Chapter 5

Synchronous Sequential Logic

5.1 INTRODUCTION

The digital circuits considered thus far have been combinational; that is, the outputs are entirely dependent on the current inputs. Although every digital system is likely to have some combinational circuits, most systems encountered in practice also include storage elements, which require that the system be described in terms of *sequential logic*. First, we need to understand what distinguishes sequential logic from combinational logic.

5.2 SEQUENTIAL CIRCUITS

A block diagram of a sequential circuit is shown in Fig. 5.1. It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the *state* of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the

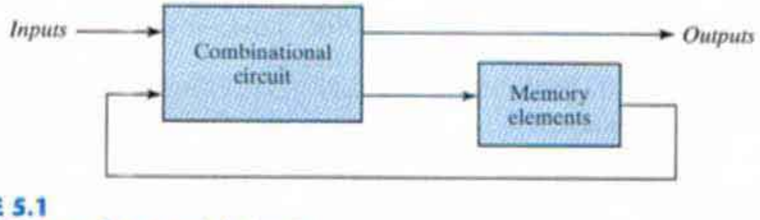


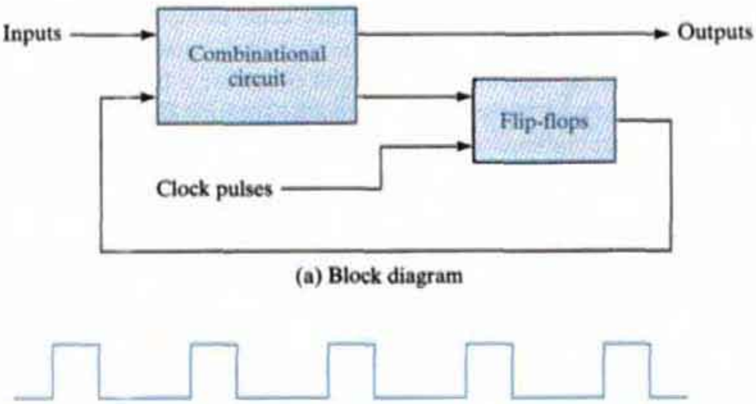
FIGURE 5.1
Block diagram of sequential circuit

storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state in the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, a sequential circuit is specified by a time sequence of inputs, outputs, and internal states. In contrast, the outputs of combinational logic depend only on the present values of the inputs.

There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A *synchronous* sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an *asynchronous* sequential circuit depends upon the input signals at any instant of time *and* the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are time-delay devices. The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device. In practice, the internal propagation delay of logic gates is of sufficient duration to produce the needed delay, so that actual delay units may not be necessary. In gate-type asynchronous systems, the storage elements consist of logic gates whose propagation delay provides the required storage. Thus, an asynchronous sequential circuit may be regarded as a combinational circuit with feedback. Because of the feedback among logic gates, an asynchronous sequential circuit may become unstable at times. The instability problem imposes many difficulties on the designer. Asynchronous sequential circuits are presented in Chapter 9.

A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time. Synchronization is achieved by a timing device called a *clock generator*, which provides a clock signal having the form of a periodic train of *clock pulses*. The clock signal is commonly denoted by the identifiers *clock* and *clk*. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse. In practice, the clock pulses determine *when* computational activity will occur within the circuit, and other signals (external inputs and otherwise) determine *what* changes will take place affecting the storage elements and the outputs. For example, a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse. Synchronous sequential circuits that use clock pulses to control storage elements are called *clocked sequential circuits* and are the type most frequently encountered in practice. They are called *synchronous circuits* because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses. The design of synchronous circuits is feasible because they seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which can be considered separately.

The storage elements (memory) used in clocked sequential circuits are called *flip-flops*. A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. The block diagram of a synchronous clocked sequential circuit is shown in Fig. 5.2. The *outputs* are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both). The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently



(a) Block diagram

(b) Timing diagram of clock pulses

FIGURE 5.2
Synchronous clocked sequential circuit

stored in the flip-flop (or both). The new value is stored (i.e., the flip-flop is updated) when a pulse of the clock signal occurs. Prior to the occurrence of the clock pulse, the combinational logic forming the next value of the flop-flop must have reached a stable value. Consequently, the speed at which the combinational logic circuits operate is critical. If the clock (synchronizing) pulses arrive at a regular interval, as shown in the timing diagram in Fig. 5.2, the combinational logic must respond to a change in the state of the flip-flop in time to be updated before the next pulse arrives. Propagation delays play an important role in determining the minimum interval between clock pulses that will allow the circuit to operate correctly. The state of the flip-flops can change only during a clock pulse transition—for example, when the value of the clock signals changes from 0 to 1. When a clock pulse is not active, the feedback loop between the value stored in the flip-flop and the value formed at the input to the flip-flop is effectively broken because the flip-flop outputs cannot change even if the outputs of the combinational circuit driving their inputs change in value. Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses.

5.3 STORAGE ELEMENTS: LATCHES

A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states. The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state. *Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches; those controlled by a clock transition are flip-flops.* Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices. The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed. Although latches are useful for storing binary information and for the design of asynchronous sequential circuits (see Section 9.3), they are not practical for use in synchronous sequential circuits. Because they are the building blocks of flip-flops, however, we will consider the fundamental storage mechanism used in latches before considering flip-flops in the next section.

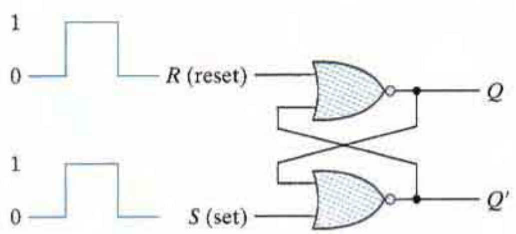
SR Latch

The *SR* latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled S for set and R for reset. The *SR* latch constructed with two cross-coupled NOR gates is shown in Fig. 5.3. The latch has two useful states. When output $Q = 1$ and $Q' = 0$, the latch is said to be in the *set state*. When $Q = 0$ and $Q' = 1$, it is in the *reset state*. Outputs Q and Q' are normally the complement of each other. However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary) occurs. If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a metastable state. Consequently, in practical applications, setting both inputs to 1 is forbidden.

Under normal conditions, both inputs of the latch remain at 0 unless the state has to be changed. The application of a momentary 1 to the S input causes the latch to go to the set state. The S input must go back to 0 before any other changes take place, in order to avoid the occurrence of an undefined next state that results from the forbidden input condition. As shown in the function table of Fig. 5.3(b), two input conditions cause the circuit to be in the set state. The first condition ($S = 1, R = 0$) is the action that must be taken by input S to bring the circuit to the set state. Removing the active input from S leaves the circuit in the same state. After both inputs return to 0, it is then possible to shift to the reset state by momentary applying a 1 to the R input. The 1 can then be removed from R, whereupon the circuit remains in the reset state. Thus, when both inputs S and R are equal to 0, the latch can be in either the set or the reset state, depending on which input was most recently a 1.

If a 1 is applied to both the S and R inputs of the latch, both outputs go to 0. This action produces an undefined next state, because the state that results from the input transitions depends on the order in which they return to 0. It also violates the requirement that outputs be the complement of each other. In normal operation, this condition is avoided by making sure that 1's are not applied to both inputs simultaneously.

