting with the initial count being equal to 0. This state can be achieved by using the capability to clear the flip-flops as indicated in Figure 7.23. But sometimes it is desirable to start with a different count. To allow this mode of operation, a counter circuit must have some inputs through which the initial count can be loaded. Using the *Clear* and *Preset* inputs for this purpose is a possibility, but a better approach is discussed below.

The circuit of Figure 7.24 can be modified to provide the parallel-load capability as shown in Figure 7.25. A two-input multiplexer is inserted before each D input. One input to the multiplexer is used to provide the normal counting operation. The other input is a data bit that can be loaded directly into the flip-flop. A control input, *Load*, is used to choose the mode of operation. The circuit counts when $Load = 0$. A new initial value, $D_3D_2D_1D_0$, is loaded into the counter when $Load = 1$.

7.10 RESET SYNCHRONIZATION

We have already mentioned that it is important to be able to clear, or *reset*, the contents of a counter prior to commencing a counting operation. This can be done using the clear capability of the individual flip-flops. But we may also be interested in resetting the count to 0 during the normal counting process. An n-bit up-counter functions naturally as a modulo-2^n counter. Suppose that we wish to have a counter that counts modulo some base that is not a power of 2. For example, we may want to design a modulo-6 counter, for which the counting sequence is 0, 1, 2, 3, 4, 5, 0, 1, and so on.

The most straightforward approach is to recognize when the count reaches 5 and then reset the counter. An AND gate can be used to detect the occurrence of the count of 5. Actually, it is sufficient to ascertain that $\mathrm{Q}_2 = \mathrm{Q}_0 = 1$, which is true only for 5 in our desired counting sequence. A circuit based on this approach is given in Figure 7.26*a*. It uses a three-bit synchronous counter of the type depicted in Figure 7.25. The parallel-load feature of the counter is used to reset its contents when the count reaches 5. The resetting action takes place at the positive clock edge after the count has reached 5. It involves loading $D_2D_1D_0 = 000$ into the flip-flops. As seen in the timing diagram in Figure 7.26*b*,

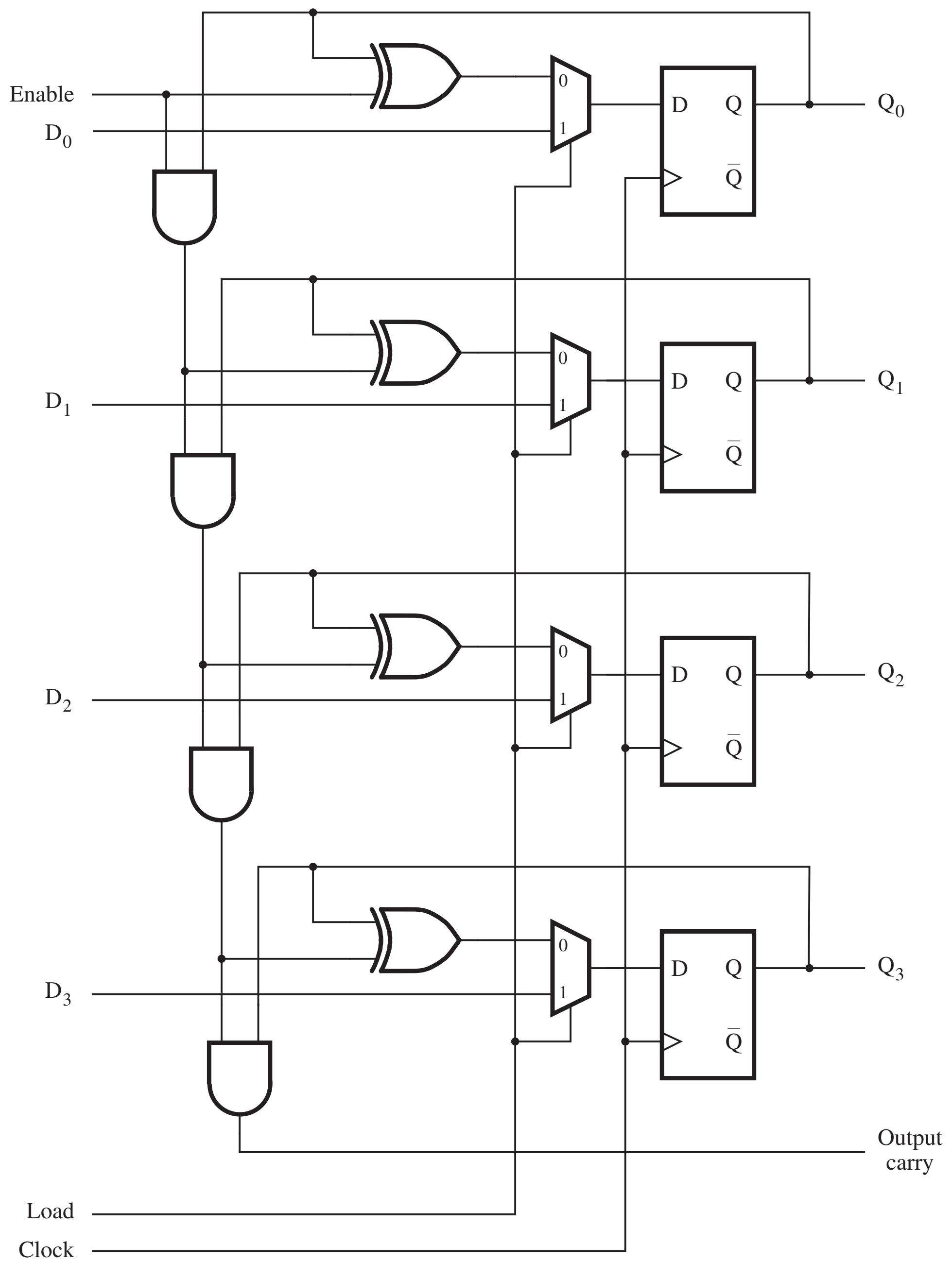
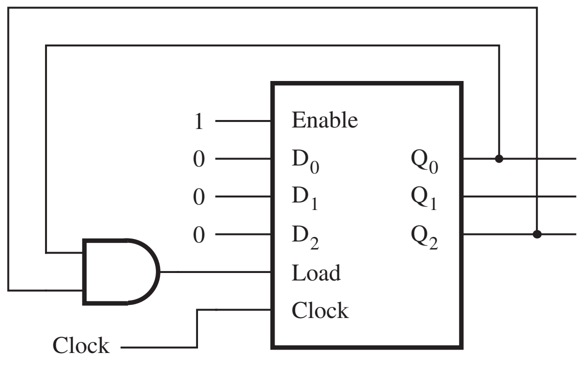
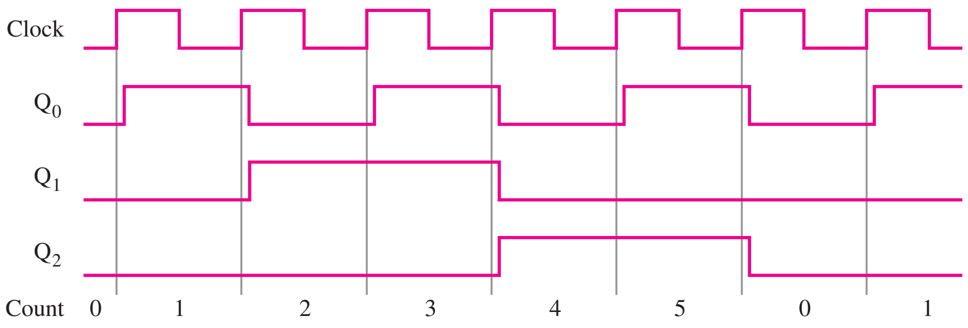


Figure 7.25 A counter with parallel-load capability.



(a) Circuit

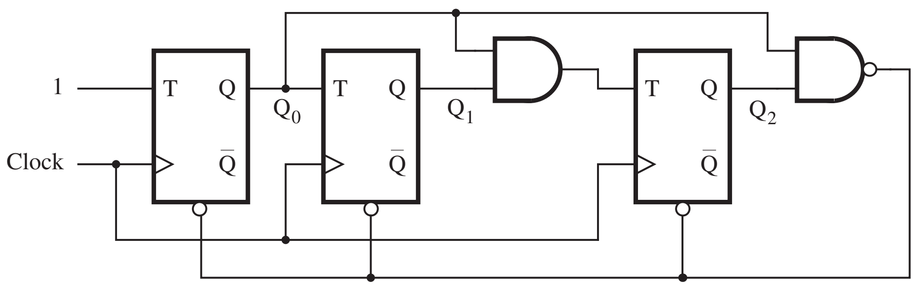


(b) Timing diagram

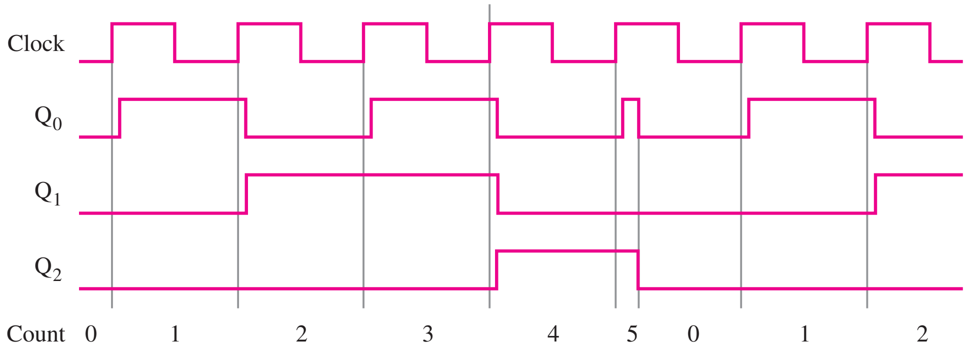
Figure 7.26 A modulo-6 counter with synchronous reset.

the desired counting sequence is achieved, with each value of the count being established for one full clock cycle. Because the counter is reset on the active edge of the clock, we say that this type of counter has a *synchronous reset*.

Consider now the possibility of using the clear feature of individual flip-flops, rather than the parallel-load approach. The circuit in Figure 7.27*a* illustrates one possibility. It uses the counter structure of Figure 7.22*a*. Since the clear inputs are active when low, a NAND gate is used to detect the occurrence of the count of 5 and cause the clearing of all three flip-flops. Conceptually, this seems to work fine, but closer examination reveals a potential problem. The timing diagram for this circuit is given in Figure 7.27*b*. It shows a difficulty that arises when the count is equal to 5. As soon as the count reaches this value, the NAND gate triggers the resetting action. The flip-flops are cleared to 0 a short time after the NAND gate has detected the count of 5. This time depends on the gate delays in the



(a) Circuit



(b) Timing diagram

Figure 7.27 A modulo-6 counter with asynchronous reset.

circuit, but not on the clock. Therefore, signal values $Q_2Q_1Q_0 = 101$ are maintained for a time that is much less than a clock cycle. Depending on a particular application of such a counter, this may be adequate, but it may also be completely unacceptable. For example, if the counter is used in a digital system where all operations in the system are synchronized by the same clock, then this narrow pulse denoting *Count* = 5 would not be seen by the rest of the system. To solve this problem, we could try to use a modulo-7 counter instead, assuming that the system would ignore the short pulse that denotes the count of 6. This is not a good way of designing circuits, because undesirable pulses often cause unforeseen difficulties in practice. The approach employed in Figure 7.27*a* is said to use *asynchronous reset*.

The timing diagrams in Figures 7.26*b* and 7.27*b* suggest that synchronous reset is a better choice than asynchronous reset. The same observation is true if the natural counting sequence has to be broken by loading some value other than zero. The new value of the count can be established cleanly using the parallel-load feature. The alternative of using the clear and preset capability of individual flip-flops to set their states to reflect the desired count has the same problems as discussed in conjunction with the asynchronous reset.

7.11 OTHER TYPES OF COUNTERS

In this section we discuss three other types of counters that can be found in practical applications. The first uses the decimal counting sequence, and the other two generate sequences of codes that do not represent binary numbers.

7.11.1 BCD COUNTER

Binary-coded-decimal (BCD) counters can be designed using the approach explained in section 7.10. A two-digit BCD counter is presented in Figure 7.28. It consists of two modulo-10 counters, one for each BCD digit, which we implemented using the parallel-load four-bit counter of Figure 7.25. Note that in a modulo-10 counter it is necessary to reset the four flip-flops after the count of 9 has been obtained. Thus the *Load* input to each stage is equal to 1 when $Q_3 = Q_0 = 1$, which causes 0s to be loaded into the flip-flops at the next positive edge of the clock signal. Whenever the count in stage 0, BCD_0, reaches 9 it is necessary to enable the second stage so that it will be incremented when the next clock

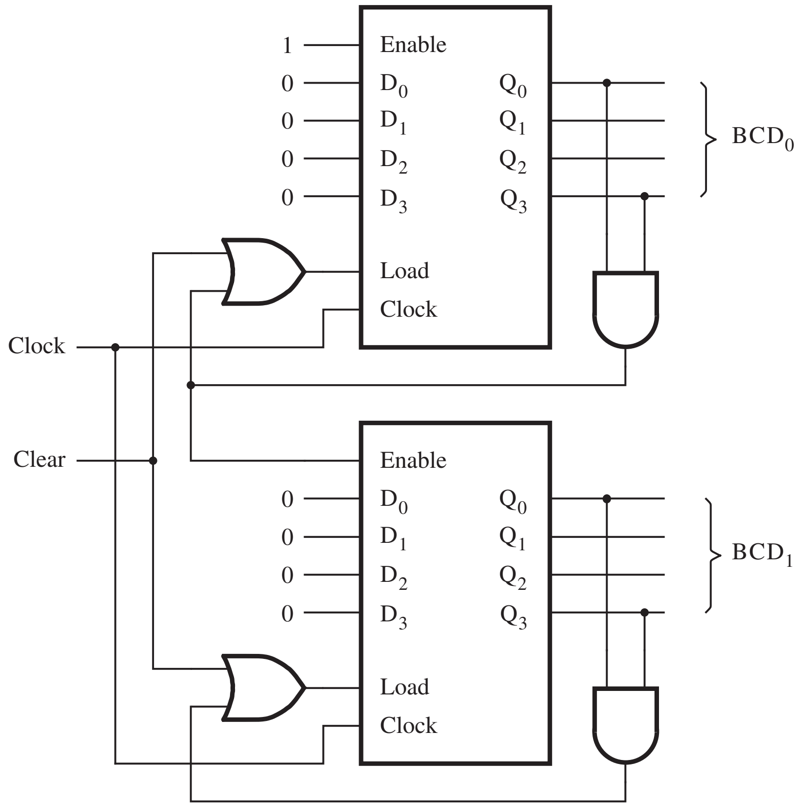


Figure 7.28 A two-digit BCD counter.

pulse arrives. This is accomplished by keeping the *Enable* signal for BCD_1 low at all times except when $BCD_0 = 9$.

In practice, it has to be possible to clear the contents of the counter by activating some control signal. Two OR gates are included in the circuit for this purpose. The control input *Clear* can be used to load 0s into the counter. Observe that in this case *Clear* is active when high. Verilog code for a two-digit BCD counter is given in Figure 7.81.

In any digital system there is usually one or more clock signals used to drive all synchronous circuitry. In the preceding counter, as well as in all counters presented in the previous figures, we have assumed that the objective is to count the number of clock pulses. Of course, these counters can be used to count the number of pulses in any signal that may be used in place of the clock signal.

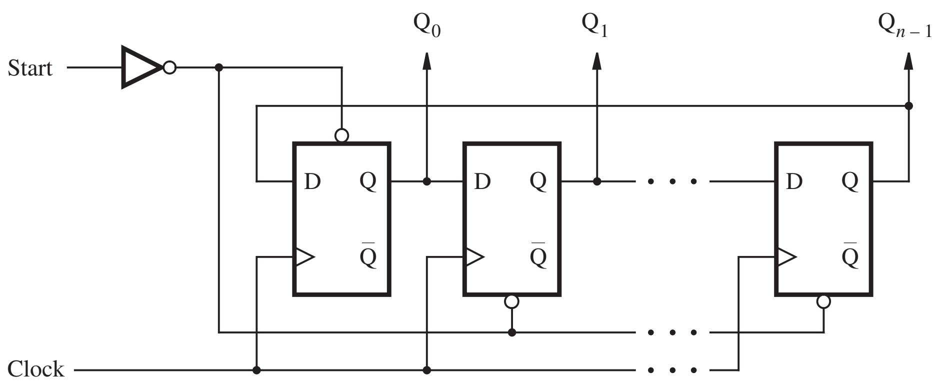
7.11.2 RING COUNTER

In the preceding counters the count is indicated by the state of the flip-flops in the counter. In all cases the count is a binary number. Using such counters, if an action is to be taken as a result of a particular count, then it is necessary to detect the occurrence of this count. This may be done using AND gates, as illustrated in Figures 7.26 through 7.28.

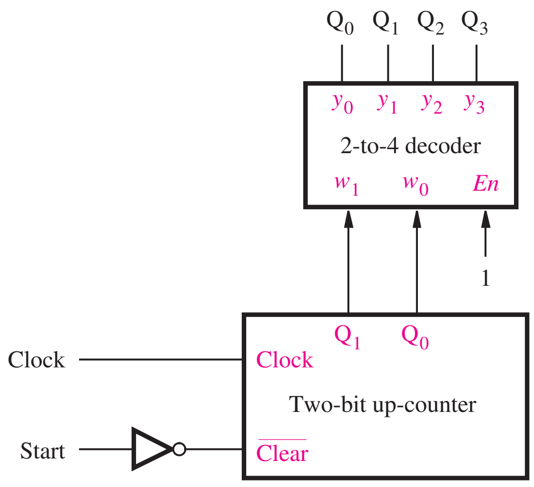
It is possible to devise a counterlike circuit in which each flip-flop reaches the state $Q_i = 1$ for exactly one count, while for all other counts $Q_i = 0$. Then Q_i indicates directly an occurrence of the corresponding count. Actually, since this does not represent binary numbers, it is better to say that the outputs of the flips-flops represent a code. Such a circuit can be constructed from a simple shift register, as indicated in Figure 7.29*a*. The Q output of the last stage in the shift register is fed back as the input to the first stage, which creates a ringlike structure. If a single 1 is injected into the ring, this 1 will be shifted through the ring at successive clock cycles. For example, in a four-bit structure, the possible codes $Q_0Q_1Q_2Q_3$ will be 1000, 0100, 0010, and 0001. As we said in section 6.2, such encoding, where there is a single 1 and the rest of the code variables are 0, is called a *one-hot code*.

The circuit in Figure 7.29*a* is referred to as a *ring counter*. Its operation has to be initialized by injecting a 1 into the first stage. This is achieved by using the *Start* control signal, which presets the left-most flip-flop to 1 and clears the others to 0. We assume that all changes in the value of the *Start* signal occur shortly after an active clock edge so that the flip-flop timing parameters are not violated.