The *SR* latch with two cross-coupled NAND gates is shown in Fig. 5.4. It operates with both inputs normally at 1, unless the state of the latch has to be changed. The application of 0

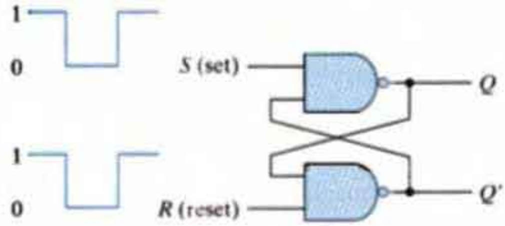


(a) Logic diagram

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

FIGURE 5.3
SR latch with NOR gates



(a) Logic diagram

S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

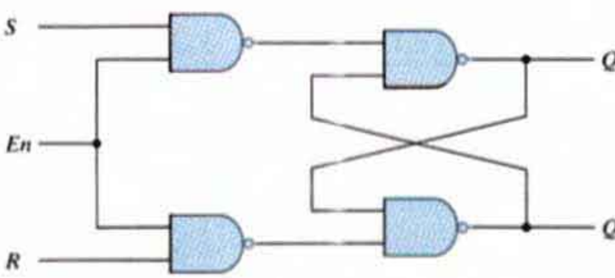
(b) Function table

FIGURE 5.4
***SR* latch with NAND gates**

to the S input causes output Q to go to 1, putting the latch in the set state. When the S input goes back to 1, the circuit remains in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the R input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.

In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an $S'R'$ latch. The primes (or, sometimes, bars over the letters) designate the fact that the inputs must be in their complement form to activate the circuit.

The operation of the basic *SR* latch can be modified by providing an additional input signal that determines (controls) *when* the state of the latch can be changed. An *SR* latch with a control input is shown in Fig. 5.5. It consists of the basic *SR* latch and two additional NAND gates. The control input *En* acts as an *enable* signal for the other two inputs. The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0. This is the quiescent condition for the *SR* latch. When the enable input goes to 1, information from the S or R input is allowed to affect the latch. The set state is reached with $S = 1$, $R = 0$, and $En = 1$ (active-high enabled). To change to the reset state, the inputs must be $S = 0$, $R = 1$, and

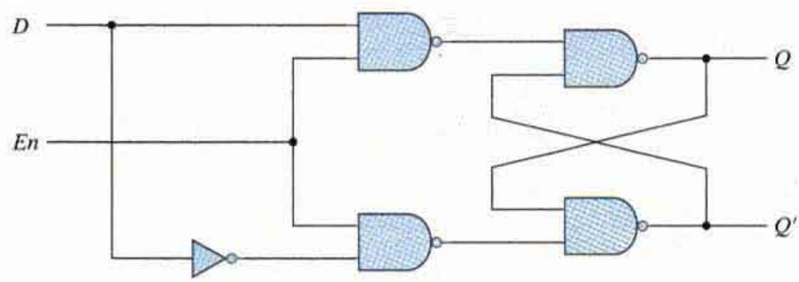


(a) Logic diagram

En	*S*	*R*	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

FIGURE 5.5
***SR* latch with control input**



(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

(b) Function table

FIGURE 5.6
D latch

$En = 1$. In either case, when En returns to 0, the circuit remains in its current state. The control input disables the circuit by applying 0 to En, so that the state of the output does not change regardless of the values of S and R. Moreover, when $En = 1$ and both the S and R inputs are equal to 0, the state of the circuit does not change. These conditions are listed in the function table accompanying the diagram.

An indeterminate condition occurs when all three inputs are equal to 1. This condition places 0's on both inputs of the basic SR latch, which puts it in the undefined state. When the enable input goes back to 0, one cannot conclusively determine the next state, because it depends on whether the S or R input goes to 0 first. This indeterminate condition makes this circuit difficult to manage, and it is seldom used in practice. Nevertheless, it is an important circuit because other useful latches and flip-flops are constructed from it.

D Latch (Transparent Latch)

One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs S and R are never equal to 1 at the same time. This is done in the D latch, shown in Fig. 5.6. This latch has only two inputs: D (data) and En (enable). The D input goes directly to the S input, and its complement is applied to the R input. As long as the enable input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of D. The D input is sampled when $En = 1$. If $D = 1$, the Q output goes to 1, placing the circuit in the set state. If $D = 0$, output Q goes to 0, placing the circuit in the reset state.

The D latch receives that designation from its ability to hold *data* in its internal storage. It is suited for use as a temporary storage for binary information between a unit and its environment. The binary information present at the data input of the D latch is transferred to the Q output when the enable input is asserted. The output follows changes in the data input as long as the enable input is asserted. This situation provides a path from input D to the output, and for this reason, the circuit is often called a *transparent* latch. When the enable input signal is deasserted, the binary information that was present at the data input at the time the transition occurred is retained (i.e., stored) at the Q output until the enable input is asserted again. Note that

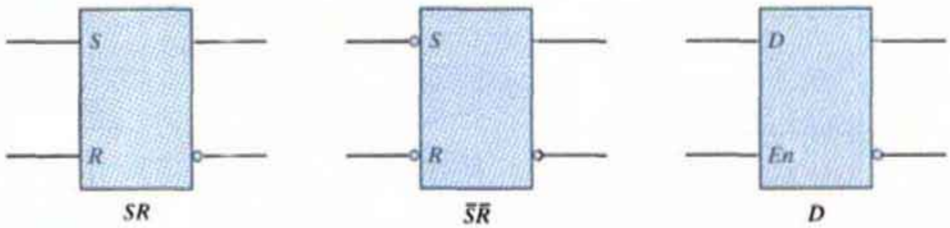


FIGURE 5.7
Graphic symbols for latches

an inverter could be placed at the enable input. Then, depending on the physical circuit, the external enabling signal will be a value of 0 (active low) or 1 (active high).

The graphic symbols for the various latches are shown in Fig. 5.7. A latch is designated by a rectangular block with inputs on the left and outputs on the right. One output designates the normal output, and the other (with the bubble designation) designates the complement output. The graphic symbol for the *SR* latch has inputs *S* and *R* indicated inside the block. In the case of a NAND gate latch, bubbles are added to the inputs to indicate that setting and resetting occur with a logic-0 signal. The graphic symbol for the *D* latch has inputs *D* and *En* indicated inside the block.

5.4 STORAGE ELEMENTS: FLIP-FLOPS

The state of a latch or flip-flop is switched by a change in the control input. This momentary change is called a *trigger*, and the transition it causes is said to trigger the flip-flop. The *D* latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level. As long as the pulse input remains at this level, any changes in the data input will change the output and the state of the latch.

As seen from the block diagram of Fig. 5.2, a sequential circuit has a feedback path from the outputs of the flip-flops to the input of the combinational circuit. Consequently, the inputs of the flip-flops are derived in part from the outputs of the same and other flip-flops. When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. The new state of a latch appears at the output while the pulse is still active. This output is connected to the inputs of the latches through the combinational circuit. If the inputs applied to the latches change while the clock pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur. The result is an unpredictable situation, since the state of the latches may keep changing for as long as the clock pulse stays at the active level. Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a common clock source.

Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a common clock. The problem with the latch is that it responds to a change in the *level* of a clock pulse. As shown in Fig. 5.8(a), a positive level response in the enable input allows changes in the output when the *D* input changes while the

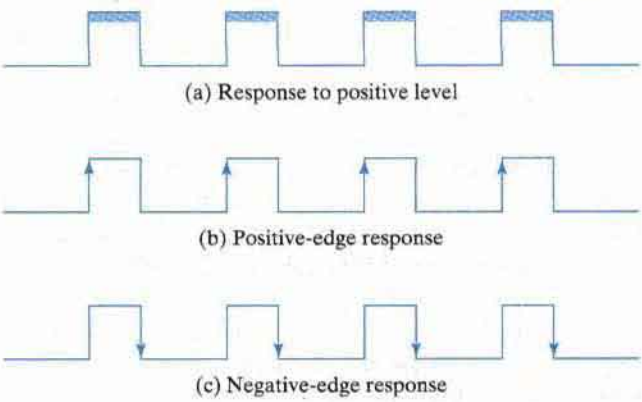


FIGURE 5.8
Clock response in latch and flip-flop

clock pulse stays at logic 1. The key to the proper operation of a flip-flop is to trigger it only during a signal *transition*. This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches. A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0. As shown in Fig. 5.8, the positive transition is defined as the positive edge and the negative transition as the negative edge. There are two ways that a latch can be modified to form a flip-flop. One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing. Another way is to produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse. We will now proceed to show the implementation of both types of flip-flops.

Edge-Triggered *D* Flip-Flop

The construction of a D flip-flop with two D latches and an inverter is shown in Fig. 5.9. The first latch is called the master and the second the slave. The circuit samples the D input and changes its output Q only at the negative edge of the synchronizing or controlling clock (designated as

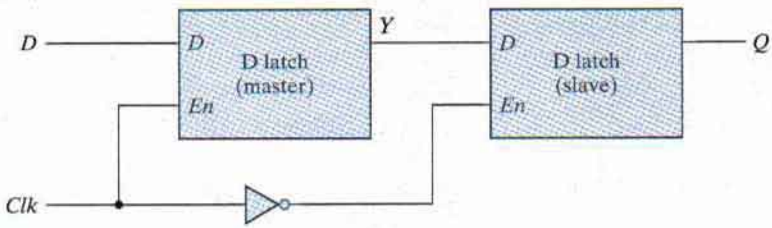


FIGURE 5.9
Master–slave *D* flip-flop

Clk). When the clock is 0, the output of the inverter is 1. The slave latch is enabled, and its output Q is equal to the master output Y. The master latch is disabled because $Clk = 0$. When the input pulse changes to the logic-1 level, the data from the external D input are transferred to the master. The slave, however, is disabled as long as the clock remains at the 1 level, because its *enable* input is equal to 0. Any change in the input changes the master output at Y, but cannot affect the slave output. When the clock pulse returns to 0, the master is disabled and is isolated from the D input. At the same time, the slave is enabled and the value of Y is transferred to the output of the flip-flop at Q. Thus, a change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1 to 0.

The behavior of the master–slave flip-flop just described dictates that (1) the output may change only once, (2) a change in the output is triggered by the negative edge of the clock, and (3) the change may occur only during the clock's negative level. The value that is produced at the output of the flip-flop is the value that was stored in the master stage immediately before the negative edge occurred. It is also possible to design the circuit so that the flip-flop output changes on the positive edge of the clock. This happens in a flip-flop that has an additional inverter between the *Clk* terminal and the junction between the other inverter and input *En* of the master latch. Such a flip-flop is triggered with a negative pulse, so that the negative edge of the clock affects the master and the positive edge affects the slave and the output terminal.

Another construction of an edge-triggered D flip-flop uses three SR latches as shown in Fig. 5.10. Two latches respond to the external D (data) and *Clk* (clock) inputs. The third latch provides the outputs for the flip-flop. The S and R inputs of the output latch are maintained at the logic-1 level when $Clk = 0$. This causes the output to remain in its present state. Input D

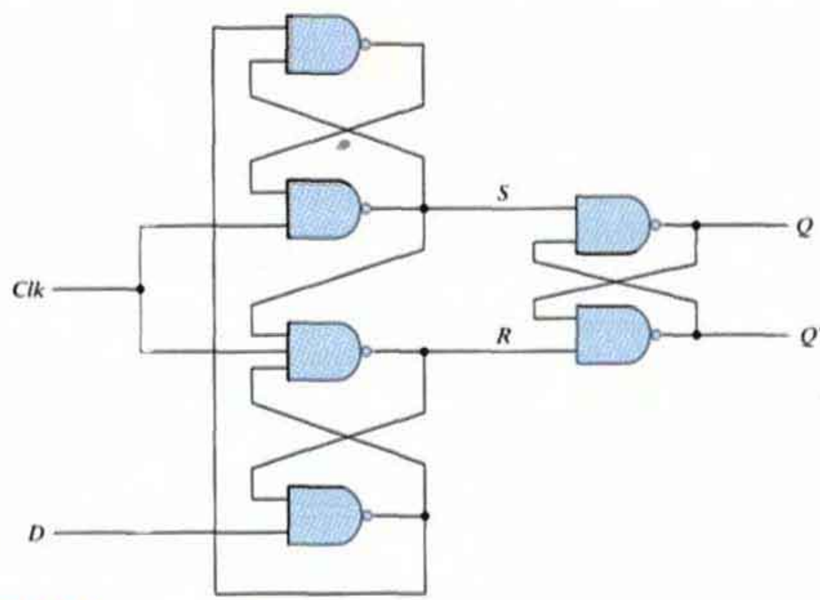


FIGURE 5.10
***D*-type positive-edge-triggered flip-flop**

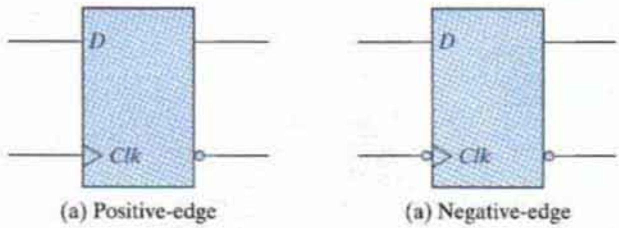


FIGURE 5.11
Graphic symbol for edge-triggered *D* flip-flop

may be equal to 0 or 1. If $D = 0$ when *Clk* becomes 1, *R* changes to 0. This causes the flip-flop to go to the reset state, making $Q = 0$. If there is a change in the *D* input while $Clk = 1$, terminal *R* remains at 0 because *Q* is 0. Thus, the flip-flop is locked out and is unresponsive to further changes in the input. When the clock returns to 0, *R* goes to 1, placing the output latch in the quiescent condition without changing the output. Similarly, if $D = 1$ when *Clk* goes from 0 to 1, *S* changes to 0. This causes the circuit to go to the set state, making $Q = 1$. Any change in *D* while $Clk = 1$ does not affect the output.

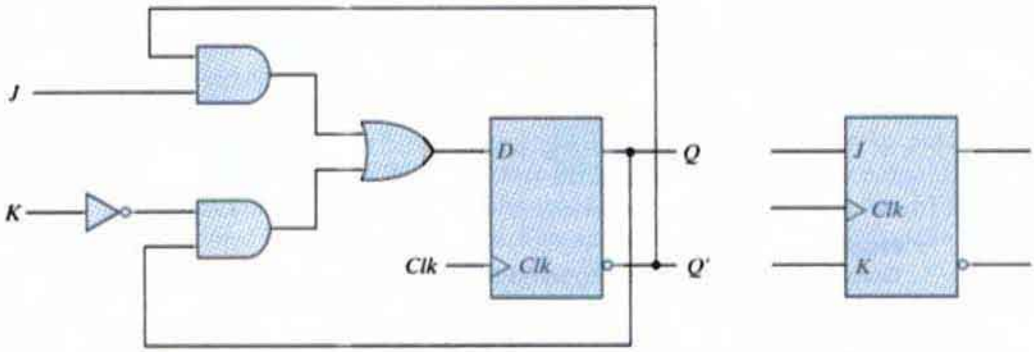
In sum, when the input clock in the positive-edge-triggered flip-flop makes a positive transition, the value of *D* is transferred to *Q*. A negative transition of the clock (i.e., from 1 to 0) does not affect the output, nor is the output affected by changes in *D* when *Clk* is in the steady logic-1 level or the logic-0 level. Hence, this type of flip-flop responds to the transition from 0 to 1 and nothing else.

The timing of the response of a flip-flop to input data and to the clock must be taken into consideration when one is using edge-triggered flip-flops. There is a minimum time called the *setup time* during which the *D* input must be maintained at a constant value prior to the occurrence of the clock transition. Similarly, there is a minimum time called the *hold time* during which the *D* input must not change after the application of the positive transition of the clock. The propagation delay time of the flip-flop is defined as the interval between the trigger edge and the stabilization of the output to a new state. These and other parameters are specified in manufacturers' data books for specific logic families.

The graphic symbol for the edge-triggered *D* flip-flop is shown in Fig. 5.11. It is similar to the symbol used for the *D* latch, except for the arrowheadlike symbol in front of the letter *Clk*, designating a *dynamic* input. The *dynamic indicator* denotes the fact that the flip-flop responds to the edge transition of the clock. A bubble outside the block adjacent to the dynamic indicator designates a negative edge for triggering the circuit. The absence of a bubble designates a positive-edge response.