The circuit in Figure 7.29*a* can be used to build a ring counter with any number of bits, *n*. For the specific case of $n = 4$, part (*b*) of the figure shows how a ring counter can be constructed using a two-bit up-counter and a decoder. When *Start* is set to 1, the counter is reset to 00. After *Start* changes back to 0, the counter increments its value in the normal way. The 2-to-4 decoder, described in section 6.2, changes the counter output into a one-hot code. For the count values 00, 01, 10, 11, 00, and so on, the decoder produces $Q_0Q_1Q_2Q_3 = 1000, 0100, 0010, 0001, 1000$, and so on. This circuit structure can be used for larger ring counters, as long as the number of bits is a power of two. We will give an example of a larger circuit that uses the ring counter in Figure 7.29*b* as a subcircuit in section 7.14.



(a) An *n*-bit ring counter



(b) A four-bit ring counter

Figure 7.29 Ring counter.

7.11.3 Johnson Counter

An interesting variation of the ring counter is obtained if, instead of the Q output, we take the $\overline{Q}$ output of the last stage and feed it back to the first stage, as shown in Figure 7.30. This circuit is known as a *Johnson counter*. An *n*-bit counter of this type generates a counting sequence of length 2*n*. For example, a four-bit counter produces the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, and so on. Note that in this sequence, only a single bit has a different value for two consecutive codes.

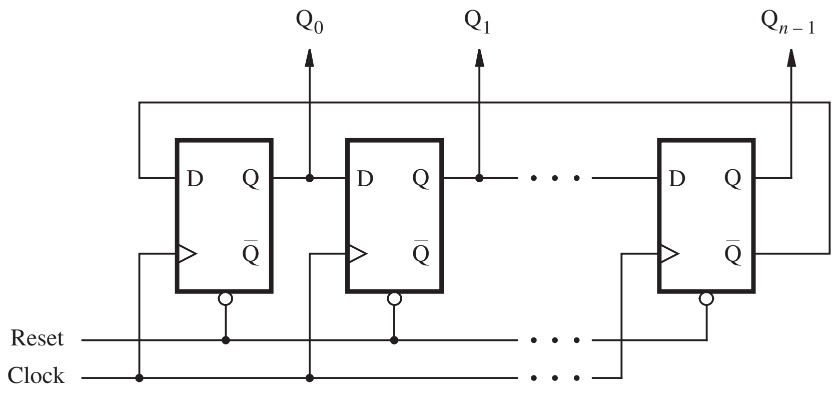


Figure 7.30 Johnson counter.

To initialize the operation of the Johnson counter, it is necessary to reset all flip-flops, as shown in the figure. Observe that neither the Johnson nor the ring counter will generate the desired counting sequence if not initialized properly.

7.11.4 Remarks on Counter Design

The sequential circuits presented in this chapter, namely, registers and counters, have a regular structure that allows the circuits to be designed using an intuitive approach. In Chapter 8 we will present a more formal approach to design of sequential circuits and show how the circuits presented in this chapter can be derived using this approach.

7.12 Using Storage Elements with CAD Tools

This section shows how circuits with storage elements can be designed using either schematic capture or Verilog code.

7.12.1 Including Storage Elements in Schematics

One way to create a circuit is to draw a schematic that builds latches and flip-flops from logic gates. Because these storage elements are used in many applications, most CAD systems provide them as prebuilt modules. Figure 7.31 shows a schematic created with a schematic capture tool, which includes three types of flip-flops that are imported from a library provided as part of the CAD system. The top element is a gated D latch, the middle element is a positive-edge-triggered D flip-flop, and the bottom one is a positive-edge-triggered T flip-flop. The D and T flip-flops have asynchronous, active-low clear and preset inputs. If these inputs are not connected in a schematic, then the CAD tool makes them inactive by assigning the default value of 1 to them.

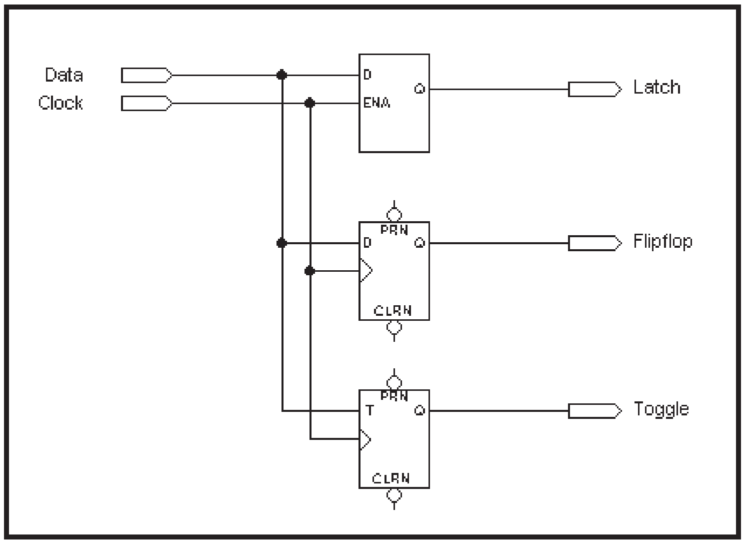


Figure 7.31 Three types of storage elements in a schematic.

When the gated D latch is synthesized for implementation in a chip, the CAD tool may not generate the cross-coupled NOR or NAND gates shown in section 7.2. In some chips, such as a CPLD, the AND-OR circuit depicted in Figure 7.32 may be preferable. This circuit is functionally equivalent to the cross-coupled version in section 7.2. The sum-of-products circuit is used because it is more suitable for implementation in a CPLD macrocell. One aspect of this circuit should be mentioned. From the functional point of view, it appears that the circuit can be simplified by removing the AND gate with the inputs *Data* and *Latch*. Without this gate, the top AND gate sets the value stored in the latch when the clock is 1, and the bottom AND gate maintains the stored value when the clock is 0. But without this gate, the circuit has a timing problem known as a *static hazard*. A detailed explanation of hazards will be given in section 9.6.

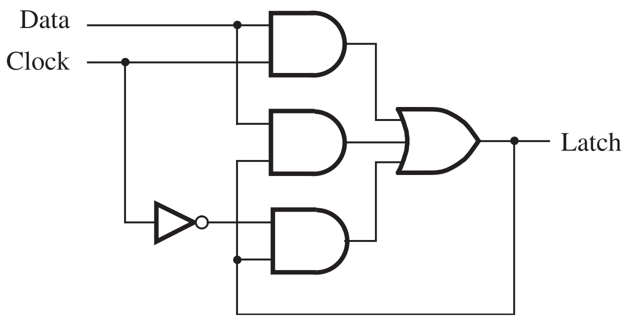


Figure 7.32 Gated D latch generated by CAD tools.

The circuit in Figure 7.31 can be implemented in a CPLD as shown in Figure 7.33. The D and T flip-flops are realized using the flip-flops on the chip that are configurable as either D or T types. The figure depicts in blue the gates and wires needed to implement the circuit in Figure 7.31.

The results of a timing simulation for the implementation in Figure 7.33 are given in Figure 7.34. The *Latch* signal, which is the output of the gated D latch, implemented as indicated in Figure 7.32, follows the *Data* input whenever the *Clock* signal is 1. Because

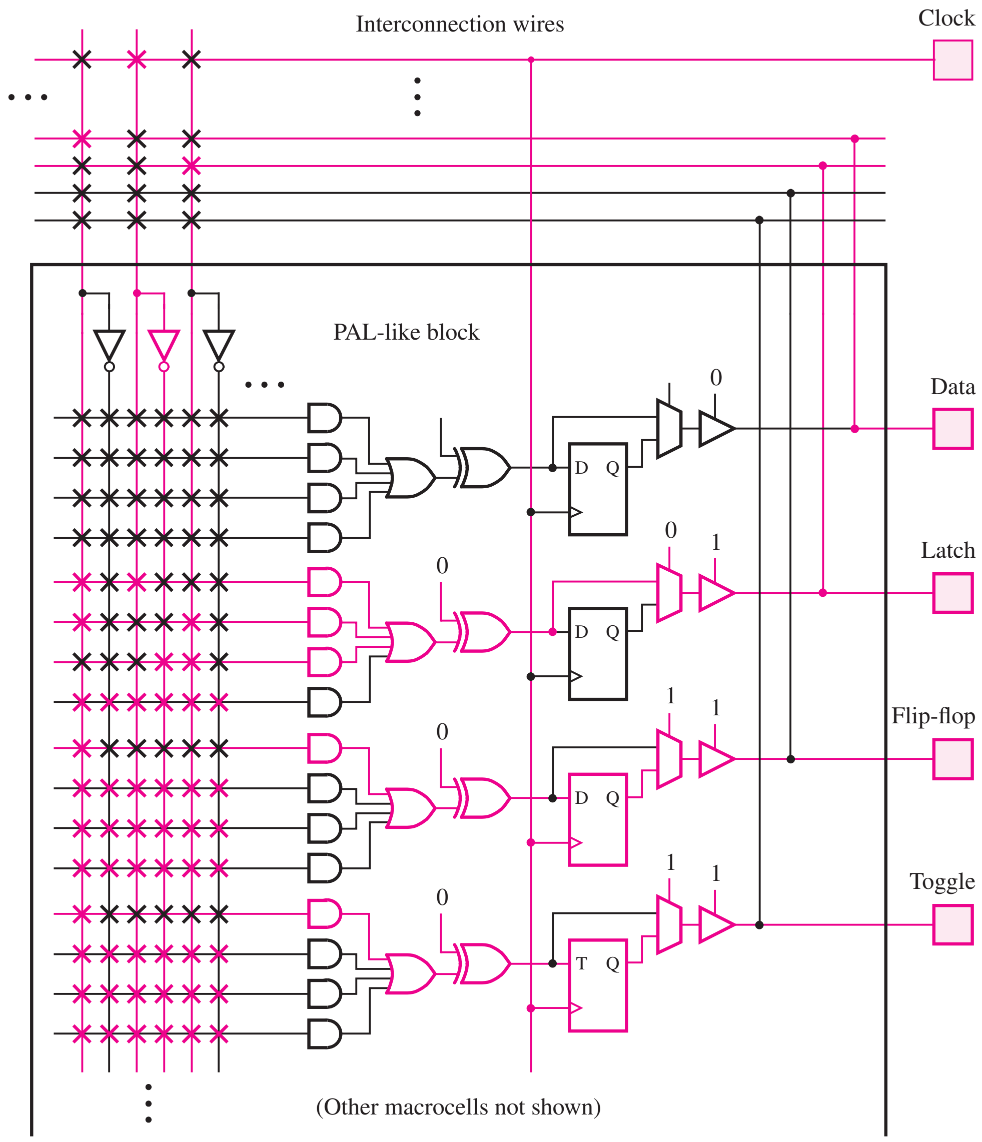


Figure 7.33 Implementation of the schematic in Figure 7.31 in a CPLD.

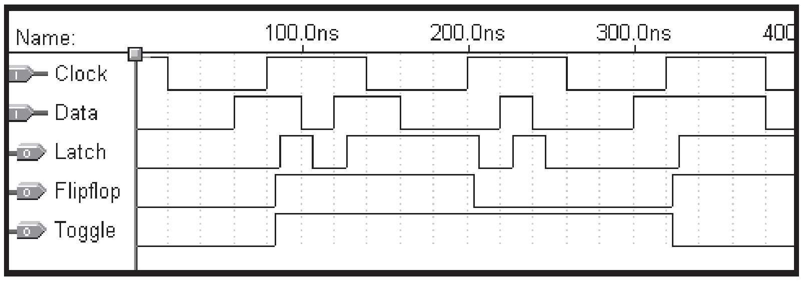


Figure 7.34 Timing simulation for the storage elements in Figure 7.31.

of propagation delays in the chip, the *Latch* signal is delayed in time with respect to the *Data* signal. Since the *Flipflop* signal is the output of the D flip-flop, it changes only after a positive clock edge. Similarly, the output of the T flip-flop, called *Toggle* in the figure, toggles when $Data = 1$ and a positive clock edge occurs. The timing diagram illustrates the delay from when the positive clock edge occurs at the input pin of the chip until a change in the flip-flop output appears at the output pin of the chip. This time is called the *clock-to-output time*, t_{co}.

7.12.2 Using Verilog Constructs for Storage Elements

In section 6.6 we described a number of Verilog constructs. We now show how these constructs can be used to describe storage elements.

A simple way of specifying a storage element is by using the **if-else** statement to describe the desired behavior responding to changes in the levels of data and clock inputs. Consider the **always** block

```
always @(Control or B)
    if (Control)
        A = B;
```

where *A* is a variable of **reg** type. This code specifies that the value of *A* should be made equal to the value of *B* when $Control = 1$. But the statement does not indicate an action that should occur when $Control = 0$. In the absence of an assigned value, the Verilog compiler assumes that the value of *A* caused by the **if** statement must be maintained until the next time this **if** statement is evaluated. This notion of *implied memory* is realized by instantiating a latch in the circuit.

Example 7.1 **CODE FOR A GATED D LATCH** The code in Figure 7.35 defines a module named *D_latch*, which has the inputs *D* and *Clk* and the output Q. The **if** clause defines that the Q output must take the value of *D* when $Clk = 1$. Since no **else** clause is given, a latch will be synthesized to maintain the value of Q when $Clk = 0$. Therefore, the code describes a gated

```
module D_latch (D, Clk, Q);
   input D, Clk;
   output Q;
   reg Q;

   always @(D or Clk)
      if (Clk)
         Q = D;

endmodule
```

Figure 7.35 Code for a gated D latch.

D latch. The sensitivity list includes *Clk* and *D* because both of these signals can cause a change in the value of the Q output.

An **always** construct is used to define a circuit that responds to changes in the signals that appear in the sensitivity list. While in the examples presented so far the **always** blocks are sensitive to the *levels* of signals, it is also possible to specify that a response should take place only at a particular edge of a signal. The desired edge is specified by using the Verilog keywords **posedge** and **negedge**, which are used to implement edge-triggered circuits.

Example 7.2

CODE FOR A D FLIP-FLOP Figure 7.36 defines a module named *flipflop*, which is a positive-edge-triggered D flip-flop. The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output. The keyword **posedge** specifies that a change may occur only on the positive edge of *Clock*. At this time the output

```
module flipflop (D, Clock, Q);
   input D, Clock;
   output Q;
   reg Q;

   always @(posedge Clock)
      Q = D;

endmodule
```

Figure 7.36 Code for a D flip-flop.

Q is set to the value of the input *D*. Since Q is of **reg** type it will maintain its value between the positive edges of the clock.

7.12.3 BLOCKING AND NON-BLOCKING ASSIGNMENTS

In all our Verilog examples presented so far we have used the equal sign for assignments, as in

f = x1 & x2;

or

C = A + B;

or

Q = D;

This notation is called a *blocking* assignment. A Verilog compiler evaluates the statements in an **always** block in the order in which they are written. If a variable is given a value by a blocking assignment statement, then this new value is used in evaluating all subsequent statements in the block.

Example 7.3 Consider the code in Figure 7.37. Since the **always** block is sensitive to the positive clock edge, both Q1 and Q2 will be implemented as the outputs of D flip-flops. However, because blocking assignments are involved, these two flip-flops will not be connected in cascade, as the reader might expect. The first statement

Q1 = D;

sets Q1 to the value of *D*. This new value is used in evaluating the subsequent statement

```
module example7_3 (D, Clock, Q1, Q2);
   input D, Clock;
   output Q1, Q2;
   reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 = D;
      Q2 = Q1;
   end

endmodule
```

Figure 7.37 Incorrect code for two cascaded flip-flops.

Q2 = Q1;

which results in Q2 = Q1 = D. The synthesized circuit has two parallel flip-flops, as illustrated in Figure 7.38. A synthesis tool will likely delete one of these redundant flip-flops as an optimization step.

Verilog also provides a *non-blocking* assignment, denoted with <=. All non-blocking assignment statements in an **always** block are evaluated using the values that the variables have when the **always** block is entered. Thus, a given variable has the same value for all statements in the block. The meaning of non-blocking is that the result of each assignment is not seen until the end of the **always** block.

Example 7.4

Figure 7.39 gives the same code as in Figure 7.37, but using non-blocking assignments. In the two statements

Q1 <= D;
Q2 <= Q1;

the variables Q1 and Q2 have some value at the start of evaluating the **always** block, and then they change to a new value concurrently at the end of the **always** block. This code generates a cascaded connection between flip-flops, which implements the shift register depicted in Figure 7.40.

The differences between blocking and non-blocking assignments are illustrated further by the following two examples.

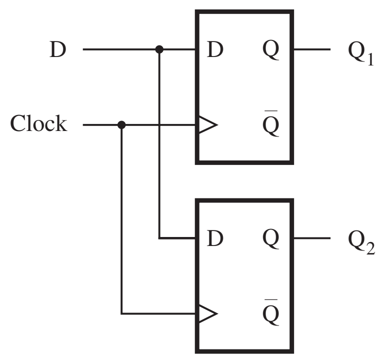


Figure 7.38 Circuit for Example 7.3.

```
module example7_4 (D, Clock, Q1, Q2);
   input D, Clock;
   output Q1, Q2;
   reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 <= D;
      Q2 <= Q1;
   end

endmodule
```

Figure 7.39 Code for two cascaded flip-flops.

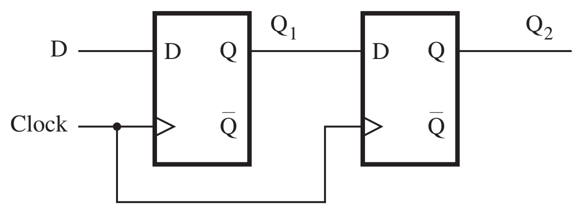


Figure 7.40 Circuit defined in Figure 7.39.

Example 7.5 Code that involves some gates in addition to flip-flops is defined in Figure 7.41 using blocking assignment statements. The resulting circuit is given in Figure 7.42. Both f and g are implemented as the outputs of D flip-flops, because the sensitivity list of the **always** block specifies the event **posedge** Clock. Since blocking assignments are used, the updated value of f generated by the statement f = x1 & x2 has to be seen immediately by the following statement g = f | x3. Thus, the AND gate that produces x1 & x2 is connected to the OR gate that feeds the g flip-flop, as shown in Figure 7.42.

Example 7.6 If non-blocking assignments are used, as given in Figure 7.43, then both f and g are updated simultaneously. Hence, the previous value of f is used in updating the value of g, which means that the output of the flip-flop that generates f is connected to the OR gate that feeds the g flip-flop. This gives rise to the circuit in Figure 7.44.

It is interesting to consider what circuit would be synthesized if the statements that specify f and g were reversed. For the code in Figure 7.41 the impact would be significant. If g is evaluated first, then the second statement does not depend on the first one, because f does not depend on g. The resulting circuit would be the same as the one in Figure 7.44.

```
module example7_5 (x1, x2, x3, Clock, f, g);
   input x1, x2, x3, Clock;
   output f, g;
   reg f, g;

   always @(posedge Clock)
   begin
      f = x1 & x2;
      g = f | x3;
   end

endmodule
```

Figure 7.41 Code for Example 7.5.

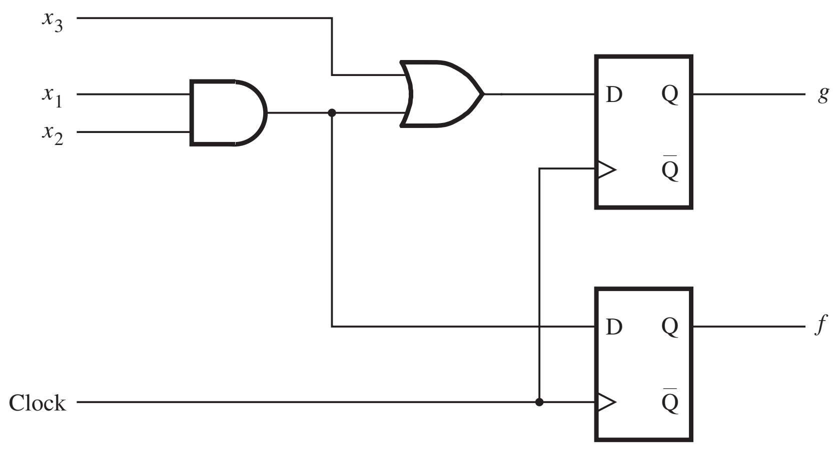


Figure 7.42 Circuit for Example 7.5.

```
module example7_6 (x1, x2, x3, Clock, f, g);
   input x1, x2, x3, Clock;
   output f, g;
   reg f, g;

   always @(posedge Clock)
   begin
      f <= x1 & x2;
      g <= f | x3;
   end

endmodule
```

Figure 7.43 Code for Example 7.6.

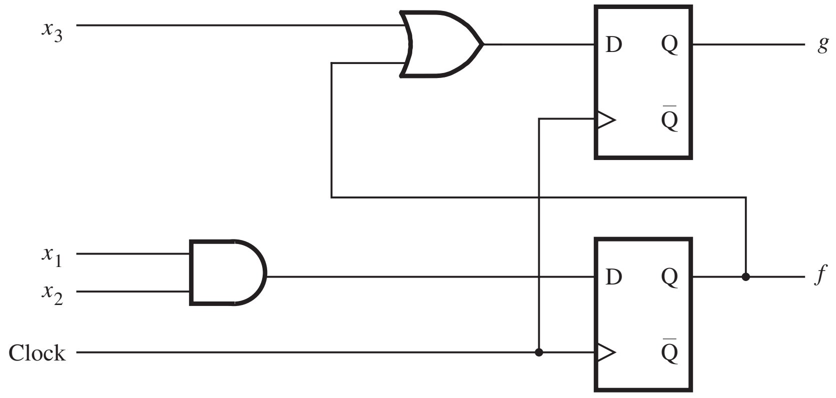


Figure 7.44 Circuit for Example 7.6.

Reversing the statement order would make no difference for the code in Figure 7.43, in which the non-blocking assignment is used.

The use of blocking assignments for sequential circuits can easily lead to wrong results, as demonstrated in Figure 7.38. The dependence on ordering of blocking assignments is dangerous, as shown in the previous example. It is better to use non-blocking assignments to describe sequential circuits.

7.12.4 Non-Blocking Assignments for Combinational Circuits

A natural question at this point is whether non-blocking assignments can be used for combinational circuits. The answer is that they can be used in most situations, but when subsequent assignments in an **always** block depend on the results of previous assignments, the non-blocking assignments can generate nonsensical circuits. As an example, assume that we have a three-bit vector $A = a_2a_1a_0$, and we wish to generate a combinational function f that is equal to 1 when there are two adjacent bits in A that have the value 1. One way to specify this function with blocking assignments is

```
always @(A)
begin
   f = A[1] & A[0];
   f = f | (A[2] & A[1]);
end
```

These statements produce the desired logic function, which is $f = a_1a_0 + a_2a_1$. Consider now changing the code to use the non-blocking assignments

```
f <= A[1] & A[0];
f <= f | (A[2] & A[1]);
```

There are two key aspects of the Verilog semantics relevant to this code:

1. The results of non-blocking assignments are visible only after all of the statements in the **always** block have been evaluated.
2. When there are multiple assignments to the same variable inside an **always** block, the result of the last assignment is maintained.

In this example, f has an unspecified initial value when we enter the **always** block. The first statement assigns $f = a_1a_0$, but this result is not visible to the second statement. It still sees the original unspecified value of f. The second assignment overrides (deletes!) the first assignment and produces the logic function $f = f + a_2a_1$. This expression does not correspond to a combinational circuit, because it represents an AND-OR circuit in which the OR-gate is fed back to itself. It is best to use blocking assignments when describing combinational circuits, so as to avoid accidentally creating a sequential circuit.

7.12.5 Flip-Flops with Clear Capability

By using a particular sensitivity list and a specific style of **if-else** statement, it is possible to include clear (or preset) signals on flip-flops.

ASYNCHRONOUS CLEAR Figure 7.45 gives a module that defines a D flip-flop with an asynchronous active-low reset (clear) input. When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0. Note that the sensitivity list specifies the negative edge of *Resetn* as an event trigger along with the positive edge of the clock. We cannot omit the keyword **negedge** because the sensitivity list cannot have both edge-triggered and level-sensitive signals. **Example 7.7**

```
module flipflop (D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output Q;
   reg Q;

   always @(negedge Resetn or posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure 7.45 D flip-flop with asynchronous reset.

```
module flipflop (D, Clock, Resetn, Q);
    input D, Clock, Resetn;
    output Q;
    reg Q;

    always @(posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;

endmodule
```

Figure 7.46 D flip-flop with synchronous reset.

Example 7.8 **SYNCHRONOUS CLEAR** Figure 7.46 shows how a D flip-flop with a synchronous reset input can be described. In this case the reset signal is acted upon only when a positive clock edge arrives. This code generates the circuit in Figure 7.15, which has an AND gate connected to the flip-flop's D input.

7.13 Using Registers and Counters with CAD Tools

In this section we show how registers and counters can be included in circuits designed with the aid of CAD tools. Examples are given using both schematic capture and Verilog code.

7.13.1 Including Registers and Counters in Schematics

In section 5.5.1 we explained that a CAD system usually includes libraries of prebuilt subcircuits. We introduced the library of parameterized modules (LPM) and used the adder/subtractor module, *lpm_add_sub*, as an example. The LPM includes modules that constitute flip-flops, registers, counters, and many other useful circuits. Figure 7.47 shows a symbol that represents the *lpm_ff* module. This module is a register with one or more positive-edge-triggered flip-flops that can be of either D or T type. The module has parameters that allow the number of flip-flops and flip-flop type to be chosen. In this case we chose to have four D flip-flops. The tutorial in Appendix D explains how the configuration of the module is done.

The D inputs to the four flip-flops, called *data* on the graphical symbol, are connected to the four-bit input signal *Data*[3..0]. The module's asynchronous active-high reset (clear) input, *aclr*, is shown in the schematic. The flip-flop outputs, *q*, are attached to the output symbol labeled Q[3..0].

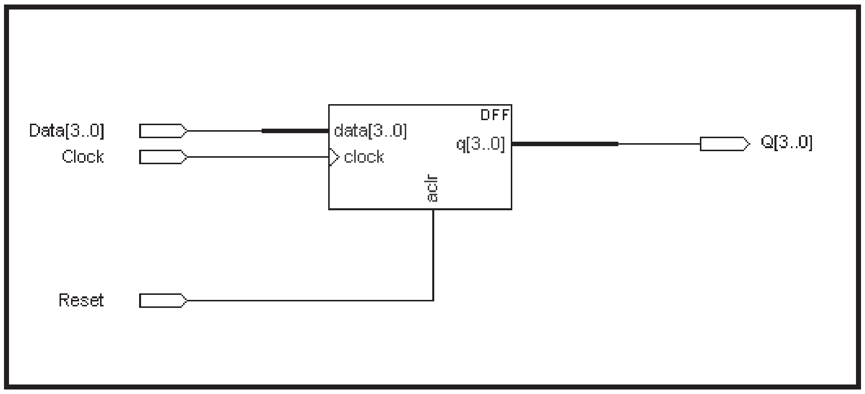


Figure 7.47 The *lpm_ff* parameterized flip-flop module.

In section 7.3 we said that a useful application of D flip-flops is to hold the results of an arithmetic computation, such as the output from an adder circuit. An example is given in Figure 7.48, which uses two LPM modules, *lpm_add_sub* and *lpm_ff*. The *lpm_add_sub* module was described in section 5.5.1. Its parameters, which are not shown in Figure 7.48, are set to configure the module as a four-bit adder circuit. The adder's four-bit data input *dataa* is driven by the *Data*[3..0] input signal. The sum bits, *result*, are connected to the *data* inputs of the *lpm_ff*, which is configured as a four-bit D register with asynchronous clear. The register generates the output of the circuit, Q[3..0], which appears on the left side of the schematic. This signal is fed back to the *datab* input of the adder. The sum bits

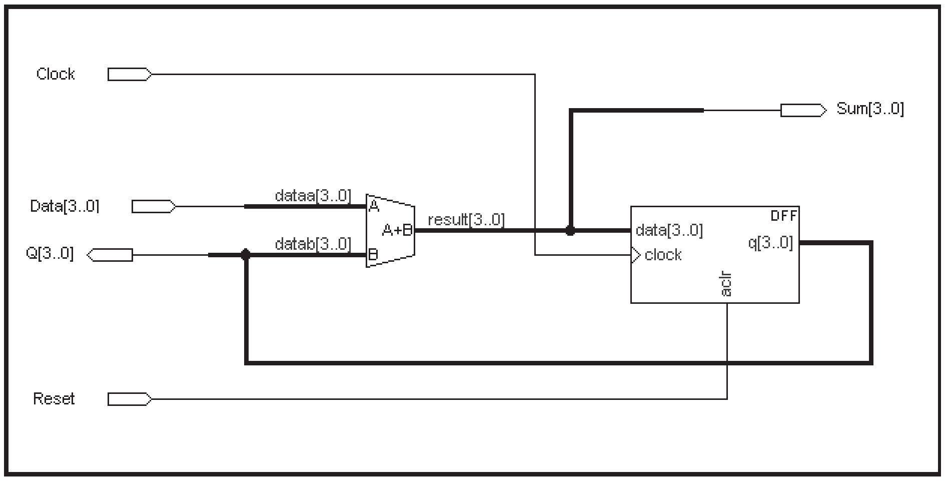


Figure 7.48 An adder with registered feedback.

from the adder are also provided as an output of the circuit, *Sum*[3..0], for ease of reference in the discussion that follows. If the register is first cleared to 0000, then the circuit can be used to add the binary numbers on the *Data*[3..0] input to a sum that is being accumulated in the register, if a new number is applied to the input during each clock cycle. A circuit that performs this function is referred to as an *accumulator* circuit.

We synthesized a circuit from the schematic and implemented the four-bit adder using the carry-lookahead structure. A timing simulation for the circuit appears in Figure 7.49. After resetting the circuit, the *Data* input is set to 0001. The adder produces the sum $0000 + 0001 = 0001$, which is then clocked into the register at the 60 ns point in time. After the t_{co} delay, Q[3..0] becomes 0001, and this causes the adder to produce the new sum $0001 + 0001 = 0010$. The time needed to generate the new sum is determined by the speed of the adder circuit, which produces the sum after 12.5 ns in this case. The new sum does not appear at the Q output until after the next positive clock edge, at 100 ns. The adder then produces 0011 as the next sum. When *Sum* changes from 0010 to 0011, some oscillations appear in the timing diagram, caused by the propagation of carry signals through the adder circuit. These oscillations are not seen at the Q output, because *Sum* is stable by the time the next positive clock edge occurs. Moving forward to the 180 ns point in time, $Sum = 0100$, and this value is clocked into the register. The adder produces the new sum 0101. Then at 200 ns *Data* is changed to 0010, which causes the sum to change to $0100 + 0010 = 0110$. At the next positive clock edge, Q is set to 0110; the value $Sum = 0101$ that was present temporarily in the circuit is not observed at the Q output. The circuit continues to add 0010 to the Q output at each successive positive clock edge.

Having simulated the behavior of the circuit, we should consider whether or not we can conclude with some certainty that the circuit works properly. Ideally, it is prudent to test all possible combinations of a circuit's inputs before declaring that it works as desired. However, in practice, such testing is often not feasible because of the number of input combinations that exist. For the circuit in Figure 7.48, we could verify that a correct sum is produced by the adder, and we could also check that each of the four flip-flops in the register properly stores either 0 or 1. We will discuss issues associated with the testing of circuits in Chapter 11.

For the circuit in Figure 7.48 to work properly, the following timing constraints must be met. When the register is clocked by a positive clock edge, a change of signal value

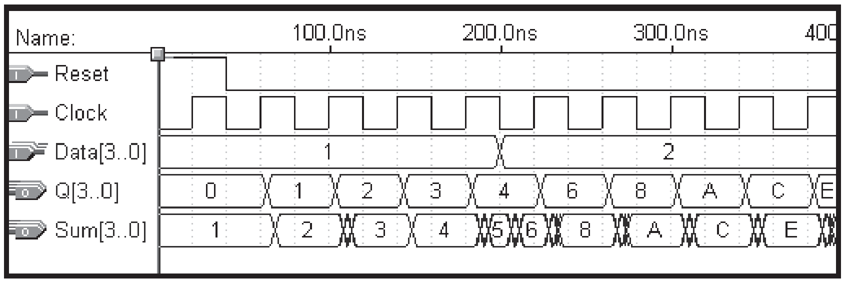


Figure 7.49 Timing simulation of the circuit from Figure 7.48.

at the register's output must propagate through the feedback path to the *datab* input of the adder. The adder then produces a new sum, which must propagate to the *data* input of the register. For the chip used to implement the circuit, the total delay incurred is 14 ns. The delay can be broken down as follows: It takes 2 ns from when the register is clocked until a change in its output reaches the *datab* input of the adder. The adder produces a new sum in 8 ns, and it takes 4 ns for the sum to propagate to the register's *data* input. In Figure 7.49 the clock period is 40 ns. Hence, after the new sum arrives at the *data* input of the register, there remain $40 - 14 = 26$ ns until the next positive clock edge occurs. The *data* input must be stable for the amount of the setup time, $t_{su} = 3$ ns, before the clock edge. Hence we have $26 - 3 = 23$ ns to spare. The clock period can be decreased by as much as 23 ns and the circuit will still work. But if the clock period is less than $40 - 23 = 17$ ns, then the circuit will not function properly. Of course, if a different chip were used to implement the circuit, then different timing results would be produced. CAD systems provide tools that can automatically determine the minimum allowable clock period for which a circuit will work correctly. The tutorial in Appendix D shows how this is done using the tools that accompany the book.

7.13.2 Using Library Modules in Verilog Code

The predefined subcircuits in a library of modules such as the LPM library can be instantiated in Verilog code. Figure 7.50 instantiates the *lpm_shiftreg* module, which is an *n*-bit shift register. The module's parameters are set using **defparam** statements. The number of flip-flops in the shift register is set to 4 using the parameter lpm_width = 4. The module can be configured to shift either left or right. The parameter lpm_direction = "RIGHT" sets the shift direction to be from the left to the right. The code uses the module's asynchronous active-high clear input, *aclr*, and the active-high parallel-load input, *load*, which allows the shift register to be loaded with the parallel data on the module's *data* input. When shifting takes place, the value on the *shiftin* input is shifted into the left-most flip-flop and the bit shifted out appears on the right-most bit of the *q* parallel output. The code uses *named*

```
module shift (Clock, Reset, w, Load, R, Q);
   input Clock, Reset, w, Load;
   input [3:0] R;
   output [3:0] Q ;

   lpm_shiftreg shift_right (.data(R), .aclr(Reset), .clock(Clock),
      .load(Load), .shiftin(w), .q(Q)) ;
      defparam shift_right.lpm_width = 4;
      defparam shift_right.lpm_direction = "RIGHT";

endmodule
```

Figure 7.50 Instantiation of the *lpm_shiftreg* module.

ports to connect the input and output signals of the *shift* module to the ports of the module. For example, the *R* input signal is connected to the module's *data* port. This is specified by writing .data(R) in the instantiation statement. Similarly, .aclr(Reset) specifies that the *Reset* input signal is connected to the *aclr* port on the module, and so on. When translated into a circuit, the *lpm_shiftreg* has the structure shown in Figure 7.19.

Predefined modules also exist for the various types of counters, which are commonly needed in logic circuits. An example is the *lpm_counter* module, which is a variable-width counter with parallel-load inputs.

7.13.3 Using Verilog Constructs for Registers and Counters

Rather than instantiating predefined subcircuits for registers, shift registers, counters, and the like, the circuits can be described in Verilog code. Figure 7.45 gives code for a D flip-flop. One way to describe an *n*-bit register is to write hierarchical code that includes *n* instances of the D flip-flop subcircuit. A simpler approach is to use the same code as in Figure 7.45 and define the *D* input and Q output as multibit signals.

Example 7.9 **AN N-BIT REGISTER** Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register module for which the number of flip-flops can be easily changed. The code for an *n*-bit register is given in Figure 7.51. The parameter *n* specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size.

```
module regn (D, Clock, Resetn, Q);
   parameter n = 16;
   input [n–1:0] D;
   input Clock, Resetn;
   output [n–1:0] Q;
   reg [n–1:0] Q;

   always @(negedge Resetn or posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure 7.51 Code for an *n*-bit register with asynchronous clear.

Example 7.10

A FOUR-BIT SHIFT REGISTER Assume that we wish to write Verilog code that represents the four-bit parallel-access shift register in Figure 7.19. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the *D* input. Figure 7.52 defines the module named *muxdff*, which represents this subcircuit. The two data inputs are named D_0 and D_1, and they are selected using the *Sel* input. The **if-else** statement specifies that on the positive clock edge if *Sel* = 0, then Q is assigned the value of D_0; otherwise, Q is assigned the value of D_1.

Figure 7.53 defines the four-bit shift register. The module *Stage3* instantiates the left-most flip-flop, which has the output Q_3, and the module *Stage0* instantiates the right-most flip-flop, Q_0. When $L = 1$, the register is loaded in parallel from the *R* input; and when $L = 0$, shifting takes place in the left to right direction. Serial data is shifted into the most-significant bit, Q_3, from the *w* input.

```
module muxdff (D0, D1, Sel, Clock, Q);
  input D0, D1, Sel, Clock;
  output Q;
  reg Q;

  always @(posedge Clock)
    if (!Sel)
      Q <= D0;
    else
      Q <= D1;

endmodule
```

Figure 7.52 Code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```
module shift4 (R, L, w, Clock, Q);
  input [3:0] R;
  input L, w, Clock;
  output [3:0] Q;
  wire [3:0] Q;

  muxdff Stage3 (w, R[3], L, Clock, Q[3]);
  muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
  muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
  muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endmodule
```

Figure 7.53 Hierarchical code for a four-bit shift register.

Example 7.11 **ALTERNATIVE CODE FOR A FOUR-BIT SHIFT REGISTER** A different style of code for the four-bit shift register is given in Figure 7.54. Instead of using subcircuits, the shift register is defined using the approach presented in Example 7.4. All actions take place at the positive edge of the clock. If $L = 1$, the register is loaded in parallel with the four bits of input R. If $L = 0$, the contents of the register are shifted to the right and the value of the input w is loaded into the most-significant bit Q_3.

Example 7.12 **AN N-BIT SHIFT REGISTER** Figure 7.55 shows the code that can be used to represent shift registers of any size. The parameter n, which has the default value 16 in the figure, sets the number of flip-flops. The code is identical to that in Figure 7.54 with two exceptions. First, R and Q are defined in terms of n. Second, the **else** clause that describes the shifting operation is generalized to work for any number of flip-flops by using a **for** loop.

Example 7.13 **UP-COUNTER** Figure 7.56 represents a four-bit up-counter with a reset input, *Resetn*, and an enable input, E. The outputs of the flip-flops in the counter are represented by the vector named Q. The **if** statement specifies an asynchronous reset of the counter if $Resetn = 0$. The **else if** clause specifies that if $E = 1$ the count is incremented on the positive clock edge.