Other Flip-Flops

Very large-scale integration circuits contain thousands of gates within one package. Circuits are constructed by interconnecting the various gates to provide a digital system. Each flip-flop is constructed from an interconnection of gates. The most economical and efficient flip-flop constructed in this manner is the edge-triggered *D* flip-flop, because it requires the smallest number



(a) Circuit diagram

(b) Graphic symbol

FIGURE 5.12
JK flip-flop

of gates. Other types of flip-flops can be constructed by using the *D* flip-flop and external logic. Two flip-flops less widely used in the design of digital systems are the *JK* and *T* flip-flops.

There are three operations that can be performed with a flip-flop: Set it to 1, reset it to 0, or complement its output. With only a single input, the *D* flip-flop can set or reset the output, depending on the value of the *D* input immediately before the clock transition. Synchronized by a clock signal, the *JK* flip-flop has two inputs and performs all three operations. The circuit diagram of a *JK* flip-flop constructed with a *D* flip-flop and gates is shown in Fig. 5.12(a). The *J* input sets the flip-flop to 1, the *K* input resets it to 0, and when both inputs are enabled, the output is complemented. This can be verified by investigating the circuit applied to the *D* input:

$$D = JQ' + K'Q$$

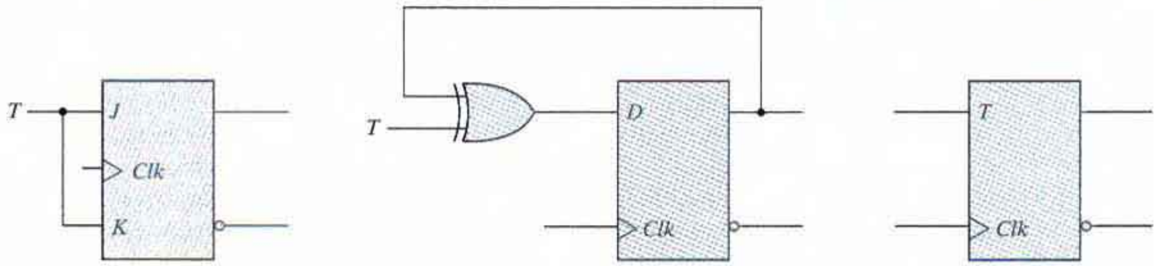
When $J = 1$ and $K = 0$, $D = Q' + Q = 1$, so the next clock edge sets the output to 1. When $J = 0$ and $K = 1$, $D = 0$, so the next clock edge resets the output to 0. When both $J = K = 1$ and $D = Q'$, the next clock edge complements the output. When both $J = K = 0$ and $D = Q$, the clock edge leaves the output unchanged. The graphic symbol for the *JK* flip-flop is shown in Fig. 5.12(b). It is similar to the graphic symbol of the *D* flip-flop, except that now the inputs are marked *J* and *K*.

The *T* (toggle) flip-flop is a complementing flip-flop and can be obtained from a *JK* flip-flop when inputs *J* and *K* are tied together. This is shown in Fig. 5.13(a). When $T = 0$ ($J = K = 0$), a clock edge does not change the output. When $T = 1$ ($J = K = 1$), a clock edge complements the output. The complementing flip-flop is useful for designing binary counters.

The *T* flip-flop can be constructed with a *D* flip-flop and an exclusive-OR gate as shown in Fig. 5.13(b). The expression for the *D* input is

$$D = T \oplus Q = TQ' + T'Q$$

When $T = 0$, $D = Q$ and there is no change in the output. When $T = 1$, $D = Q'$ and the output complements. The graphic symbol for this flip-flop has a *T* symbol in the input.



(a) From *JK* flip-flop (b) From *D* flip-flop (c) Graphic symbol

FIGURE 5.13
T flip-flop

Characteristic Tables

A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. The characteristic tables of three types of flip-flops are presented in Table 5.1. They define the next state (i.e., the state that results from a clock transition) as a function of the inputs and the present state. $Q(t)$ refers to the present state (i.e., the state present prior to the application of a clock edge). $Q(t + 1)$ is the next state one clock period later. Note that the clock edge input is not included in the characteristic table, but is implied to occur between times t and $t + 1$. Thus, $Q(t)$ denotes the state of the flip-flop immediately before the clock edge, and $Q(t + 1)$ denotes the state that results from the clock transition.

The characteristic table for the *JK* flip-flop shows that the next state is equal to the present state when inputs J and K are both equal to 0. This condition can be expressed as $Q(t + 1) = Q(t)$, indicating that the clock produces no change of state. When $K = 1$ and

Table 5.1
Flip-Flop Characteristic Tables

JK Flip-Flop

J	K	Q(t + 1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

D Flip-Flop

D	Q(t + 1)	
0	0	Reset
1	1	Set

T Flip-Flop

T	Q(t + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

$J = 0$, the clock resets the flip-flop and $Q(t + 1) = 0$. With $J = 1$ and $K = 0$, the flip-flop sets and $Q(t + 1) = 1$. When both J and K are equal to 1, the next state changes to the complement of the present state, a transition that can be expressed as $Q(t + 1) = Q'(t)$.

The next state of a D flip-flop is dependent only on the D input and is independent of the present state. This can be expressed as $Q(t + 1) = D$. It means that the next-state value is equal to the value of D. Note that the D flip-flop does not have a "no-change" condition. Such a condition can be accomplished either by disabling the clock or by operating the clock by having the output of the flip-flop connected into the D input. Either method effectively circulates the output of the flip-flop when the state of the flip-flop must remain unchanged.

The characteristic table of the T flip-flop has only two conditions: When $T = 0$, the clock edge does not change the state; when $T = 1$, the clock edge complements the state of the flip-flop.

Characteristic Equations

The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation. For the D flip-flop, we have the characteristic equation

$$Q(t + 1) = D$$

which states that the next state of the output will be equal to the value of input D in the present state. The characteristic equation for the JK flip-flop can be derived from the characteristic table or from the circuit of Fig. 5.12. We obtain

$$Q(t + 1) = JQ' + K'Q$$

where Q is the value of the flip-flop output prior to the application of a clock edge. The characteristic equation for the T flip-flop is obtained from the circuit of Fig. 5.13:

$$Q(t + 1) = T \oplus Q = TQ' + T'Q$$

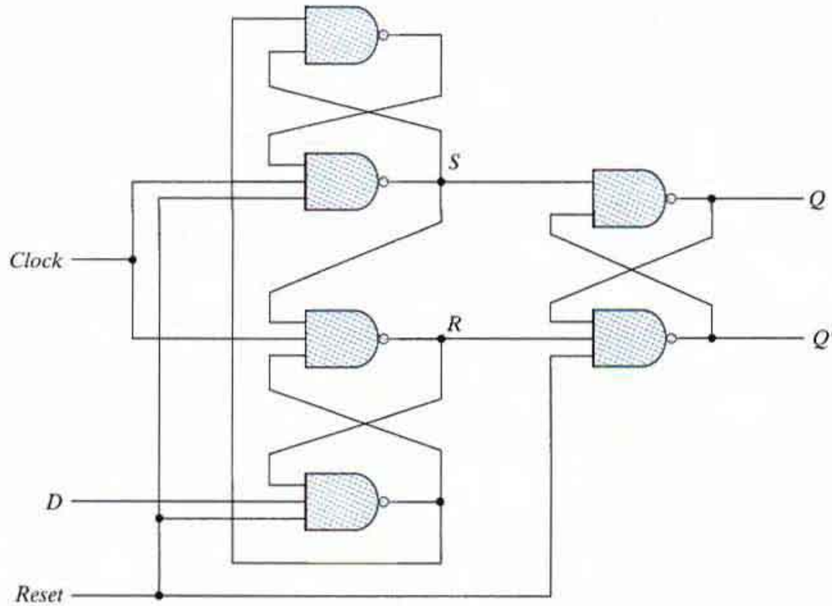
Direct Inputs

Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock. The input that sets the flip-flop to 1 is called *preset* or *direct set*. The input that clears the flip-flop to 0 is called *clear* or *direct reset*. When power is turned on in a digital system, the state of the flip-flops is unknown. The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.

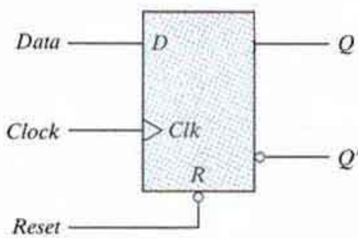
A positive-edge-triggered D flip-flop with active-low asynchronous reset is shown in Fig. 5.14. The circuit diagram is the same as the one in Fig. 5.10, except for the additional reset input connections to three NAND gates. When the reset input is 0, it forces output Q' to stay at 1, which, in turn, clears output Q to 0, thus resetting the flip-flop. Two other connections from the reset input ensure that the S input of the third SR latch stays at logic 1 while the reset input is at 0, regardless of the values of D and Clk.

The graphic symbol for the D flip-flop with a direct reset has an additional input marked with R. The bubble along the input indicates that the reset is active at the logic-0 level. Flip-flops with a direct set use the symbol S for the asynchronous set input.

The function table specifies the operation of the circuit. When $R = 0$, the output is reset to 0. This state is independent of the values of D or Clk. Normal clock operation can proceed only



(a) Circuit diagram



(b) Graphic symbol

R	Clk	D	Q	Q'
0	X	X	0	1
0	↑	0	0	1
0	↑	1	1	0

(b) Function table

FIGURE 5.14
***D* flip-flop with asynchronous reset**

after the reset input goes to logic 1. The clock at *Clk* is shown with an upward arrow to indicate that the flip-flop triggers on the positive edge of the clock. The value in *D* is transferred to *Q* with every positive-edge clock signal, provided that $R = 1$.

5.5 ANALYSIS OF CLOCKED SEQUENTIAL CIRCUITS

Analysis describes what a given circuit will do under certain operating conditions. The behavior of a clocked sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops. The outputs and the next state are both a function of the inputs and the present state. The analysis of a sequential circuit consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states. It is also possible to write Boolean expressions that describe the behavior of the sequential circuit. These expressions must include the necessary time sequence, either directly or indirectly.

A logic diagram is recognized as a clocked sequential circuit if it includes flip-flops with clock inputs. The flip-flops may be of any type, and the logic diagram may or may not include combinational circuit gates. In this section, we introduce an algebraic representation for specifying the next-state condition in terms of the present state and inputs. A state table and state diagram are then presented to describe the behavior of the sequential circuit. Another algebraic representation is introduced for specifying the logic diagram of sequential circuits. Examples are used to illustrate the various procedures.

State Equations

The behavior of a clocked sequential circuit can be described algebraically by means of state equations. A *state equation* (also called a *transition equation*) specifies the next state as a function of the present state and inputs. Consider the sequential circuit shown in Fig. 5.15. It consists

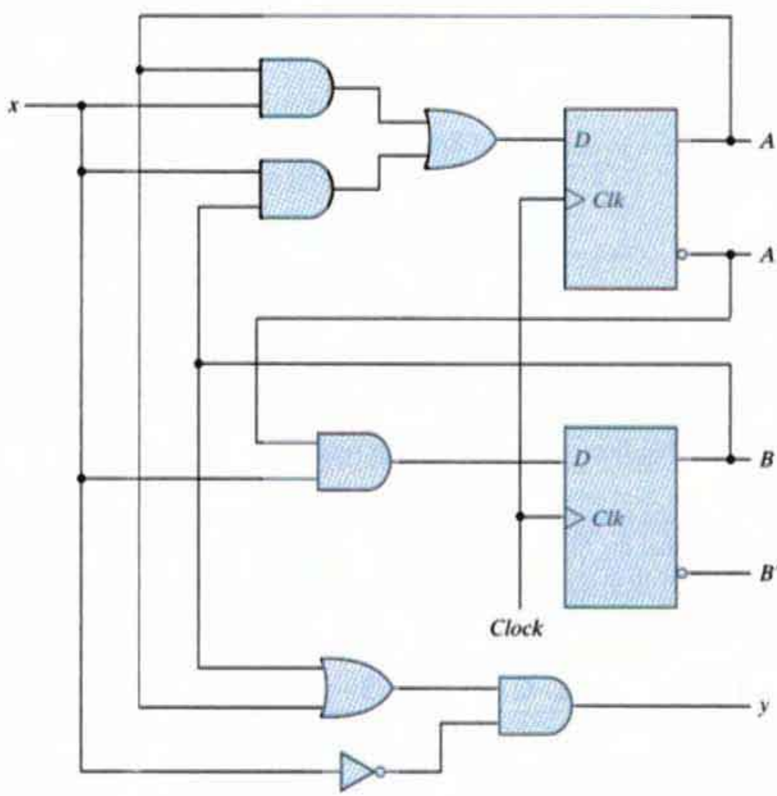


FIGURE 5.15
Example of sequential circuit

of two D flip-flops A and B, an input x and an output y. Since the D input of a flip-flop determines the value of the next state (i.e., the state reached after the clock transition), it is possible to write a set of state equations for the circuit:

$$A(t + 1) = A(t)x(t) + B(t)x(t)$$
$$B(t + 1) = A'(t)x(t)$$

A state equation is an algebraic expression that specifies the condition for a flip-flop state transition. The left side of the equation, with $(t + 1)$, denotes the next state of the flip-flop one clock edge later. The right side of the equation is a Boolean expression that specifies the present state and input conditions that make the next state equal to 1. Since all the variables in the Boolean expressions are a function of the present state, we can omit the designation (t) after each variable for convenience and can express the state equations in the more compact form

$$A(t + 1) = Ax + Bx$$
$$B(t + 1) = A'x$$

The Boolean expressions for the state equations can be derived directly from the gates that form the combinational circuit part of the sequential circuit, since the D values of the combinational circuit determine the next state. Similarly, the present-state value of the output can be expressed algebraically as

$$y(t) = [A(t) + B(t)]x'(t)$$

By removing the symbol (t) for the present state, we obtain the output Boolean equation:

$$y = (A + B)x'$$

State Table

The time sequence of inputs, outputs, and flip-flop states can be enumerated in a *state table* (sometimes called a *transition table*). The state table for the circuit of Fig. 5.15 is shown in Table 5.2.

Table 5.2
State Table for the Circuit of Fig. 5.15

Present State		Input	Next State		Output
A	***B***	***x***	***A***	***B***	***y***
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

The table consists of four sections labeled *present state*, *input*, *next state*, and *output*. The present-state section shows the states of flip-flops A and B at any given time t. The input section gives a value of x for each possible present state. The next-state section shows the states of the flip-flops one clock cycle later, at time $t + 1$. The output section gives the value of y at time t for each present state and input condition.

The derivation of a state table requires listing all possible binary combinations of present states and inputs. In this case, we have eight binary combinations from 000 to 111. The next-state values are then determined from the logic diagram or from the state equations. The next state of flip-flop A must satisfy the state equation

$$A(t + 1) = Ax + Bx$$

The next-state section in the state table under column A has three 1's where the present state of A and input x are both equal to 1 or the present state of B and input x are both equal to 1. Similarly, the next state of flip-flop B is derived from the state equation

$$B(t + 1) = A'x$$

and is equal to 1 when the present state of A is 0 and input x is equal to 1. The output column is derived from the output equation

$$y = Ax' + Bx'$$

The state table of a sequential circuit with D-type flip-flops is obtained by the same procedure outlined in the previous example. In general, a sequential circuit with m flip-flops and n inputs needs 2^{m+n} rows in the state table. The binary numbers from 0 through $2^{m+n} - 1$ are listed under the present-state and input columns. The next-state section has m columns, one for each flip-flop. The binary values for the next state are derived directly from the state equations. The output section has as many columns as there are output variables. Its binary value is derived from the circuit or from the Boolean function in the same manner as in a truth table.