```
module shift4 (R, L, w, Clock, Q);
   input [3:0] R;
   input L, w, Clock;
   output [3:0] Q;
   reg [3:0] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] <= Q[1];
         Q[1] <= Q[2];
         Q[2] <= Q[3];
         Q[3] <= w;
      end

endmodule
```

Figure 7.54 Alternative code for a four-bit shift register.

```
module shiftn (R, L, w, Clock, Q);
    parameter n = 16;
    input [n–1:0] R;
    input L, w, Clock;
    output [n–1:0] Q;
    reg [n–1:0] Q;
    integer k;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
        begin
            for (k = 0; k < n–1; k = k+1)
                Q[k] <= Q[k+1];
            Q[n–1] <= w;
        end

endmodule
```

Figure 7.55 An *n*-bit shift register.

```
module upcount (Resetn, Clock, E, Q);
    input Resetn, Clock, E;
    output [3:0] Q;
    reg [3:0] Q;

    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            Q <= 0;
        else if (E)
            Q <= Q + 1;

endmodule
```

Figure 7.56 Code for a four-bit up-counter.

Example 7.14

UP-COUNTER WITH PARALLEL LOAD The code in Figure 7.57 defines an up-counter that has a parallel-load input in addition to a reset input. The parallel data is provided as the input vector R. The first **if** statement provides the same asynchronous reset as in Figure 7.56. The **else if** clause specifies that if $L = 1$ the flip-flops in the counter are loaded in

```
module upcount (R, Resetn, Clock, E, L, Q);
    input [3:0] R;
    input Resetn, Clock, E, L;
    output [3:0] Q;
    reg [3:0] Q;

    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            Q <= 0;
        else if (L)
            Q <= R;
        else if (E)
            Q <= Q + 1;

endmodule
```

Figure 7.57 A four-bit up-counter with a parallel load.

parallel from the *R* inputs on the positive clock edge. If $L = 0$, the count is incremented, under control of the enable input *E*.

Example 7.15 **DOWN-COUNTER WITH PARALLEL LOAD** Figure 7.58 shows the code for a down-counter named *downcount*. A down-counter is normally used by loading it with some starting count and then decrementing its contents. The starting count is represented in the code by the vector *R*. On the positive clock edge, if $L = 1$ the counter is loaded with the input *R*, and if $L = 0$ the count is decremented. The counter also includes an enable input, *E*. Setting

```
module downcount (R, Clock, E, L, Q);
    parameter n = 8;
    input [n−1:0] R;
    input Clock, L, E;
    output [n−1:0] Q;
    reg [n−1:0] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else if (E)
            Q <= Q − 1;

endmodule
```

Figure 7.58 A down-counter with a parallel load.

```
module updowncount (R, Clock, L, E, up_down, Q);
  parameter n = 8;
  input [n−1:0] R;
  input Clock, L, E, up_down;
  output [n−1:0] Q;
  reg [n−1:0] Q;
  integer direction;

  always @(posedge Clock)
  begin
    if (up_down)
      direction = 1;
    else
      direction = −1;
    if (L)
      Q <= R;
    else if (E)
      Q <= Q + direction;
  end

endmodule
```

Figure 7.59 Code for an up/down counter.

$E = 0$ prevents the contents of the flip-flops from changing when an active clock edge occurs.

Example 7.16

UP/DOWN COUNTER Verilog code for an up/down counter is given in Figure 7.59. This module combines the capabilities of the counters defined in Figures 7.57 and 7.58. It includes a control signal *up_down* that governs the direction of counting. It also includes an **integer** variable named *direction*, which is equal to 1 for up-count and equal to −1 for down-count.

7.14 Design Examples

This section presents examples of digital systems that make use of some of the building blocks described in this chapter and in Chapter 6.

7.14.1 Bus Structure

Digital systems often contain a set of registers used to store data. Figure 7.60 gives an example of a system that has k n-bit registers, $R1$ to Rk. Each register is connected to a

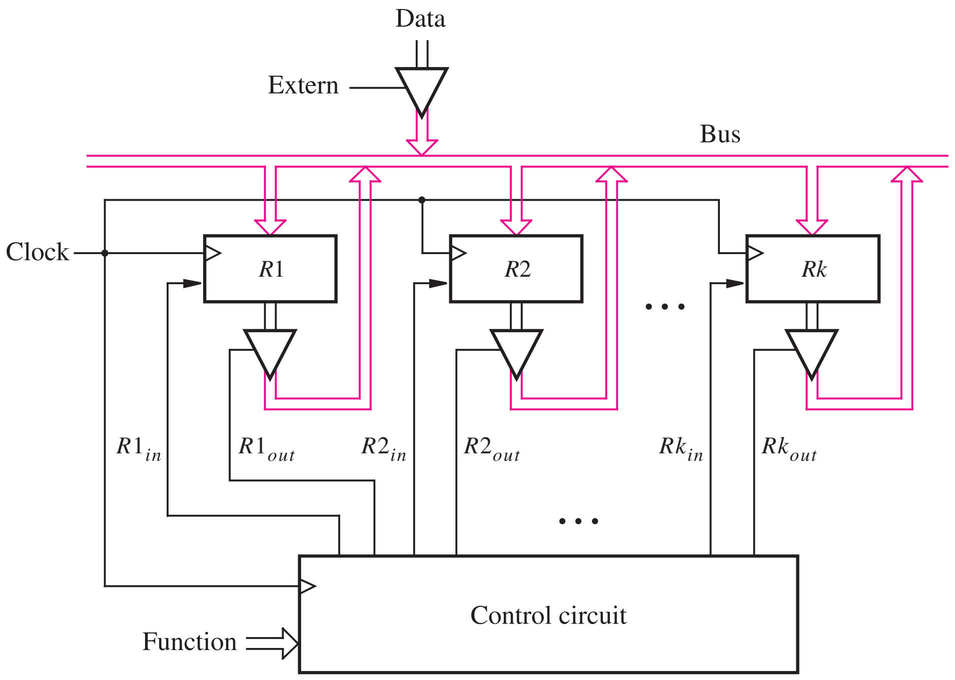


Figure 7.60 A digital system with k registers.

common set of n wires, which are used to transfer data into and out of the registers. This common set of wires is usually called a *bus*. In addition to registers, in a real system other types of circuit blocks would be connected to the bus. The figure shows how n bits of data can be placed on the bus from another circuit block, using the control input *Extern*. The data stored in any of the registers can be transferred via the bus to a different register or to another circuit block that is connected to the bus.

It is essential to ensure that only one circuit block attempts to place data onto the bus wires at any given time. In Figure 7.60 each register is connected to the bus through an n-bit tri-state buffer. A control circuit is used to ensure that only one of the tri-state buffer enable inputs, $R1_{out}, \ldots, Rk_{out}$, is asserted at a given time. The control circuit also produces the signals $R1_{in}, \ldots, Rk_{in}$, which control when data is loaded into each register. In general, the control circuit could perform a number of functions, such as transferring the data stored in one register into another register and the like. Figure 7.60 shows an input signal named *Function* that instructs the control circuit to perform a particular task. The control circuit is synchronized by a clock input, which is the same clock signal that controls the k registers.

Figure 7.61 provides a more detailed view of how the registers from Figure 7.60 can be connected to a bus. To keep the picture simple, 2 two-bit registers are shown, but the same scheme can be used for larger registers. For register $R1$, two tri-state buffers enabled by $R1_{out}$ are used to connect each flip-flop output to a wire in the bus. The D input on

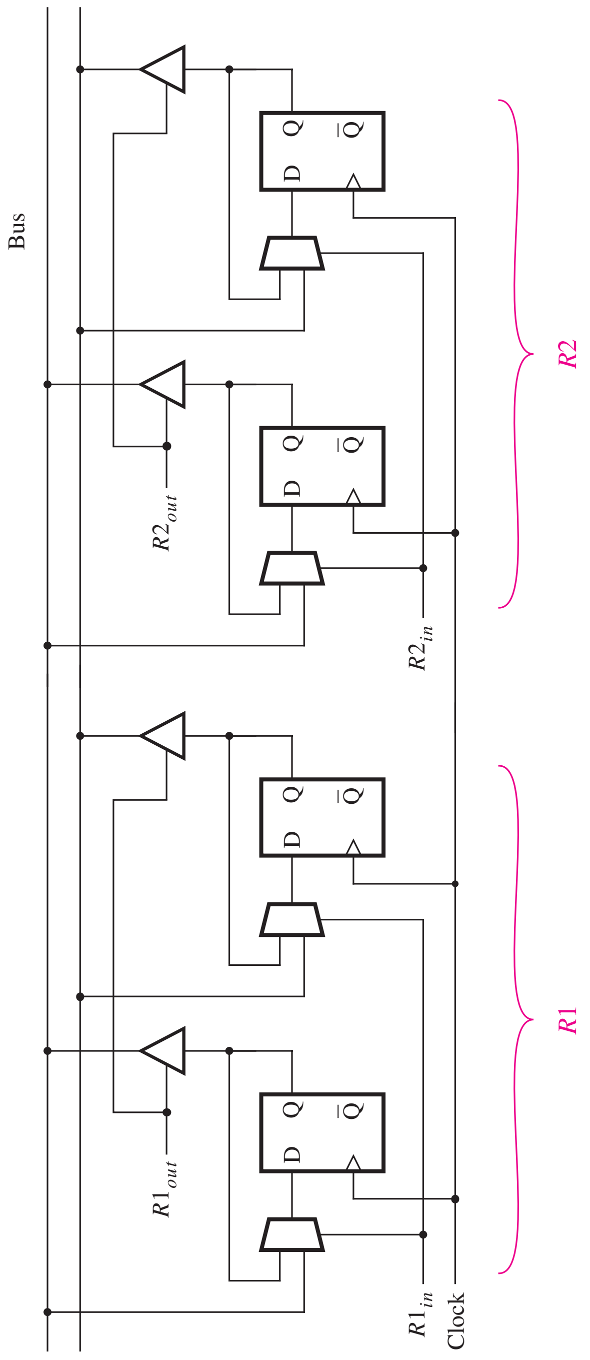


Figure 7.61 Details for connecting registers to a bus.

each flip-flop is connected to a 2-to-1 multiplexer, whose select input is controlled by $R1_{in}$. If $R1_{in} = 0$, the flip-flops are loaded from their Q outputs; hence the stored data does not change. But if $R1_{in} = 1$, data is loaded into the flip-flops from the bus. Instead of using multiplexers on the flip-flop inputs, one could attempt to connect the D inputs on the flip-flops directly to the bus. Then it is necessary to control the clock inputs on all flip-flops to ensure that they are clocked only when new data should be loaded into the register. This approach is not good because it may happen that different flip-flops will be clocked at slightly different times, leading to a problem known as *clock skew*. A detailed discussion of the issues related to the clocking of flip-flops is provided in section 10.3.

The system in Figure 7.60 can be used in many different ways, depending on the design of the control circuit and on how many registers and other circuit blocks are connected to the bus. As a simple example, consider a system that has three registers, $R1$, $R2$, and $R3$. Each register is connected to the bus as indicated in Figure 7.61. We will design a control circuit that performs a single function—it swaps the contents of registers $R1$ and $R2$, using $R3$ for temporary storage.

The required swapping is done in three steps, each needing one clock cycle. In the first step the contents of $R2$ are transferred into $R3$. Then the contents of $R1$ are transferred into $R2$. Finally, the contents of $R3$, which are the original contents of $R2$, are transferred into $R1$. Note that we say that the contents of one register, R_i, are "transferred" into another register, R_j. This jargon is commonly used to indicate that the new contents of R_j will be a copy of the contents of R_i. The contents of R_i are not changed as a result of the transfer. Therefore, it would be more precise to say that the contents of R_i are "copied" into R_j.

Using a Shift Register for Control

There are many ways to design a suitable control circuit for the swap operation. One possibility is to use the left-to-right shift register shown in Figure 7.62. Assume that the reset input is used to clear the flip-flops to 0. Hence the control signals $R1_{in}$, $R1_{out}$, and so on are not asserted, because the shift register outputs have the value 0. The serial input w normally has the value 0. We assume that changes in the value of w are synchronized to occur shortly after the active clock edge. This assumption is reasonable because w would normally be generated as the output of some circuit that is controlled by the same clock signal. When the desired swap should be performed, w is set to 1 for one clock cycle, and then w returns to 0. After the next active clock edge, the output of the left-most flip-flop

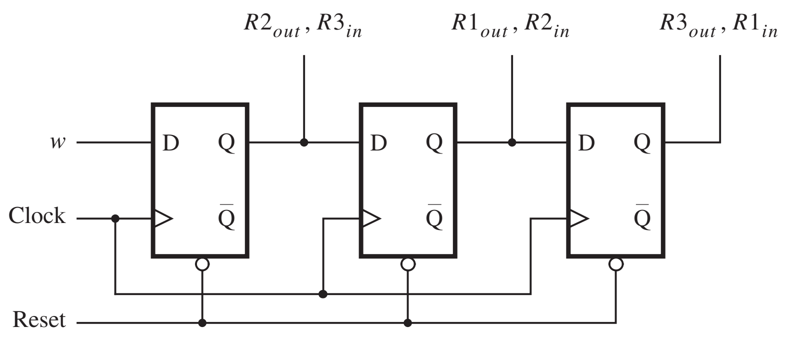


Figure 7.62 A shift-register control circuit.

becomes equal to 1, which asserts both $R2_{out}$ and $R3_{in}$. The contents of register $R2$ are placed onto the bus wires and are loaded into register $R3$ on the next active clock edge. This clock edge also shifts the contents of the shift register, resulting in $R1_{out} = R2_{in} = 1$. Note that since w is now 0, the first flip-flop is cleared, causing $R2_{out} = R3_{in} = 0$. The contents of $R1$ are now on the bus and are loaded into $R2$ on the next clock edge. After this clock edge the shift register contains 001 and thus asserts $R3_{out}$ and $R1_{in}$. The contents of $R3$ are now on the bus and are loaded into $R1$ on the next clock edge.

Using the control circuit in Figure 7.62, when w changes to 1 the swap operation does not begin until after the next active clock edge. We can modify the control circuit so that it starts the swap operation in the same clock cycle in which w changes to 1. One possible approach is illustrated in Figure 7.63. The reset signal is used to set the shift-register contents to 100, by presetting the left-most flip-flop to 1 and clearing the other two flip-flops. As long as $w = 0$, the output control signals are not asserted. When w changes to 1, the signals $R2_{out}$ and $R3_{in}$ are immediately asserted and the contents of $R2$ are placed onto the bus. The next active clock edge loads this data into $R3$ and also shifts the shift register contents to 010. Since the signal $R1_{out}$ is now asserted, the contents of $R1$ appear on the bus. The next clock edge loads this data into $R2$ and changes the shift register contents to 001. The contents of $R3$ are now on the bus; this data is loaded into $R1$ at the next clock edge, which also changes the shift register contents to 100. We assume that w had the value 1 for only one clock cycle; hence the output control signals are not asserted at this point. It may not be obvious to the reader how to design a circuit such as the one in Figure 7.63, because we have presented the design in an ad hoc fashion. In section 8.3 we will show how this circuit can be designed using a more formal approach.

The circuit in Figure 7.63 assumes that a preset input is available on the left-most flip-flop. If the flip-flop has only a clear input, then we can use the equivalent circuit shown in Figure 7.64. In this circuit we use the $\overline{\text{Q}}$ output of the left-most flip-flop and also complement the input to this flip-flop by using a NOR gate instead of an OR gate.

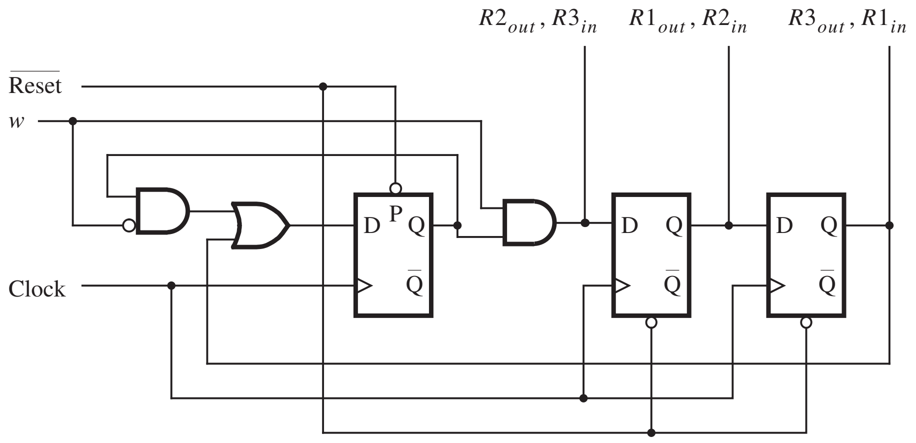


Figure 7.63 A modified control circuit.

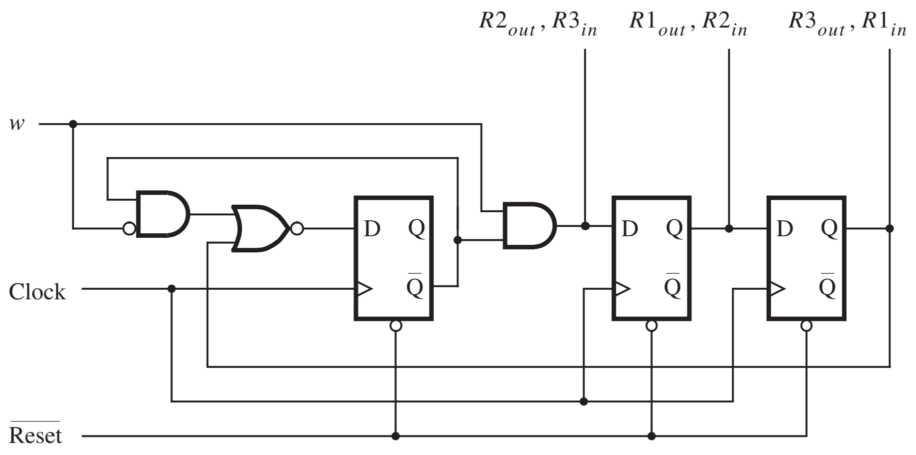


Figure 7.64 A modified version of the circuit in Figure 7.63.

Using Multiplexers to Implement a Bus

In Figure 7.60 we used tri-state buffers to control access to the bus. An alternative approach is to use multiplexers, as depicted in Figure 7.65. The outputs of each register are connected to a multiplexer. This multiplexer's output is connected to the inputs of the registers, thus realizing the bus. The multiplexer select inputs determine which register's contents appear on the bus. Although the figure shows just one multiplexer symbol, we actually need one multiplexer for each bit in the registers. For example, assume that there are 4 eight-bit registers, $R1$ to $R4$, plus the externally-supplied eight-bit *Data*. To

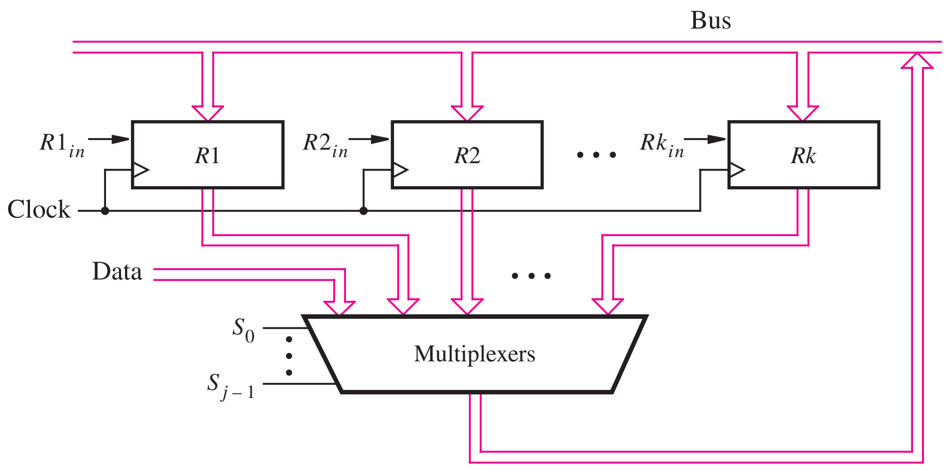


Figure 7.65 Using multiplexers to implement a bus.

interconnect them, we need eight 5-to-1 multiplexers. In Figure 7.62 we used a shift register to implement the control circuit. A similar approach can be used with multiplexers. The signals that control when data is loaded into a register, like $R1_{in}$, can still be connected directly to the shift-register outputs. However, instead of using control signals like $R1_{out}$ to place the contents of a register onto the bus, we have to generate the select inputs for the multiplexers. One way to do so is to connect the shift-register outputs to an encoder circuit that produces the select inputs for the multiplexer. We discussed encoder circuits in section 6.3.

The tri-state buffer and multiplexer approaches for implementing a bus are both equally valid. However, some types of chips, such as most PLDs, do not contain a sufficient number of tri-state buffers to realize even moderately large buses. In such chips the multiplexer-based approach is the only practical alternative. In practice, circuits are designed with CAD tools. If the designer describes the circuit using tri-state buffers, but there are not enough such buffers in the target device, then the CAD tools automatically produce an equivalent circuit that uses multiplexers.

Verilog Code

This section presents Verilog code for our circuit example that swaps the contents of two registers. We first give the code for the style of circuit in Figure 7.60 that uses tri-state buffers to implement the bus and then give the code for the style of circuit in Figure 7.65 that uses multiplexers. The code is written in a hierarchical fashion, using subcircuits for the registers, tri-state buffers, and the shift register. Figure 7.66 gives the code for an n-bit register of the type in Figure 7.61. The number of bits in the register is set by the parameter n, which has the default value of 8. The register is specified such that if the input $Rin = 1$, then the flip-flops are loaded from the n-bit input R. Otherwise, the flip-flops retain their presently stored values.

Figure 7.67 gives the code for a subcircuit that represents n tri-state buffers, each enabled by the input E. The inputs to the buffers are the n-bit signal Y, and the outputs are the n-bit signal F. The conditional assignment statement specifies that the output of