It is sometimes convenient to express the state table in a slightly different form having only three sections: present state, next state, and output. The input conditions are enumerated under the next-state and output sections. The state table of Table 5.2 is repeated in Table 5.3 in this second form. For each present state, there are two possible next states and outputs, depending on the value of the input. One form may be preferable to the other, depending on the application.

Table 5.3
Second Form of the State Table

Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
A	**B**	**A**	**B**	**A**	**B**	**y**	**y**
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

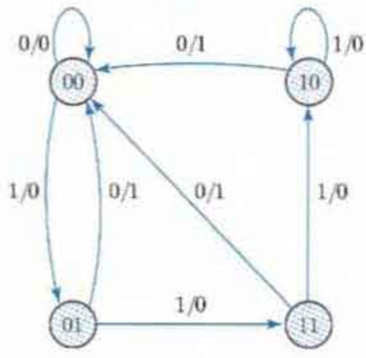


FIGURE 5.16
State diagram of the circuit of Fig. 5.15

State Diagram

The information available in a state table can be represented graphically in the form of a state diagram. In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting the circles. The state diagram of the sequential circuit of Fig. 5.15 is shown in Fig. 5.16. The state diagram provides the same information as the state table and is obtained directly from Table 5.2 or Table 5.3. The binary number inside each circle identifies the state of the flip-flops. The directed lines are labeled with two binary numbers separated by a slash. The input value during the present state is labeled first, and the number after the slash gives the output during the present state with the given input. (It is important to remember that the bit value listed for the output along the directed line occurs during the present state and with the indicated input, and has nothing to do with the transition to the next state.) For example, the directed line from state 00 to 01 is labeled 1/0, meaning that when the sequential circuit is in the present state 00 and the input is 1, the output is 0. After the next clock cycle, the circuit goes to the next state, 01. If the input changes to 0, then the output becomes 1, but if the input remains at 1, the output stays at 0. This information is obtained from the state diagram along the two directed lines emanating from the circle with state 01. A directed line connecting a circle with itself indicates that no change of state occurs.

There is no difference between a state table and a state diagram, except in the manner of representation. The state table is easier to derive from a given logic diagram and the state equation. The state diagram follows directly from the state table. The state diagram gives a pictorial view of state transitions and is the form more suitable for human interpretation of the circuit's operation. For example, the state diagram of Fig. 5.16 clearly shows that, starting from state 00, the output is 0 as long as the input stays at 1. The first 0 input after a string of 1's gives an output of 1 and transfers the circuit back to the initial state, 00. The machine represented by the state diagram acts to detect a zero in the bit stream of data.

Flip-Flop Input Equations

The logic diagram of a sequential circuit consists of flip-flops and gates. The interconnections among the gates form a combinational circuit and may be specified algebraically with Boolean expressions. The knowledge of the type of flip-flops and a list of the Boolean expressions of the combinational circuit provide the information needed to draw the logic diagram of the sequential circuit. The part of the combinational circuit that generates external outputs is described algebraically by a set of Boolean functions called *output equations*. The part of the circuit that generates the inputs to flip-flops is described algebraically by a set of Boolean functions called flip-flop *input equations* (or, sometimes, *excitation equations*). We will adopt the convention of using the flip-flop input symbol to denote the input equation variable and a subscript to designate the name of the flip-flop output. For example, the following input equation specifies an OR gate with inputs x and y connected to the D input of a flip-flop whose output is labeled with the symbol Q:

$$D_Q = x + y$$

The sequential circuit of Fig. 5.15 consists of two D flip-flops A and B, an input x, and an output y. The logic diagram of the circuit can be expressed algebraically with two flip-flop input equations and an output equation:

$$D_A = Ax + Bx$$
$$D_B = A'x$$
$$y = (A + B)x'$$

The three equations provide the necessary information for drawing the logic diagram of the sequential circuit. The symbol D_A specifies a D flip-flop labeled A. D_B specifies a second D flip-flop labeled B. The Boolean expressions associated with these two variables and the expression for output y specify the combinational circuit part of the sequential circuit.

The flip-flop input equations constitute a convenient algebraic form for specifying the logic diagram of a sequential circuit. They imply the type of flip-flop from the letter symbol, and they fully specify the combinational circuit that drives the flip-flops. Note that the expression for the input equation for a D flip-flop is identical to the expression for the corresponding state equation. This is because of the characteristic equation that equates the next state to the value of the D input: $Q(t + 1) = D_Q$.

Analysis with *D* Flip-Flops

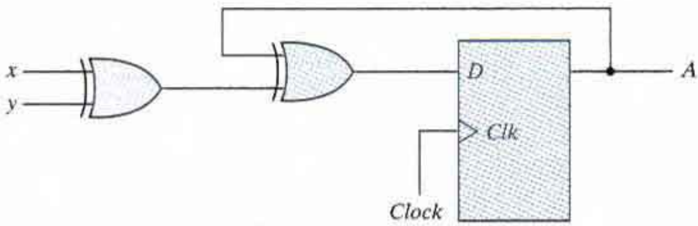
We will summarize the procedure for analyzing a clocked sequential circuit with D flip-flops by means of a simple example. The circuit we want to analyze is described by the input equation

$$D_A = A \oplus x \oplus y$$

The D_A symbol implies a D flip-flop with output A. The x and y variables are the inputs to the circuit. No output equations are given, which implies that the output comes from the output of the flip-flop. The logic diagram is obtained from the input equation and is drawn in Fig. 5.17(a).

The state table has one column for the present state of flip-flop A, two columns for the two inputs, and one column for the next state of A. The binary numbers under Axy are listed from 000 through 111 as shown in Fig. 5.17(b). The next-state values are obtained from the state equation

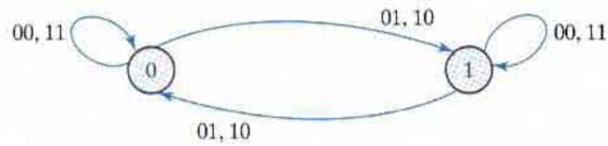
$$A(t + 1) = A \oplus x \oplus y$$



(a) Circuit diagram

Present state	Inputs	Next state
A	x y	A
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

(b) State table



(c) State diagram

FIGURE 5.17
Sequential circuit with *D* flip-flop

The expression specifies an odd function and is equal to 1 when only one variable is 1 or when all three variables are 1. This is indicated in the column for the next state of A.

The circuit has one flip-flop and two states. The state diagram consists of two circles, one for each state as shown in Fig. 5.17(c). The present state and the output can be either 0 or 1, as indicated by the number inside the circles. A slash on the directed lines is not needed, because there is no output from a combinational circuit. The two inputs can have four possible combinations for each state. Two input combinations during each state transition are separated by a comma to simplify the notation.

Analysis with *JK* Flip-Flops

A state table consists of four sections: present state, inputs, next state, and outputs. The first two are obtained by listing all binary combinations. The output section is determined from the output equations. The next-state values are evaluated from the state equations. For a D-type flip-flop, the state equation is the same as the input equation. When a flip-flop other than the D type is used, such as JK or T, it is necessary to refer to the corresponding characteristic table or characteristic equation to obtain the next-state values. We will illustrate the procedure first by using the characteristic table and again by using the characteristic equation.

The next-state values of a sequential circuit that uses *JK*- or *T*-type flip-flops can be derived as follows:

1. Determine the flip-flop input equations in terms of the present state and input variables.
2. List the binary values of each input equation.
3. Use the corresponding flip-flop characteristic table to determine the next-state values in the state table.

As an example, consider the sequential circuit with two *JK* flip-flops *A* and *B* and one input *x*, as shown in Fig. 5.18. The circuit has no outputs; therefore, the state table does not need an output column. (The outputs of the flip-flops may be considered as the outputs in this case.) The circuit can be specified by the flip-flop input equations

$$J_A = B \quad K_A = Bx'$$
$$J_B = x' \quad K_B = A'x + Ax' = A \oplus x$$

The state table of the sequential circuit is shown in Table 5.4. The present-state and input columns list the eight binary combinations. The binary values listed under the columns labeled *flip-flop inputs* are not part of the state table, but they are needed for the purpose of evaluating the next state as specified in step 2 of the procedure. These binary values are obtained directly from the four input equations in a manner similar to that for obtaining a truth table from a Boolean expression. The next state of each flip-flop is evaluated from the corresponding *J* and *K* inputs and the characteristic table of the *JK* flip-flop listed in Table 5.1. There are four cases to consider. When $J = 1$ and $K = 0$, the next state is 1. When $J = 0$ and

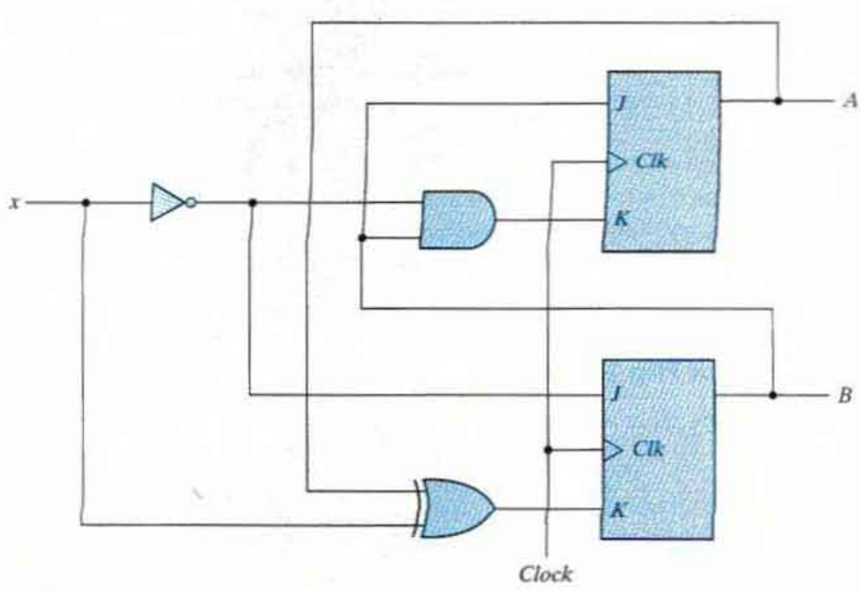


FIGURE 5.18
Sequential circuit with *JK* flip-flop

Table 5.4
State Table for Sequential Circuit with JK Flip-Flops

Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

$K = 1$, the next state is 0. When $J = K = 0$, there is no change of state and the next-state value is the same as that of the present state. When $J = K = 1$, the next-state bit is the complement of the present-state bit. Examples of the last two cases occur in the table when the present state AB is 10 and input x is 0. JA and KA are both equal to 0 and the present state of A is 1. Therefore, the next state of A remains the same and is equal to 1. In the same row of the table, JB and KB are both equal to 1. Since the present state of B is 0, the next state of B is complemented and changes to 1.

The next-state values can also be obtained by evaluating the state equations from the characteristic equation. This is done by using the following procedure:

1. Determine the flip-flop input equations in terms of the present state and input variables.
2. Substitute the input equations into the flip-flop characteristic equation to obtain the state equations.
3. Use the corresponding state equations to determine the next-state values in the state table.

The input equations for the two JK flip-flops of Fig. 5.18 were listed a couple of paragraphs ago. The characteristic equations for the flip-flops are obtained by substituting A or B for the name of the flip-flop, instead of Q:

$$A(t + 1) = JA' + K'A$$
$$B(t + 1) = JB' + K'B$$

Substituting the values of J_A and K_A from the input equations, we obtain the state equation for A:

$$A(t + 1) = BA' + (Bx')'A = A'B + AB' + Ax$$

The state equation provides the bit values for the column headed "Next State" for A in the state table. Similarly, the state equation for flip-flop B can be derived from the characteristic equation by substituting the values of J_B and K_B:

$$B(t + 1) = x'B' + (A \oplus x)'B = B'x' + ABx + A'Bx'$$

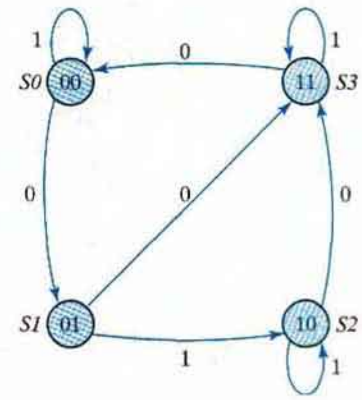


FIGURE 5.19
State diagram of the circuit of Fig. 5.18

The state equation provides the bit values for the column headed "Next State" for B in the state table. Note that the columns in Table 5.4 headed "Flip-Flop Inputs" are not needed when state equations are used.

The state diagram of the sequential circuit is shown in Fig. 5.19. Note that since the circuit has no outputs, the directed lines out of the circles are marked with one binary number only, to designate the value of input x.

Analysis With *T* Flip-Flops

The analysis of a sequential circuit with T flip-flops follows the same procedure outlined for JK flip-flops. The next-state values in the state table can be obtained by using either the characteristic table listed in Table 5.1 or the characteristic equation

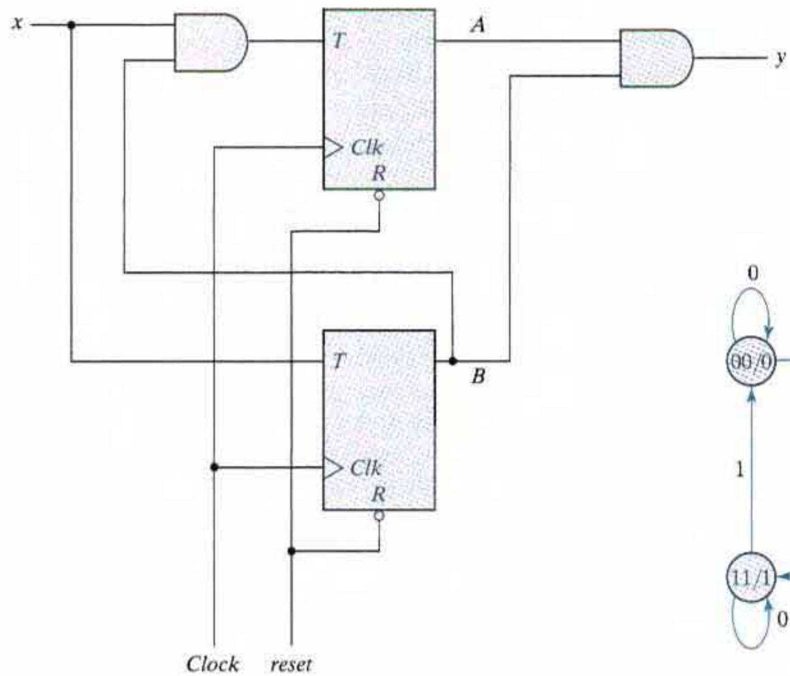
$$Q(t + 1) = T \oplus Q = T'Q + TQ'$$

Now consider the sequential circuit shown in Fig. 5.20. It has two flip-flops A and B, one input x, and one output y and can be described algebraically by two input equations and an output equation:

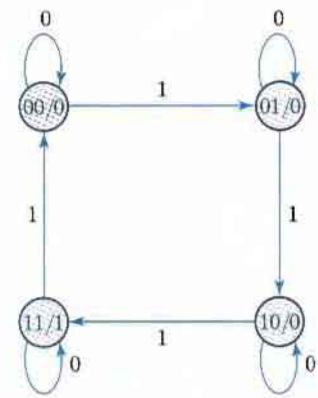
$$T_A = Bx$$
$$T_B = x$$
$$y = AB$$

The state table for the circuit is listed in Table 5.5. The values for y are obtained from the output equation. The values for the next state can be derived from the state equations by substituting T_A and T_B in the characteristic equations, yielding

$$A(t + 1) = (Bx)'A + (Bx)A' = AB' + Ax' + A'Bx$$
$$B(t + 1) = x \oplus B$$



(a) Circuit diagram



(b) State diagram

FIGURE 5.20
Sequential circuit with *T* flip-flops

The next-state values for A and B in the state table are obtained from the expressions of the two state equations.