```
module regn (R, Rin, Clock, Q);
  parameter n = 8;
  input [n-1:0] R;
  input Rin, Clock;
  output [n-1:0] Q;
  reg [n-1:0] Q;

  always @(posedge Clock)
    if (Rin)
      Q <= R;

endmodule
```

Figure 7.66 Code for an n-bit register of the type in Figure 7.61.

```
module trin (Y, E, F);
    parameter n = 8;
    input [n−1:0] Y;
    input E;
    output [n−1:0] F;
    wire [n−1:0] F;

    assign F = E ? Y : 'bz;

endmodule
```

Figure 7.67 Code for an n-bit tri-state module.

each buffer is set to $F = Y$ if $E = 1$; otherwise, the output is set to the high impedance value z. The conditional assignment statement uses an unsized number to define the high impedance case. The Verilog compiler will make the size of this number the same as the size of vector Y, namely n. We cannot define the number as n'bz because the size of a sized number cannot be given as a parameter.

Figure 7.68 defines a shift register that can be used to implement the control circuit in Figure 7.62. The number of flip-flops is set by the generic parameter m, which has the default value of 4. The shift register has an active-low asynchronous reset input. The shift operation is defined with a **for** loop in the style used in Example 7.12.

```
module shiftr (Resetn, w, Clock, Q);
    parameter m = 4;
    input Resetn, w, Clock;
    output [1:m] Q;
    reg [1:m] Q;
    integer k;

    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
        begin
            for (k = m; k > 1 ; k = k−1)
                Q[k] <= Q[k−1];
            Q[1] <= w;
        end

endmodule
```

Figure 7.68 Code for the shift register in Figure 7.62.

The code in Figure 7.69 represents a digital system like the one in Figure 7.60, with 3 eight-bit registers, *R*1, *R*2, and *R*3. The circuit in Figure 7.60 includes tri-state buffers that are used to place *n* bits of externally supplied data on the bus. In Figure 7.69, these buffers are instantiated in the module *tri_ext*. Each of the eight buffers is enabled by the input signal *Extern*, and the data inputs on the buffers are attached to the eight-bit signal *Data*. When *Extern* = 1, the value of *Data* is placed on the bus, which is represented by the signal *BusWires*. The *BusWires* vector represents the circuit's output as well as the internal bus wiring. We declared this vector to be of **tri** type rather than of **wire** type. The keyword **tri** is treated in the same way as the keyword **wire** by the Verilog compiler. The designation **tri** makes it obvious to a reader that the synthesized connections will have tri-state capability.

We assume that a three-bit control signal named *RinExt* exists, which allows the externally supplied data to be loaded from the bus into register *R*1, *R*2, or *R*3. The *RinExt*

```
module swap (Data, Resetn, w, Clock, Extern, RinExt, BusWires);
  input [7:0] Data;
  input Resetn, w, Clock, Extern;
  input [1:3] RinExt;
  output [7:0] BusWires;
  tri [7:0] BusWires;
  wire [1:3] Rin, Rout, Q;
  wire [7:0] R1, R2, R3;

  shiftr control (Resetn, w, Clock, Q);
    defparam control.m = 3;

  assign Rin[1] = RinExt[1] | Q[3];
  assign Rin[2] = RinExt[2] | Q[2];
  assign Rin[3] = RinExt[3] | Q[1];
  assign Rout[1] = Q[2];
  assign Rout[2] = Q[1];
  assign Rout[3] = Q[3];

  regn reg_1 (BusWires, Rin[1], Clock, R1);
  regn reg_2 (BusWires, Rin[2], Clock, R2);
  regn reg_3 (BusWires, Rin[3], Clock, R3);

  trin tri_ext (Data, Extern, BusWires);
  trin tri_1 (R1, Rout[1], BusWires);
  trin tri_2 (R2, Rout[2], BusWires);
  trin tri_3 (R3, Rout[3], BusWires);

endmodule
```

Figure 7.69 A digital system like the one in Figure 7.60.

input is not shown in Figure 7.60, to keep the figure simple, but it would be generated by the same external circuit block that produces *Extern* and *Data*. When *RinExt*[1] = 1, the data on the bus is loaded into register *R*1; when *RinExt*[2] = 1, the data is loaded into *R*2; and when *RinExt*[3] = 1, the data is loaded into *R*3.

In Figure 7.69 the three-bit shift register is instantiated using the *shiftr* module under the instance name *control*. The outputs of the shift register are the three-bit signal Q. The parameter that defines the number of flip-flops in the *shiftr* module, *m*, has the default value of 4. Since we need to instantiate only a three-bit shift register, we have to change the value of parameter *m*. The parameter is set with the statement

```
defparam control.m = 3;
```

The **defparam** statement defines the values of the parameters indicated. The intended module instance is identified using the syntax *instance_name.parameter_name*. In our example, the instance name is *control* and the parameter name is *m*.

The next three statements in Figure 7.69 connect Q to the control signals that determine when data is loaded into each register, which are represented by the three-bit signal *Rin*. The signals *Rin*[1], *Rin*[2], and *Rin*[3] in the code correspond to the signals $R1_{in}$, $R2_{in}$, and $R3_{in}$ in Figure 7.60. As specified in Figure 7.62, the left-most shift-register output, Q[1], controls when data is loaded into register *R*3. Similarly, Q[2] controls register *R*2, and Q[3] controls *R*1. Each bit in *Rin* is ORed with the corresponding bit in *RinExt* so that externally supplied data can be stored in the registers as discussed above. The code also connects the shift-register outputs to the enable inputs, *Rout*, on the tri-state buffers. Figure 7.62 shows that Q[1] is used to put the contents of *R*2 onto the bus; hence *Rout*[2] is assigned the value of Q[1]. Similarly, *Rout*[1] is assigned the value of Q[2], and *Rout*[3] is assigned the value of Q[3]. The remaining statements in the code instantiate the registers and tri-state buffers in the system.

Verilog Code Using Multiplexers

Figure 7.70 shows how the code in Figure 7.69 can be modified to use multiplexers instead of tri-state buffers. Using the circuit structure shown in Figure 7.65, the bus is implemented with eight 4-to-1 multiplexers. Three of the data inputs on each 4-to-1 multiplexer are connected to one bit from registers *R*1, *R*2, and *R*3. The fourth data input is connected to one bit of the *Data* input signal to allow externally supplied data to be written into the registers. When the shift register's contents are 000, the multiplexers select *Data* to be placed on the bus. This data is loaded into the register selected by *RinExt*. It is loaded into *R*1 if *RinExt*[1] = 1, *R*2 if *RinExt*[2] = 1, and *R*3 if *RinExt*[3] = 1.

The *Rout* signal in Figure 7.69, which enables the tri-state buffers connected to the bus, is not needed for the multiplexer implementation. Instead, we have to provide the select inputs on the multiplexers. In Figure 7.70, the shift-register outputs are called Q. These signals generate the *Rin* control signals for the registers in the same way as shown in Figure 7.69. We said in the discussion concerning Figure 7.65 that an encoder is needed between the shift-register outputs and the multiplexer select inputs. A suitable encoder is described in the first **if-else** statement in Figure 7.70. It produces the multiplexer select inputs, which are named S. It sets $S = 00$ when the shift register contains 000, $S = 10$ when the shift register contains 100, and so on, as given in the code. The multiplexers are described by

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
   input [7:0] Data;
   input Resetn, w, Clock;
   input [1:3] RinExt;
   output [7:0] BusWires;
   reg [7:0] BusWires;
   wire [1:3] Rin, Q;
   wire [7:0] R1, R2, R3;
   reg [1:0] S;

   shiftr control (Resetn, w, Clock, Q);
      defparam control.m = 3;
   assign Rin[1] = RinExt[1] | Q[3];
   assign Rin[2] = RinExt[2] | Q[2];
   assign Rin[3] = RinExt[3] | Q[1];
   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);

   always @(Q or Data or R1 or R2 or R3 or S)
   begin
      // Encoder
      if (Q == 3'b000)  S = 2'b00;
      else if (Q == 3'b100)  S = 2'b10;
      else if (Q == 3'b010)  S = 2'b01;
      else  S = 2'b11;

      // Multiplexers
      if (S == 2'b00)  BusWires = Data;
      else if (S == 2'b01)  BusWires = R1;
      else if (S == 2'b10)  BusWires = R2;
      else  BusWires = R3;
   end

endmodule
```

Figure 7.70 Using multiplexers to implement a bus.

the second **if-else** statement, which places the value of *Data* onto the bus (*BusWires*) if $S = 00$, the contents of register $R1$ if $S = 01$, and so on. Using this scheme, when the swap operation is not active, the multiplexers place the bits from the *Data* input on the bus.

As described above, Figure 7.70 uses two **if-else** statements, one to describe an encoder and the other to describe the bus multiplexers. A simpler approach is to write a single **if-else** statement as shown in Figure 7.71. Here, each clause specifies directly which signal should

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
    input [7:0] Data;
    input Resetn, w, Clock;
    input [1:3] RinExt;
    output [7:0] BusWires;
    reg [7:0] BusWires;
    wire [1:3] Rin, Q;
    wire [7:0] R1, R2, R3;

    shiftr control (Resetn, w, Clock, Q);
        defparam control.m = 3;

    assign Rin[1] = RinExt[1] | Q[3];
    assign Rin[2] = RinExt[2] | Q[2];
    assign Rin[3] = RinExt[3] | Q[1];

    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);

    always @(Q or Data or R1 or R2 or R3)
    begin
        if (Q == 3'b000)   BusWires = Data;
        else if (Q == 3'b100)   BusWires = R2;
        else if (Q == 3'b010)   BusWires = R1;
        else   BusWires = R3;
    end

endmodule
```

Figure 7.71 A simplified version of the specification in Figure 7.70.

appear on *BusWires* for each pattern of the shift-register outputs. The circuit generated from the code in Figure 7.71 is equivalent to the one generated from the code in Figure 7.70.

Figure 7.72 gives an example of a timing simulation for a circuit synthesized from the code in Figure 7.71. In the first half of the simulation, the circuit is reset, and the contents of registers $R1$ and $R2$ are initialized. The hex value 55 is loaded into $R1$, and the value AA is loaded into $R2$. The clock edge at 275 ns, marked by the vertical reference line in Figure 7.72, loads the value $w = 1$ into the shift register. The contents of $R2$ (AA) then appear on the bus and are loaded into $R3$ by the clock edge at 325 ns. Following this clock edge, the contents of the shift register are 010, and the data stored in $R1$ (55) is on the bus. The clock edge at 375 ns loads this data into $R2$ and changes the shift register to 001. The contents of $R3$ (AA) now appear on the bus and are loaded into $R1$ by the clock edge at 425 ns. The shift register is now in state 000, and the swap is completed.

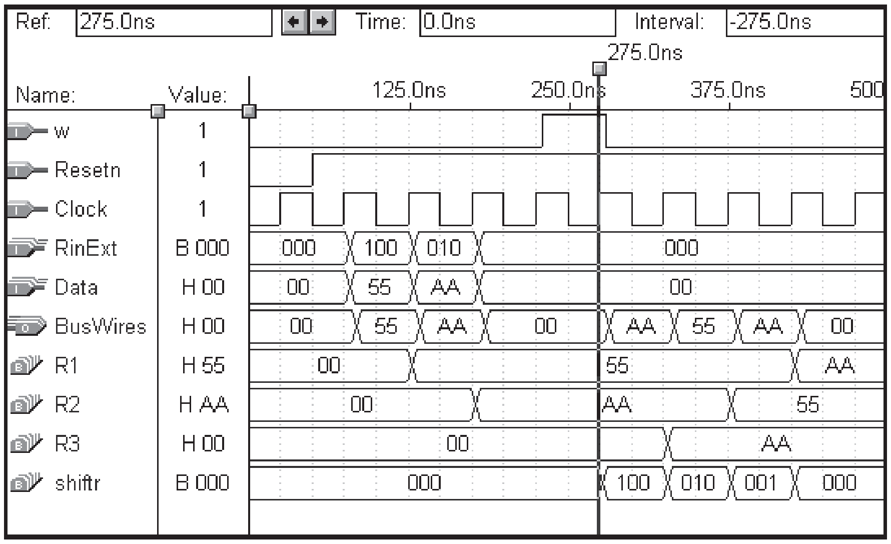


Figure 7.72 Timing simulation for the Verilog code in Figure 7.71.

7.14.2 Simple Processor

A second example of a digital system like the one in Figure 7.60 is shown in Figure 7.73. It has four n-bit registers, $R0, \ldots, R3$, that are connected to the bus with tri-state buffers. External data can be loaded into the registers from the n-bit *Data* input, which is connected to the bus using tri-state buffers enabled by the *Extern* control signal. The system also includes an adder/subtractor module. One of its data inputs is provided by an n-bit register, A, that is attached to the bus, while the other data input, B, is directly connected to the bus. If the *AddSub* signal has the value 0, the module generates the sum $A + B$; if $AddSub = 1$, the module generates the difference $A - B$. To perform the subtraction, we assume that the adder/subtractor includes the required XOR gates to form the 2's complement of B, as discussed in section 5.3. The register G stores the output produced by the adder/subtractor. The A and G registers are controlled by the signals A_{in}, G_{in}, and G_{out}.

The system in Figure 7.73 can perform various functions, depending on the design of the control circuit. As an example, we will design a control circuit that can perform the four operations listed in Table 7.2. The left column in the table shows the name of an operation and its operands; the right column indicates the function performed in the operation. For the *Load* operation the meaning of $Rx \leftarrow Data$ is that the data on the external *Data* input is transferred across the bus into any register, Rx, where Rx can be $R0$ to $R3$. The *Move* operation copies the data stored in register Ry into register Rx. In the table the square brackets, as in $[Rx]$, refer to the *contents* of a register. Since only a single transfer across the bus is needed, both the *Load* and *Move* operations require only one step (clock cycle) to be completed. The *Add* and *Sub* operations require three steps, as follows: In the first step

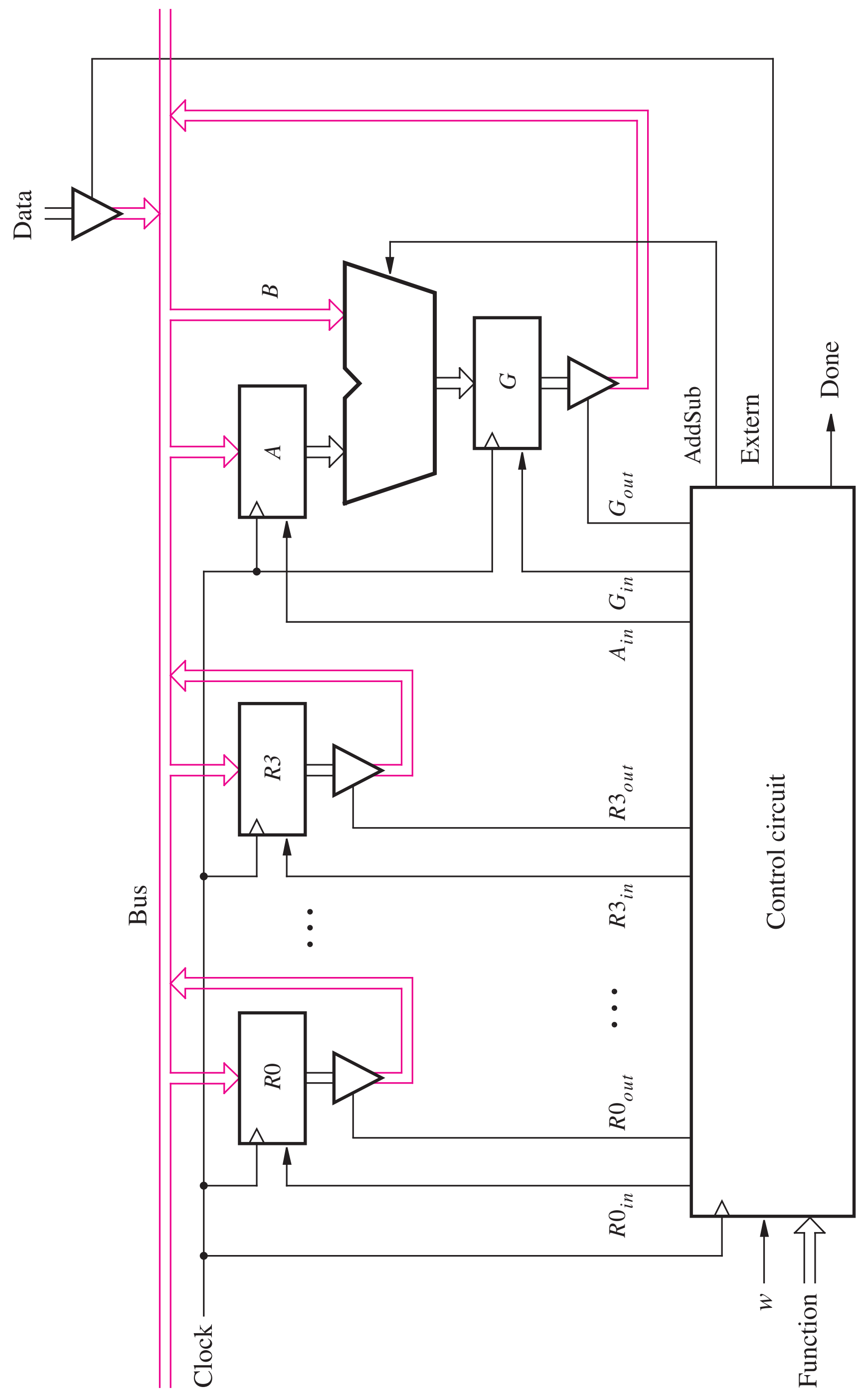


Figure 7.73 A digital system that implements a simple processor.

Table 7.2 Operations performed in the processor.

Operation	Function performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

the contents of *Rx* are transferred across the bus into register *A*. Then in the next step, the contents of *Ry* are placed onto the bus. The adder/subtractor module performs the required function, and the results are stored in register *G*. Finally, in the third step the contents of *G* are transferred into *Rx*.

A digital system that performs the types of operations listed in Table 7.2 is usually called a *processor*. The specific operation to be performed at any given time is indicated using the control circuit input named *Function*. The operation is initiated by setting the *w* input to 1, and the control circuit asserts the *Done* output when the operation is completed.

In Figure 7.60 we used a shift register to implement the control circuit. It is possible to use a similar design for the system in Figure 7.73. To illustrate a different approach, we will base the design of the control circuit on a counter. This circuit has to generate the required control signals in each step of each operation. Since the longest operations (*Add* and *Sub*) need three steps (clock cycles), a two-bit counter can be used. Figure 7.74 shows a two-bit up-counter connected to a 2-to-4 decoder. Decoders are discussed in section 6.2. The decoder is enabled at all times by setting its enable (*En*) input permanently to the value 1. Each of the decoder outputs represents a step in an operation. When no operation is currently being performed, the count value is 00; hence the T_0 output of the decoder is

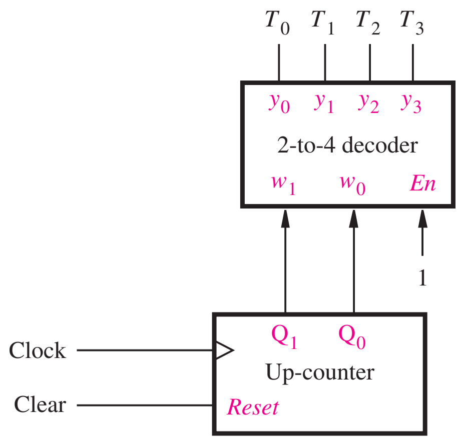


Figure 7.74 A part of the control circuit for the processor.

asserted. In the first step of an operation, the count value is 01, and T_1 is asserted. During the second and third steps of the *Add* and *Sub* operations, T_2 and T_3 are asserted, respectively.

In each of steps T_0 to T_3, various control signal values have to be generated by the control circuit, depending on the operation being performed. Figure 7.75 shows that the operation is specified with six bits, which form the *Function* input. The two left-most bits, $F = f_1 f_0$, are used as a two-bit number that identifies the operation. To represent *Load*, *Move*, *Add*, and *Sub*, we use the codes $f_1 f_0 = 00$, 01, 10, and 11, respectively. The inputs $Rx_1 Rx_0$ are a binary number that identifies the *Rx* operand, while $Ry_1 Ry_0$ identifies the *Ry* operand. The *Function* inputs are stored in a six-bit Function Register when the FR_{in} signal is asserted.

Figure 7.75 also shows three 2-to-4 decoders that are used to decode the information encoded in the *F*, *Rx*, and *Ry* inputs. We will see shortly that these decoders are included as a convenience because their outputs provide simple-looking logic expressions for the various control signals.

The circuits in Figures 7.74 and 7.75 form a part of the control circuit. Using the input *w* and the signals $T_0, \ldots, T_3, I_0, \ldots, I_3, X_0, \ldots, X_3$, and $Y_0, \ldots, Y_3$, we will show how to derive the rest of the control circuit. It has to generate the outputs *Extern*, *Done*, A_{in}, G_{in}, G_{out}, *AddSub*, $R0_{in}, \ldots, R3_{in}$, and $R0_{out}, \ldots, R3_{out}$. The control circuit also has to generate the *Clear* and FR_{in} signals used in Figures 7.74 and 7.75.

Clear and FR_{in} are defined in the same way for all operations. *Clear* is used to ensure that the count value remains at 00 as long as $w = 0$ and no operation is being executed. Also, it is used to clear the count value to 00 at the end of each operation. Hence an appropriate

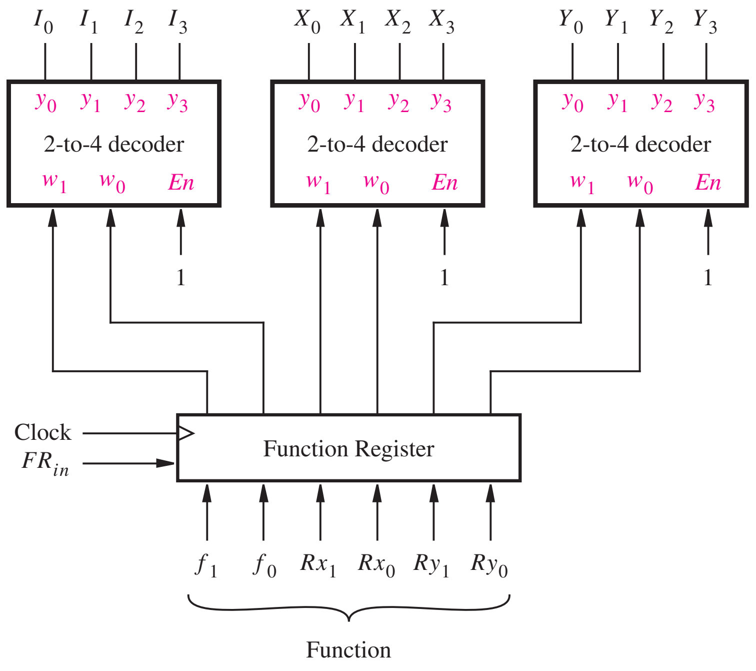


Figure 7.75 The function register and decoders.

logic expression is

$$Clear = \overline{w}\,T_0 + Done$$

The FR_{in} signal is used to load the values on the *Function* inputs into the Function Register when w changes to 1. Hence

$$FR_{in} = wT_0$$

The rest of the outputs from the control circuit depend on the specific step being performed in each operation. The values that have to be generated for each signal are shown in Table 7.3. Each row in the table corresponds to a specific operation, and each column represents one time step. The *Extern* signal is asserted only in the first step of the *Load* operation. Therefore, the logic expression that implements this signal is

$$Extern = I_0T_1$$

Done is asserted in the first step of *Load* and *Move*, as well as in the third step of *Add* and *Sub*. Hence

$$Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

The A_{in}, G_{in}, and G_{out} signals are asserted in the *Add* and *Sub* operations. A_{in} is asserted in step T_1, G_{in} is asserted in T_2, and G_{out} is asserted in T_3. The *AddSub* signal has to be set to 0 in the *Add* operation and to 1 in the *Sub* operation. This is achieved with the following logic expressions

$$\begin{aligned} A_{in} &= (I_2 + I_3)T_1 \\ G_{in} &= (I_2 + I_3)T_2 \\ G_{out} &= (I_2 + I_3)T_3 \\ AddSub &= I_3 \end{aligned}$$

The values of $R0_{in}, \ldots, R3_{in}$ are determined using either the $X_0, \ldots, X_3$ signals or the $Y_0, \ldots, Y_3$ signals. In Table 7.3 these actions are indicated by writing either $R_{in} = X$ or $R_{in} = Y$. The meaning of $R_{in} = X$ is that $R0_{in} = X_0$, $R1_{in} = X_1$, and so on. Similarly, the values of $R0_{out}, \ldots, R3_{out}$ are specified using either $R_{out} = X$ or $R_{out} = Y$.

Table 7.3 Control signals asserted in each operation/time step.

	T_1	T_2	T_3
(Load): I_0	$Extern$, $R_{in} = X$, $Done$		
(Move): I_1	$R_{in} = X$, $R_{out} = Y$, $Done$		
(Add): I_2	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, $AddSub = 0$	G_{out}, $R_{in} = X$, $Done$
(Sub): I_3	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, $AddSub = 1$	G_{out}, $R_{in} = X$, $Done$

We will develop the expressions for $R0_{in}$ and $R0_{out}$ by examining Table 7.3 and then show how to derive the expressions for the other register control signals. The table shows that $R0_{in}$ is set to the value of X_0 in the first step of both the *Load* and *Move* operations and in the third step of both the *Add* and *Sub* operations, which leads to the expression

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

Similarly, $R0_{out}$ is set to the value of Y_0 in the first step of *Move*. It is set to X_0 in the first step of *Add* and *Sub* and to Y_0 in the second step of these operations, which gives

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

The expressions for $R1_{in}$ and $R1_{out}$ are the same as those for $R0_{in}$ and $R0_{out}$ except that X_1 and Y_1 are used in place of X_0 and Y_0. The expressions for $R2_{in}$, $R2_{out}$, $R3_{in}$, and $R3_{out}$ are derived in the same way.

The circuits shown in Figures 7.74 and 7.75, combined with the circuits represented by the above expressions, implement the control circuit in Figure 7.73.

Processors are extremely useful circuits that are widely used. We have presented only the most basic aspects of processor design. However, the techniques presented can be extended to design realistic processors, such as modern microprocessors. The interested reader can refer to books on computer organization for more details on processor design [1–2].

Verilog Code

In this section we give two different styles of Verilog code for describing the system in Figure 7.73. The first style uses tri-state buffers to represent the bus, and it gives the logic expressions shown above for the outputs of the control circuit. The second style of code uses multiplexers to represent the bus, and it uses **case** statements that correspond to Table 7.3 to describe the outputs of the control circuit.

Verilog code for an up-counter is shown in Figure 7.56. A modified version of this counter, named *upcount*, is shown in the code in Figure 7.76. It has a synchronous reset input, which is active high. Other subcircuits that we use in the Verilog code for the processor are the *dec2to4*, *regn*, and *trin* modules in Figures 6.35, 7.66, and 7.67.

```
module upcount (Clear, Clock, Q);
    input Clear, Clock;
    output [1:0] Q;
    reg [1:0] Q;

    always @(posedge Clock)
        if (Clear)
            Q <= 0;
        else
            Q <= Q + 1;

endmodule
```

Figure 7.76 A two-bit up-counter with synchronous reset.

Complete code for the processor is given in Figure 7.77. The instantiated modules *counter* and *decT* represent the subcircuits in Figure 7.74. Note that we have assumed that the circuit has an active-high reset input, *Reset*, which is used to initialize the counter to 00. The statement **assign** Func = {F, Rx, Ry} uses the concatenate operator to create the six-bit signal *Func*, which represents the inputs to the Function Register in Figure 7.75. The *functionreg* module represents the Function Register with the data inputs *Func* and the

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
   input [7:0] Data;
   input Reset, w, Clock;
   input [1:0] F, Rx, Ry;
   output [7:0] BusWires;
   output Done;
   wire [7:0] BusWires;
   reg [0:3] Rin, Rout;
   reg [7:0] Sum;
   wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
   wire [1:0] Count;
   wire [0:3] T, I, Xreg, Y;
   wire [7:0] R0, R1, R2, R3, A, G;
   wire [1:6] Func, FuncReg;
   integer k;

   upcount counter (Clear, Clock, Count);
   dec2to4 decT (Count, 1, T);

   assign Clear = Reset | Done | (~w & T[0]);
   assign Func = {F, Rx, Ry};
   assign FRin = w & T[0];

   regn functionreg (Func, FRin, Clock, FuncReg);
      defparam functionreg.n = 6;
   dec2to4 decI (FuncReg[1:2], 1, I);
   dec2to4 decX (FuncReg[3:4], 1, Xreg);
   dec2to4 decY (FuncReg[5:6], 1, Y);

   assign Extern = I[0] & T[1];
   assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
   assign Ain = (I[2] | I[3]) & T[1];
   assign Gin = (I[2] | I[3]) & T[2];
   assign Gout = (I[2] | I[3]) & T[3];
   assign AddSub = I[3];

. . . continued in Part b.
```

Figure 7.77 Code for the prcoessor (Part *a*).

```
    // RegCntl
    always @(I or T or Xreg or Y)
        for (k = 0; k < 4; k = k+1)
        begin
            Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
                ((I[2] | I[3]) & T[1] & Y[k]);
            Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
                ((T[1] & Xreg[k]) | (T[2] & Y[k])));
        end

    trin tri_ext (Data, Extern, BusWires);
    regn  reg_0 (BusWires, Rin[0], Clock, R0);
    regn  reg_1 (BusWires, Rin[1], Clock, R1);
    regn  reg_2 (BusWires, Rin[2], Clock, R2);
    regn  reg_3 (BusWires, Rin[3], Clock, R3);

    trin  tri_0 (R0, Rout[0], BusWires);
    trin  tri_1 (R1, Rout[1], BusWires);
    trin  tri_2 (R2, Rout[2], BusWires);
    trin  tri_3 (R3, Rout[3], BusWires);
    regn  reg_A (BusWires, Ain, Clock, A);

    // alu
    always @(AddSub or A or BusWires)
        if (!AddSub)
            Sum = A + BusWires;
        else
            Sum = A − BusWires;

    regn  reg_G (Sum, Gin, Clock, G);
    trin  tri_G (G, Gout, BusWires);

endmodule
```

Figure 7.77 Code for the processor (Part *b*).

outputs *FuncReg*. The instantiated modules *decI*, *decX*, and *decY* represent the decoders in Figure 7.75. Following these statements the previously derived logic expressions for the outputs of the control circuit are given. For $R0_{in}, \ldots, R3_{in}$ and $R0_{out}, \ldots, R3_{out}$, a **for** loop is used to produce the expressions.

At the end of the code, the adder/subtractor module is defined and the tri-state buffers and registers in the processor are instantiated.

Using Multiplexers and Case Statements

We showed in Figure 7.65 that a bus can be implemented with multiplexers, rather than tri-state buffers. Verilog code that describes the processor using this approach is shown in Figure 7.78. The code illustrates a different way of describing the control circuit in the

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output [7:0] BusWires;
    output Done;
    reg [7:0] BusWires, Sum;
    reg [0:3] Rin, Rout;
    reg Extern, Done, Ain, Gin, Gout, AddSub;
    wire [1:0] Count, I;
    wire [0:3] Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg, Sel;

    wire Clear = Reset | Done | (~w & ~Count[1] & ~Count[0]);
    upcount counter (Clear, Clock, Count);
    assign Func = {F, Rx, Ry};
    wire FRin = w & ~Count[1] & ~Count[0];
    regn functionreg (Func, FRin, Clock, FuncReg);
        defparam functionreg.n = 6;
    assign I = FuncReg[1:2];
    dec2to4 decX (FuncReg[3:4], 1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1, Y);

    always @(Count or I or Xreg or Y)
    begin
        Extern = 1'b0; Done = 1'b0; Ain = 1'b0; Gin = 1'b0;
        Gout = 1'b0; AddSub = 1'b0; Rin = 4'b0; Rout = 4'b0;
        case (Count)
            2'b00: ; //no signals asserted in time step T0
            2'b01:   //define signals in time step T1
                case (I)
                    2'b00: begin //Load
                              Extern = 1'b1; Rin = Xreg; Done = 1'b1;
                           end
                    2'b01: begin //Move
                              Rout = Y; Rin = Xreg; Done = 1'b1;
                           end
                    default: begin //Add, Sub
                                Rout = Xreg; Ain = 1'b1;
                             end
                endcase
. . . continued in Part b.
```

Figure 7.78 Alternative code for the processor (Part *a*).

```
            2'b10: //define signals in time step T2
                case(I)
                    2'b10: begin //Add
                                Rout = Y; Gin = 1'b1;
                           end
                    2'b11: begin //Sub
                                Rout = Y; AddSub = 1'b1; Gin = 1'b1;
                           end
                    default: ; //Add, Sub
                endcase
            2'b11:
                case (I)
                    2'b10, 2'b11: begin
                                        Gout = 1'b1; Rin = Xreg; Done = 1'b1;
                                  end
                    default: ; //Add, Sub
                endcase
        endcase
    end

    regn reg_0 (BusWires, Rin[0], Clock, R0);
    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);
    regn reg_A (BusWires, Ain, Clock, A);

. . . continued in Part c.
```

Figure 7.78 Alternative code for the processor (Part *b*).

processor. It does not give logic expressions for the signals *Extern*, *Done*, and so on, as in Figure 7.77. Instead, **case** statements are used to represent the information shown in Table 7.3. Each control signal is first assigned the value 0 as a default. This is required because the **case** statements specify the values of the control signals only when they should be asserted, as we did in Table 7.3. As explained in section 7.12.2, when the value of a signal is not specified, the signal retains its current value. This implied memory results in a feedback connection in the synthesized circuit. We avoid this problem by providing the default value of 0 for each of the control signals involved in the **case** statements.

In Figure 7.77 the decoders *decT* and *decI* are used to decode the *Count* signal and the stored values of the *F* input, respectively. The *decT* decoder has the outputs $T_0, \ldots, T_3$, and *decI* produces $I_0, \ldots, I_3$. In Figure 7.78 these two decoders are not used, because they do not serve a useful purpose in this code. Instead, the signal *I* is defined as a two-bit signal, and the two-bit signal *Count* is used instead of T. These signals are used in the **case** statements. The code sets *I* to the value of the two left-most bits in the Function Register, which correspond to the stored values of the input F.

```
    // alu
    always @(AddSub or A or BusWires)
    begin
        if (!AddSub)
            Sum = A + BusWires;
        else
            Sum = A − BusWires;
    end

    regn reg_G (Sum, Gin, Clock, G);
    assign Sel = {Rout, Gout, Extern};

    always @(Sel or R0 or R1 or R2 or R3 or G or Data)
    begin
        if (Sel == 6'b100000)
            BusWires = R0;
        else if (Sel == 6'b010000)
            BusWires = R1;
        else if (Sel == 6'b001000)
            BusWires = R2;
        else if (Sel == 6'b000100)
            BusWires = R3;
        else if (Sel == 6'b000010)
            BusWires = G;
        else BusWires = Data;
    end

endmodule
```

Figure 7.78 Alternative code for the processor (Part *c*).

There are two nested levels of **case** statements. The first one enumerates the possible values of *Count*. For each alternative in this **case** statement, which represents a column in Table 7.3, there is a nested **case** statement that enumerates the four values of *I*. As indicated by the comments in the code, the nested **case** statements correspond exactly to the information given in Table 7.3.

At the end of Figure 7.78, the bus is described with an **if-else** statement which represents multiplexers that place the appropriate data onto *BusWires*, depending on the values of R_{out}, G_{out}, and *Extern*.

The circuits synthesized from the code in Figures 7.77 and 7.78 are functionally equivalent. The style of code in Figure 7.78 has the advantage that it does not require the manual effort of analyzing Table 7.3 to generate the logic expressions for the control signals in Figure 7.77. By using the style of code in Figure 7.78, these expressions are produced automatically by the Verilog compiler as a result of analyzing the **case** statements. The style of code in Figure 7.78 is less prone to careless errors. Also, using this style of code it would be straightforward to provide additional capabilities in the processor, such as adding other operations.

We synthesized a circuit to implement the code in Figure 7.78 in a chip. Figure 7.79 gives an example of the results of a timing simulation. Each clock cycle in which $w = 1$ in this timing diagram indicates the start of an operation. In the first such operation, at 250 ns in the simulation time, the values of both inputs F and Rx are 00. Hence the operation corresponds to "*Load* $R0$, *Data*." The value of *Data* is 2A, which is loaded into $R0$ on the next positive clock edge. The next operation loads 55 into register $R1$, and the subsequent operation loads 22 into $R2$. At 850 ns the value of the input F is 10, while $Rx = 01$ and $Ry = 00$. This operation is "*Add* $R1$, $R0$." In the following clock cycle, the contents of $R1$ (55) appear on the bus. This data is loaded into register A by the clock edge at 950 ns, which also results in the contents of $R0$ (2A) being placed on the bus. The adder/subtractor module generates the correct sum (7F), which is loaded into register G at 1050 ns. After this clock edge the new contents of G (7F) are placed on the bus and loaded into register $R1$ at 1150 ns. Two more operations are shown in the timing diagram. The one at 1250 ns ("*Move* $R3$, $R1$") copies the contents of $R1$ (7F) into $R3$. Finally, the operation starting at 1450 ns ("*Sub* $R3$, $R2$") subtracts the contents of $R2$ (22) from the contents of $R3$ (7F), producing the correct result, $7F - 22 = 5D$.

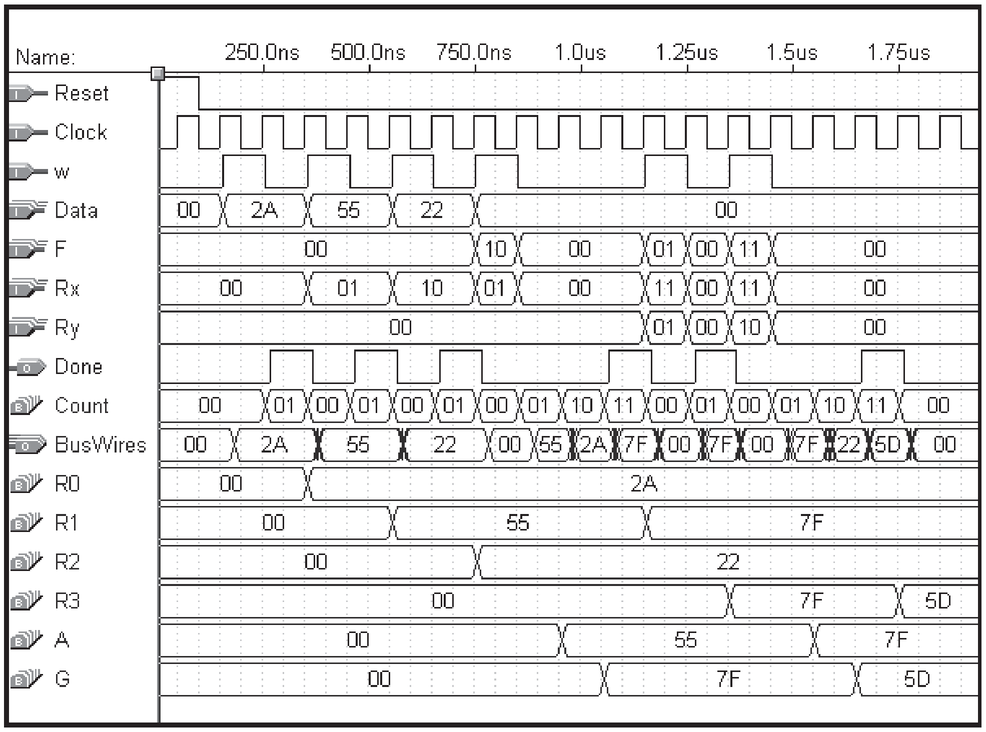


Figure 7.79 Timing simulation for the Verilog code in Figure 7.78.

7.14.3 Reaction Timer

We showed in Chapter 3 that electronic devices operate at remarkably fast speeds, with the typical delay through a logic gate being less than 1 ns. In this example we use a logic circuit to measure the speed of a much slower type of device—a person.

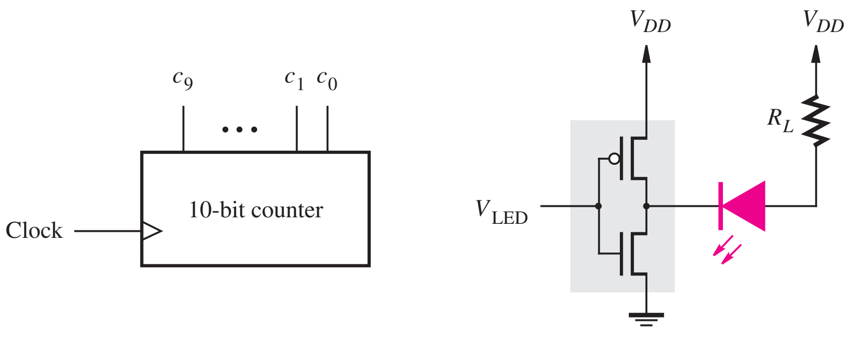
We will design a circuit that can be used to measure the reaction time of a person to a specific event. The circuit turns on a small light, called a *light-emitting diode (LED)*. In response to the LED being turned on, the person attempts to press a switch as quickly as possible. The circuit measures the elapsed time from when the LED is turned on until the switch is pressed.

To measure the reaction time, a clock signal with an appropriate frequency is needed. In this example we use a 100 Hz clock, which measures time at a resolution of 1/100 of a second. The reaction time can then be displayed using two digits that represent fractions of a second from 00/100 to 99/100.

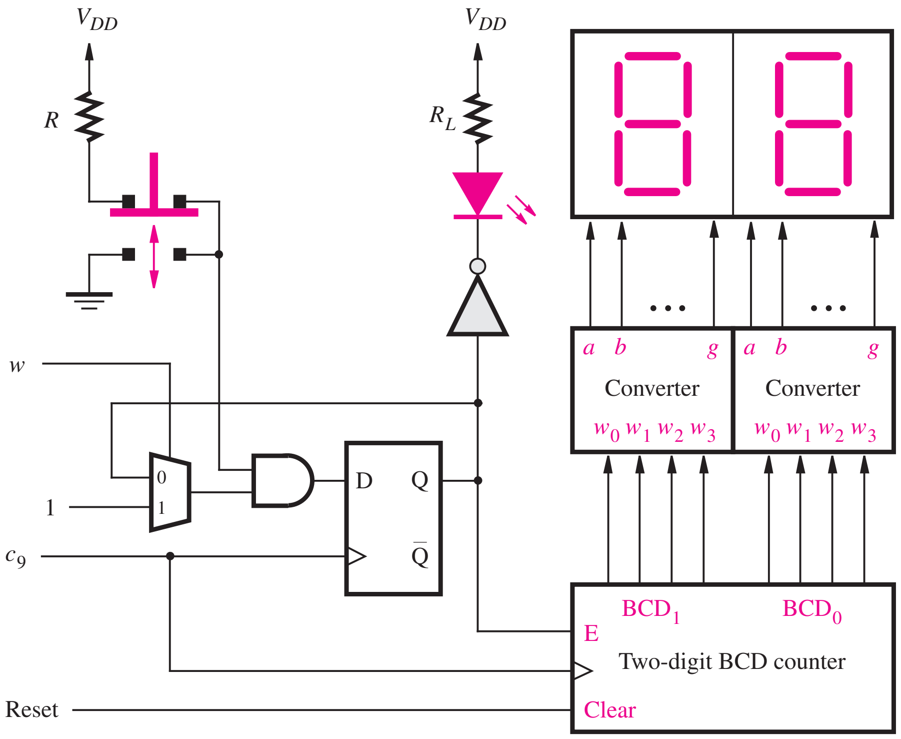
Digital systems often include high-frequency clock signals to control various subsystems. In this case assume the existence of an input clock signal with the frequency 102.4 kHz. From this signal we can derive the required 100 Hz signal by using a counter as a *clock divider*. A timing diagram for a four-bit counter is given in Figure 7.22. It shows that the least-significant bit output, Q_0, of the counter is a periodic signal with half the frequency of the clock input. Hence we can view Q_0 as dividing the clock frequency by two. Similarly, the Q_1 output divides the clock frequency by four. In general, output Q_i in an n-bit counter divides the clock frequency by 2^{i+1}. In the case of our 102.4 kHz clock signal, we can use a 10-bit counter, as shown in Figure 7.80*a*. The counter output c_9 has the required 100 Hz frequency because $102400 \text{ Hz}/1024 = 100$ Hz.

The reaction timer circuit has to be able to turn an LED on and off. The graphical symbol for an LED is shown in blue in Figure 7.80*b*. Small blue arrows in the symbol represent the light that is emitted when the LED is turned on. The LED has two terminals: the one on the left in the figure is the *cathode*, and the terminal on the right is the *anode*. To turn the LED on, the cathode has to be set to a lower voltage than the anode, which causes a current to flow through the LED. If the voltages on its two terminals are equal, the LED is off.

Figure 7.80*b* shows one way to control the LED, using an inverter. If the input voltage $V_{LED} = 0$, then the voltage at the cathode is equal to V_{DD}; hence the LED is off. But if $V_{LED} = V_{DD}$, the cathode voltage is 0 V and the LED is on. The amount of current that flows is limited by the value of the resistor R_L. This current flows through the LED and the NMOS transistor in the inverter. Since the current flows *into* the inverter, we say that the inverter *sinks* the current. The maximum current that a logic gate can sink without sustaining permanent damage is usually called I_{OL}, which stands for the "maximum current when the output is low." The value of R_L is chosen such that the current is less than I_{OL}. As an example assume that the inverter is implemented inside a PLD device. The typical value of I_{OL}, which would be specified in the data sheet for the PLD, is about 12 mA. For $V_{DD} = 5$ V, this leads to $R_L \approx 450\ \Omega$ because $5\ V/450\ \Omega = 11$ mA (there is actually a small voltage drop across the LED when it is turned on, but we ignore this for simplicity). The amount of light emitted by the LED is proportional to the current flow. If 11 mA is insufficient, then the inverter should be implemented in



(a) Clock divider (b) LED circuit



(c) Push-button switch, LED, and 7-segment displays

Figure 7.80 A reaction-timer circuit.

a buffer chip, like those described in section 3.5, because buffers provide a higher value of I_{OL}.

The complete reaction-timer circuit is illustrated in Figure 7.80*c*, with the inverter from part (*b*) shaded in grey. The graphical symbol for a push-button switch is shown in the top left of the diagram. The switch normally makes contact with the top terminals, as depicted in the figure. When depressed, the switch makes contact with the bottom terminals; when released, it automatically springs back to the top position. In the figure the switch is connected such that it normally produces a logic value of 1, and it produces a 0 pulse when pressed.

The push-button switch is connected to the clear input on a D flip-flop. The output of this flip-flop determines whether the LED is on or off, and it also provides the count enable input to a two-digit BCD counter. As discussed in section 7.11, each digit in a BCD counter has four bits that take the values 0000 to 1001. Thus the counting sequence can be viewed as decimal numbers from 00 to 99. A circuit for the BCD counter is given in Figure 7.28. In Figure 7.80*c* both the flip-flop and the counter are clocked by the c_9 output of the clock divider in part (*a*) of the figure. The intended use of the reaction-timer circuit is to first depress the switch to turn off the LED and disable the counter. Then the *Reset* input is asserted to clear the contents of the counter to 00. The input w normally has the value 0, which keeps the flip-flop cleared and prevents the count value from changing. The reaction test is initiated by setting $w = 1$ for one c_9 clock cycle. After the next positive edge of c_9, the flip-flop output becomes a 1, which turns on the LED. We assume that w returns to 0 after one clock cycle, but the flip-flop output remains at 1 because of the 2-to-1 multiplexer connected to the D input. The counter is then incremented every 1/100 of a second. Each digit in the counter is connected through a code converter to a 7-segment display, which we described in the discussion for Figure 6.25. When the user depresses the switch, the flip-flop is cleared, which turns off the LED and stops the counter. The two-digit display shows the elapsed time to the nearest 1/100 of a second from when the LED was turned on until the user was able to respond by depressing the switch.

Verilog Code

To describe the circuit in Figure 7.80*c* using Verilog code, we can make use of subcircuits for the BCD counter and the 7-segment code converter. The code for the latter subcircuit is given in Figure 6.38 and is not repeated here. Code for the BCD counter, which represents the circuit in Figure 7.28, is shown in Figure 7.81. The two-digit BCD output is represented by the 2 four-bit signals *BCD*1 and *BCD*0. The *Clear* input provides a synchronous reset for both digits in the counter. If $E = 1$, the count value is incremented on the positive clock edge; and if $E = 0$, the count value is unchanged. Each digit can take the values from 0000 to 1001.

Figure 7.82 gives the code for the reaction timer. The input signal *Pushn* represents the value produced by the push-button switch. The output signal *LEDn* represents the output of the inverter that is used to control the LED. The two 7-segment displays are controlled by the seven-bit signals *Digit*1 and *Digit*0.

In Figure 7.61 we showed how a register, R, can be designed with a control signal R_{in}. If $R_{in} = 1$ data is loaded into the register on the active clock edge and if $R_{in} = 0$, the stored contents of the register are not changed. The flip-flop in Figure 7.80 is used in the same

```
module BCDcount (Clock, Clear, E, BCD1, BCD0);
   input Clock, Clear, E;
   output [3:0] BCD1, BCD0;
   reg [3:0] BCD1, BCD0;

   always @(posedge Clock)
   begin
      if (Clear)
      begin
         BCD1 <= 0;
         BCD0 <= 0;
      end
      else if (E)
         if (BCD0 == 4'b1001)
         begin
            BCD0 <= 0;
            if (BCD1 == 4'b1001)
               BCD1 <= 0;
            else
               BCD1 <= BCD1 + 1;
         end
         else
            BCD0 <= BCD0 + 1;
   end

endmodule
```

Figure 7.81 Code for the two-digit BCD counter in Figure 7.28.

way. If $w = 1$, the flip-flop is loaded with the value 1, but if $w = 0$ the stored value in the flip-flop is not changed. This circuit is described by the **always** block in Figure 7.82, which also includes a synchronous reset input. We have chosen to use a synchronous reset because the flip-flop output is connected to the enable input E on the BCD counter. As we know from the discussion in section 7.3, it is important that all signals connected to flip-flops meet the required setup and hold times. The push-button switch can be pressed at any time and is not synchronized to the c_9 clock signal. By using a synchronous reset for the flip-flop in Figure 7.80, we avoid possible timing problems in the counter.

The flip-flop output is named *LED*, which is inverted to produce the *LEDn* signal that controls the LED. In the device used to implement the circuit, *LEDn* would be generated by a buffer that is connected to an output pin on the chip package. If a PLD is used, this buffer has the associated value of $I_{OL} = 12$ mA that we mentioned earlier. At the end of Figure 7.82, the BCD counter and 7-segment code converters are instantiated as subcircuits.

A simulation of the reaction-timer circuit implemented in a chip is shown in Figure 7.83. Initially, *Pushn* is set to 0 to simulate depressing the switch to turn off the LED, and

```
module reaction (c9, Reset, w, Pushn, LEDn, Digit1, Digit0);
   input c9, Reset, w, Pushn;
   output LEDn;
   output [1:7] Digit1, Digit0;
   wire LEDn;
   wire [1:7] Digit1, Digit0;
   reg LED;
   wire [3:0] BCD1, BCD0;

   always @(posedge c9)
   begin
      if (Pushn == 0)
         LED <= 0;
      else if (w)
         LED <= 1;
   end

   assign LEDn = ~LED;
   BCDcount counter (c9, Reset, LED, BCD1, BCD0);
   seg7 seg1 (BCD1, Digit1);
   seg7 seg0 (BCD0, Digit0);

endmodule
```

Figure 7.82 Code for the reaction timer.

then *Pushn* returns to 1. Also, *Reset* is asserted to clear the counter. When *w* changes to 1, the circuit sets *LEDn* to 0, which represents the LED being turned on. After some amount of time, the switch will be depressed. In the simulation we arbitrarily set *Pushn* to 0 after 18 c_9 clock cycles. Thus this choice represents the case when the person's reaction time is about 0.18 seconds. In human terms this duration is a very short time; for electronic circuits it is a very long time. An inexpensive personal computer can perform tens of millions of operations in 0.18 seconds!

7.14.4 Register Transfer Level (RTL) Code

At this point, we have introduced most of the Verilog constructs that are needed for synthesis. Most of our examples give behavioral code, utilizing **if-else** statements, **case** statements, **for** loops, and other procedural statements. It is possible to write behavioral code in a style that resembles a computer program, in which there is a complex flow of control with many loops and branches. With such code, sometimes called *high-level* behavioral code, it is difficult to relate the code to the final hardware implementation; it may even be difficult to predict what circuit a high-level synthesis tool will produce. In this book we do not use the high-level

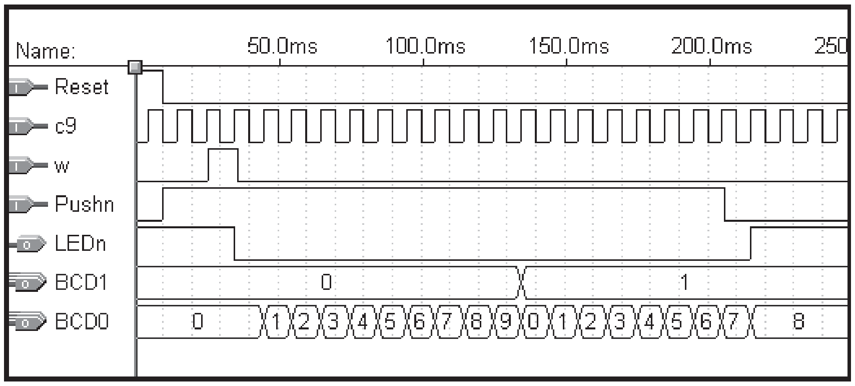


Figure 7.83 Simulation of the reaction timer circuit.

style of code. Instead, we present Verilog code in such a way that the code can be easily related to the circuit that is being described. Most design modules presented are fairly small, to facilitate simple descriptions. Larger designs are built by interconnecting the smaller modules. This approach is usually referred to as the *register-transfer level* (RTL) style of code. It is the most popular design method used in practice. RTL code is characterized by a straightforward flow of control through the code; it comprises well-understood subcircuits that are connected together in a simple way.

7.15 Concluding Remarks

In this chapter we have presented circuits that serve as basic storage elements in digital systems. These elements are used to build larger units such as registers, shift registers, and counters. Many other texts that deal with this material are available [3–11]. We have illustrated how circuits with flip-flops can be described using Verilog code. More information on Verilog can be found in [12–19]. In the next chapter a more formal method for designing circuits with flip-flops will be presented.

Problems

7.1 Consider the timing diagram in Figure P7.1. Assuming that the *D* and *Clock* inputs shown are applied to the circuit in Figure 7.12, draw waveforms for the Q_a, Q_b, and Q_c signals.

7.2 Can the circuit in Figure 7.3 be modified to implement an SR latch? Explain your answer.

7.3 Figure 7.5 shows a latch built with NOR gates. Draw a similar latch using NAND gates. Derive its truth table and show its timing diagram.

7.4 Show a circuit that implements the gated SR latch using NAND gates only.

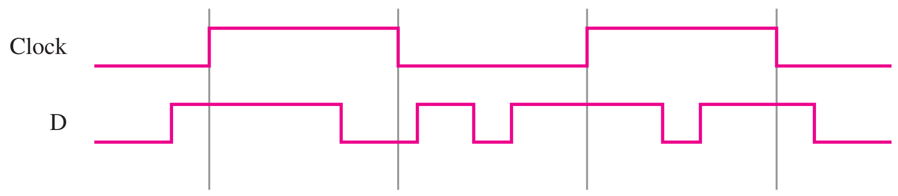


Figure P7.1 Timing diagram for problem 7.1.

7.5 Given a 100-MHz clock signal, derive a circuit using D flip-flops to generate 50-MHz and 25-MHz clock signals. Draw a timing diagram for all three clock signals, assuming reasonable delays.

7.6 An SR flip-flop is a flip-flop that has set and reset inputs like a gated SR latch. Show how an SR flip-flop can be constructed using a D flip-flop and other logic gates.

7.7 The gated SR latch in Figure 7.6*a* has unpredictable behavior if the S and R inputs are both equal to 1 when the *Clk* changes to 0. One way to solve this problem is to create a *set-dominant* gated SR latch in which the condition $S = R = 1$ cause the latch to be set to 1. Design a set-dominant gated SR latch and show the circuit.

7.8 Show how a JK flip-flop can be constructed using a T flip-flop and other logic gates.

7.9 Consider the circuit in Figure P7.2. Assume that the two NAND gates have much longer (about four times) propagation delay than the other gates in the circuit. How does this circuit compare with the circuits that we discussed in this chapter?

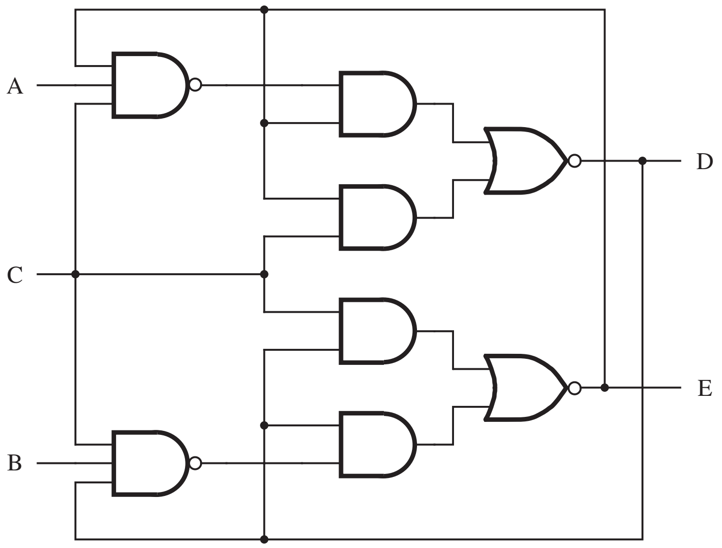


Figure P7.2 Circuit for problem 7.9.

7.10 Write Verilog code that represents a T flip-flop with an asynchronous clear input. Use behavioral code, rather than structural code.

7.11 Write Verilog code that represents a JK flip-flop. Use behavioral code, rather than structural code.

7.12 Synthesize a circuit for the code written for problem 7.11 by using your CAD tools. Simulate the circuit and show a timing diagram that verifies the desired functionality.

7.13 A four-bit barrel shifter is a combinational circuit with four data inputs, two control inputs, and two data outputs. It allows the data inputs to be shifted onto the outputs by 0, 1, 2, or 3 bit positions, with the rightmost bits wrapping around (rotating) to the leftmost bits. For example, if the data inputs are 1100 and the control input specifies a two-bit shift, then the output would be 0011. If the data input is 1110, a two-bit rotation produces 1011. Design a four-bit shift register using a barrel shifter that can shift to the right by 0, 1, 2, or 3 positions.

7.14 Write Verilog code for the shift register described in problem 7.13.

7.15 Design a four-bit synchronous counter with parallel load. Use T flip-flops, instead of the D flip-flops used in section 7.9.3.

7.16 Design a three-bit up/down counter using T flip-flops. It should include a control input called $\overline{\text{Up}}/\text{Down}$. If $\overline{\text{Up}}/\text{Down} = 0$, then the circuit should behave as an up-counter. If $\overline{\text{Up}}/\text{Down} = 1$, then the circuit should behave as a down-counter.

7.17 Repeat problem 7.16 using D flip-flops.

7.18 The circuit in Figure P7.3 looks like a counter. What is the sequence that this circuit counts in?

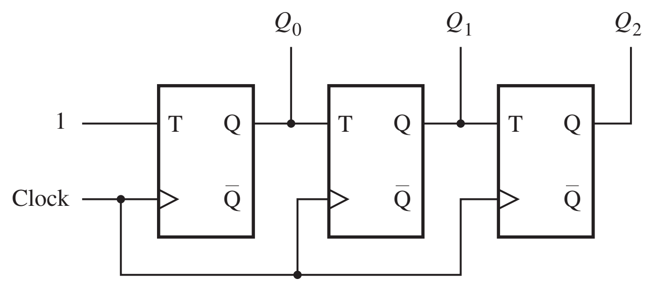


Figure P7.3 The circuit for problem 7.18.

7.19 Consider the circuit in Figure P7.4. How does this circuit compare with the circuit in Figure 7.17? Can the circuits be used for the same purposes? If not, what is the key difference between them?

7.20 Construct a NOR-gate circuit, similar to the one in Figure 7.11*a*, which implements a negative-edge-triggered D flip-flop.

7.21 Write Verilog code that represents a modulo-12 up-counter with synchronous reset.

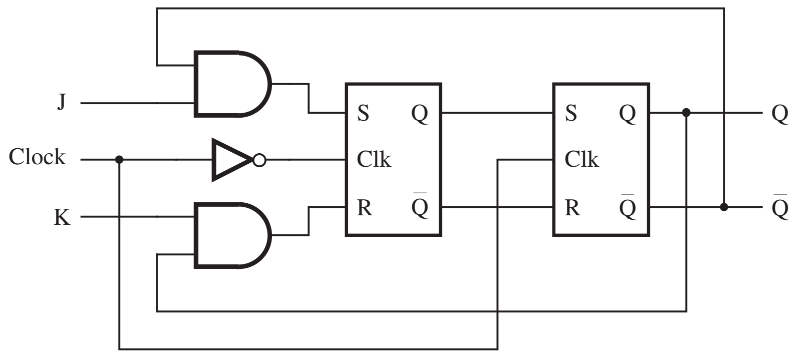


Figure P7.4 Circuit for problem 7.19.

7.22 For the flip-flops in the counter in Figure 7.25, assume that $t_{su} = 3$ ns, $t_h = 1$ ns, and the propagation delay through a flip-flop is 1 ns. Assume that each AND gate, XOR gate, and 2-to-1 multiplexer has the propagation delay equal to 1 ns. What is the maximum clock frequency for which the circuit will operate correctly?

7.23 Write Verilog code that represents an eight-bit Johnson counter. Synthesize the code with your CAD tools and give a timing simulation that shows the counting sequence.

7.24 Write Verilog code in the style shown in Figure 7.55 that represents a ring counter. Your code should have a parameter n that sets the number of flip-flops in the counter.

7.25 Write Verilog code that describes the functionality of the circuit shown in Figure 7.48.

7.26 Write Verilog code that instantiates the *lpm_counter* module from the LPM library. Configure the module as a 32-bit up-counter. For the counter circuit in Figure 7.24, we said that the AND-gate chain can be thought of as the carry-chain. The FLEX 10K FPGA contains special-purpose logic to implement this carry-chain such that it has minimal propagation delay. Use the MAX+plusII synthesis options to implement the *lpm_counter* in two ways: with the dedicated carry-chain used and with the dedicated carry-chain not used. Use the Timing Analyzer in MAX+plusII to determine the maximum speed of operation of the counter in both cases. See the tutorials in Appendices B, C, and D for instructions on using the appropriate features of the CAD tools.

7.27 Figure 7.69 gives Verilog code for a digital system that swaps the contents of two registers, $R1$ and $R2$, using register $R3$ for temporary storage. Create an equivalent schematic using your CAD tools for this system. Synthesize a circuit for this schematic and perform a timing simulation.

7.28 Repeat problem 7.27 using the control circuit in Figure 7.63.

7.29 Modify the code in Figure 7.71 to use the control circuit in Figure 7.63. Synthesize the code for implementation in a chip and perform a timing simulation.

7.30 In section 7.14.2 we designed a processor that performs the operations listed in Table 7.3. Design a modified circuit that performs an additional operation Swap Rx, Ry. This operation swaps the contents of registers *Rx* and *Ry*. Use three bits $f_2 f_1 f_0$ to represent the input *F* shown in Figure 7.75 because there are now five operations, rather than four. Add a new register, named *Tmp*, into the system, to be used for temporary storage during the swap operation. Show logic expressions for the outputs of the control circuit, as was done in section 7.14.2.

7.31 A ring oscillator is a circuit that has an odd number, n, of inverters connected in a ringlike structure, as shown in Figure P7.5. The output of each inverter is a periodic signal with a certain period.
(a) Assume that all the inverters are identical; hence they all have the same delay, t_p. Let the output of one of the inverters be named f. Give an equation that expresses the period of the signal f in terms of n and t_p.
(b) For this part you are to design a circuit that can be used to experimentally measure the delay t_p through one of the inverters in the ring oscillator. Assume the existence of an input called *Reset* and another called *Interval*. The timing of these two signals is shown in Figure P7.6. The length of time for which *Interval* has the value 1 is known. Assume that this length of time is 100 ns. Design a circuit that uses the *Reset* and *Interval* signals and the signal f from part (a) to experimentally measure t_p. In your design you may use logic gates and subcircuits such as adders, flip-flops, counters, registers, and so on.

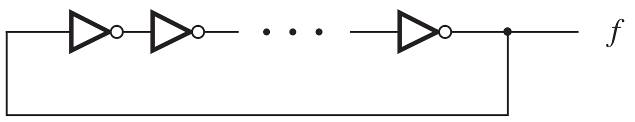


Figure P7.5 A ring oscillator.

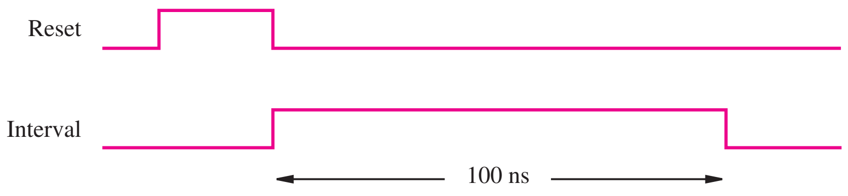
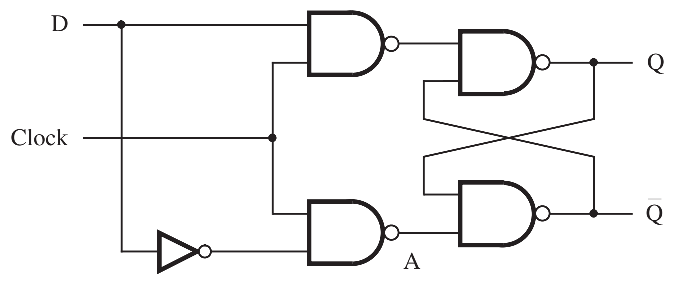


Figure P7.6 Timing of signals for problem 7.31.

7.32 A circuit for a gated D latch is shown in Figure P7.7. Assume that the propagation delay through either a NAND gate or an inverter is 1 ns. Complete the timing diagram given in the figure, which shows the signal values with 1 ns resolution.

7.33 A logic circuit has two inputs, *Clock* and *Start*, and two outputs, f and g. The behavior of the circuit is described by the timing diagram in Figure P7.8. When a pulse is received on the *Start* input, the circuit produces pulses on the f and g outputs as shown in the



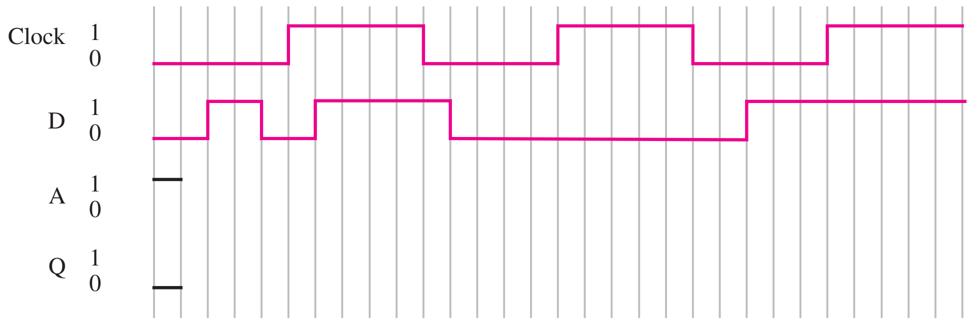


Figure P7.7 Circuit and timing diagram for problem 7.32.

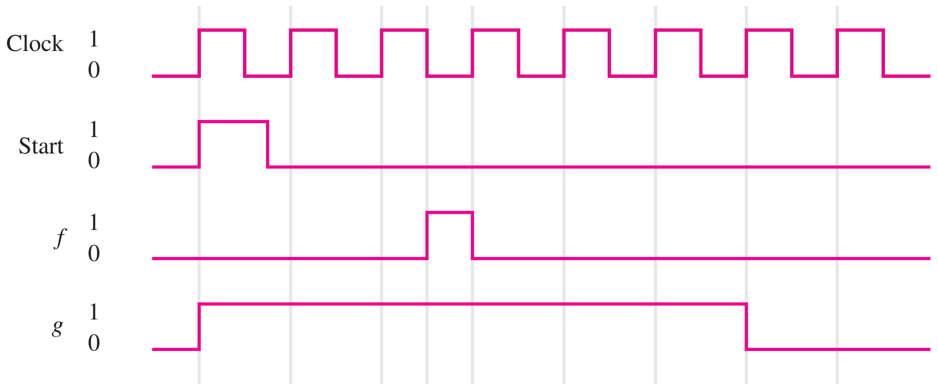


Figure P7.8 Timing diagram for problem 7.33.

timing diagram. Design a suitable circuit using only the following components: a three-bit resettable positive-edge-triggered synchronous counter and basic logic gates. For your answer assume that the delays through all logic gates and the counter are negligible.

7.34 The following code checks for adjacent ones in an n-bit vector.

```
always @(A)
begin
    f = A[1] & A[0];
    for (k = 2; k < n; k = k+1)
        f = f | (A[k] & A[k−1]);
end
```

With blocking assignments this code produces the desired logic function, which is $f = a_1a_0 + \cdots + a_{n-1}a_{n-2}$. What logic function is produced if we change the code to use non-blocking assignments?

7.35 The Verilog code in Figure P7.9 represents a 3-bit *linear-feedback shift register* (LFSR). This type of circuit generates a counting sequence of pseudo-random numbers that repeats after $2^n - 1$ clock cycles, where n is the number of flip-flops in the LFSR. Synthesize a circuit to implement the LFSR in a chip. Draw a diagram of the circuit. Simulate the circuit's behavior by loading the pattern 001 into the LFSR and then enabling the register to count. What is the counting sequence?

```
module lfsr (R, L, Clock, Q);
    input [0:2] R;
    input L, Clock;
    output [0:2] Q;
    reg [0:2] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
            Q <= {Q[2], Q[0] ^ Q[2], Q[1]};

endmodule
```

Figure P7.9 Code for a linear-feedback shift register.

7.36 Repeat problem 7.35 for the Verilog code in Figure P7.10.

7.37 The Verilog code in Figure P7.11 is equivalent to the code in Figure P7.9, except that blocking assignments are used. Draw the circuit represented by this code. What is its counting sequence?

7.38 The Verilog code in Figure P7.12 is equivalent to the code in Figure P7.10, except that blocking assignments are used. Draw the circuit represented by this code. What is its counting sequence?

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output [0:2] Q;
   reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
         Q <= {Q[2], Q[0], Q[1] ^ Q[2]};

endmodule
```

Figure P7.10 Code for a linear-feedback shift register.

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output [0:2] Q;
   reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] = Q[2];
         Q[1] = Q[0] ^ Q[2];
         Q[2] = Q[1];
      end

endmodule
```

Figure P7.11 Code for problem 7.37.

7.39 A universal shift register can shift in both the left-to-right and right-to-left directions, and it has parallel-load capability. Draw a circuit for such a shift register.

7.40 Write Verilog code for a universal shift register with *n* bits.

```
module lfsr (R, L, Clock, Q);
  input [0:2] R;
  input L, Clock;
  output [0:2] Q;
  reg [0:2] Q;

  always @(posedge Clock)
    if (L)
      Q <= R;
    else
    begin
      Q[0] = Q[2];
      Q[1] = Q[0];
      Q[2] = Q[1] ^ Q[2];
    end

endmodule
```

Figure P7.12 Code for problem 7.38.

REFERENCES

1. V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed., (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 2nd ed., (Morgan Kaufmann: San Francisco, CA, 1998).
3. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
4. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
5. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).
6. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, NJ, 1995).
7. R. H. Katz, *Contemporary Logic Design*, (Benjamin/Cummings: Redwood City, CA, 1994).
8. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed., (West: St. Paul, MN, 1993).
10. J. F. Wakerly, *Digital Design Principles and Practices*, (Prentice-Hall: Englewood Cliffs, NJ, 1990).

11. E. J. McCluskey, *Logic Design Principles*, (Prentice-Hall: Englewood Cliffs, NJ, 1986).
12. Institute of Electrical and Electronics Engineers, *IEEE Standard Verilog Hardware Description Language Reference Manual*, (IEEE: Piscataway, NJ, 1995).
13. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 4th ed., (Kluwer: Norwell, MA, 1998).
14. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, (Prentice-Hall: Upper Saddle River, NJ, 1996).
15. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
16. Z. Navabi, *Verilog Digital System Design*, (McGraw-Hill: New York, 1999).
17. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
18. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
19. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).