The state diagram of the circuit is shown in Fig. 5.20(b). As long as input x is equal to 1, the circuit behaves as a binary counter with a sequence of states 00, 01, 10, 11, and back to 00.

Table 5.5
State Table for Sequential Circuit with T Flip-Flops

Present State		Input	Next State		Output
A	***B***	***x***	***A***	***B***	***y***
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

When $x = 0$, the circuit remains in the same state. Output y is equal to 1 when the present state is 11. Here, the output depends on the present state only and is independent of the input. The two values inside each circle and separated by a slash are for the present state and output.

Mealy and Moore Models of Finite State Machines

The most general model of a sequential circuit has inputs, outputs, and internal states. It is customary to distinguish between two models of sequential circuits: the Mealy model and the Moore model. Both are shown in Figure 5.21. They differ only in the way the output is generated. In the Mealy model, the output is a function of both the present state and the input. In the Moore model, the output is a function of only the present state. A circuit may have both types of outputs. The two models of a sequential circuit are commonly referred to as a finite state machine, abbreviated FSM. The Mealy model of a sequential circuit is referred to as a Mealy FSM or Mealy machine. The Moore model is referred to as a Moore FSM or Moore machine.

An example of a Mealy model is given in Fig. 5.15. Output y is a function of both input x and the present state of A and B. The corresponding state diagram in Fig. 5.16 shows both the input and output values, separated by a slash along the directed lines between the states.

An example of a Moore model is given in Fig. 5.18. Here, the output is a function of the present state only. The corresponding state diagram in Fig. 5.19 has only inputs marked along the

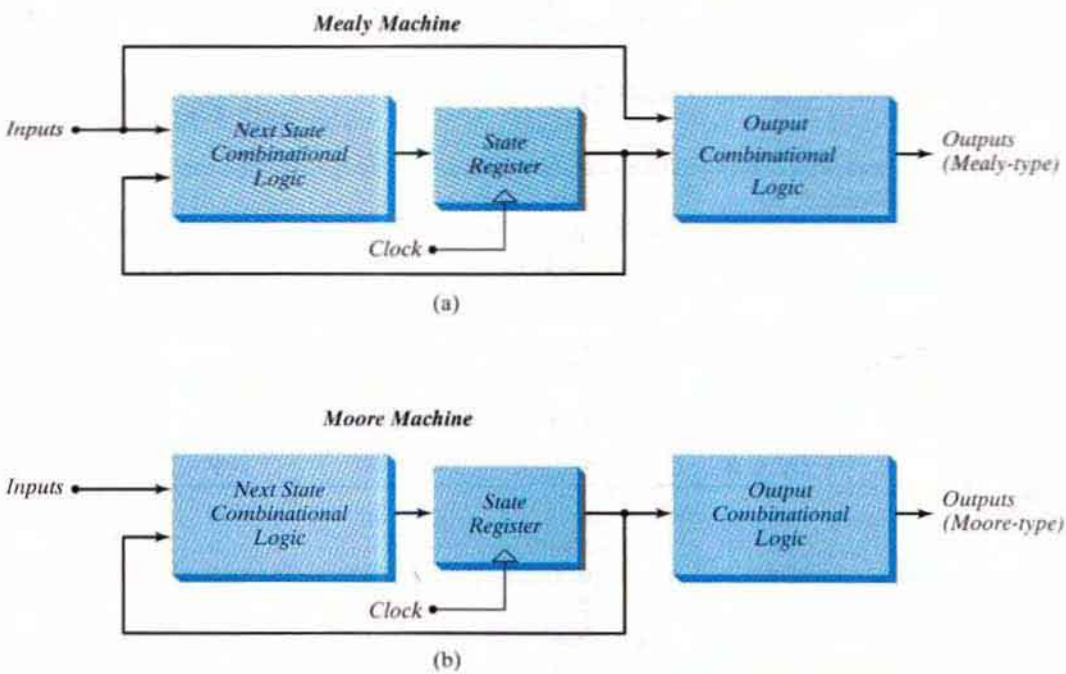


FIGURE 5.21
Block diagrams of Mealy and Moore state machines

directed lines. The outputs are the flip-flop states marked inside the circles. Another example of a Moore model is the sequential circuit of Fig. 5.20. The output depends only on flip-flop values, and that makes it a function of the present state only. The input value in the state diagram is labeled along the directed line, but the output value is indicated inside the circle together with the present state.

In a Moore model, the outputs of the sequential circuit are synchronized with the clock, because they depend only on flip-flop outputs that are synchronized with the clock. In a Mealy model, the outputs may change if the inputs change during the clock cycle. Moreover, the outputs may have momentary false values because of the delay encountered from the time that the inputs change and the time that the flip-flop outputs change. In order to synchronize a Mealy-type circuit, the inputs of the sequential circuit must be synchronized with the clock and the outputs must be sampled immediately before the clock edge. The inputs are changed at the inactive edge of the clock to ensure that the inputs to the flip-flops stabilize before the active edge of the clock occurs. Thus, the output of the Mealy machine is the value that is present immediately before the active edge of the clock.

5.6 SYNTHESIZABLE HDL MODELS OF SEQUENTIAL CIRCUITS

The Verilog hardware description language (HDL) was introduced in Section 3.10. Combinational circuits were described in Section 4.12, and behavioral modeling with Verilog was introduced in that section as well. Behavioral models are abstract representations of the functionality of digital hardware. Designers write behavioral models to quickly describe how a circuit is to operate, without having to first specify its hardware. In this section, we continue the discussion of behavioral modeling and present description and examples of flip-flops and sequential circuits in preparation for modeling more complex circuits.

Behavioral Modeling

There are two kinds of abstract behaviors in the Verilog HDL. Behavior declared by the keyword **initial** is called *single-pass behavior* and specifies a single statement or a block statement (i.e., a list of statements enclosed by either a **begin** ... **end** or a **fork** ... **join** keyword pair). A single-pass behavior expires after the associated statement executes. In practice, designers use single-pass behavior primarily to prescribe stimulus signals in a test bench—never to model the behavior of a circuit—because synthesis tools do not accept descriptions that use the **initial** statement. The **always** keyword declares a *cyclic behavior*. Both types of behaviors begin executing when the simulator launches at time $t = 0$. The **initial** behavior expires after its statement executes; the **always** behavior executes and reexecutes indefinitely, until the simulation is stopped. A module may contain an arbitrary number of **initial** or **always** behavioral statements. They execute concurrently with respect to each other starting at time 0 and may interact through common variables. Here's a word description of how an **always** statement works for a simple model of a D flip-flop: Whenever the rising edge of the clock occurs, if the reset input is asserted, the output q gets 0; otherwise the output Q gets the value of the input D. The execution of statements triggered by the clock is repeated until the simulation ends. We'll see shortly how to write this description in Verilog.

An **initial** behavioral statement executes only once. It begins its execution at the start of simulation and expires after all of its statements have completed execution. As mentioned at the end of Section 4.12, the **initial** statement is useful for generating input signals to simulate a design. In simulating a sequential circuit, it is necessary to generate a clock source for triggering the flip-flops. The following are two possible ways to provide a free-running clock that operates for a specified number of cycles:

```
initial
 begin
  clock = 1'b0;
  repeat (30)
   #10 clock = ~clock;
 end
```

```
initial
 begin
  clock = 1'b0;
 end

initial 300 $finish;
always #10 clock = ~clock;
```

In the first version, the **initial** block contains two statements enclosed within the **begin** and **end** keywords. The first statement sets *clock* to 0 at time = 0. The second statement specifies a loop that reexecutes 30 times to wait 10 time units and then complement the value of *clock*. This produces 15 clock cycles, each with a cycle time of 20 time units. In the second version, the first **initial** behavior has a single statement that sets *clock* to 0 at time = 0, and it then expires (causes no further simulation activity). The second single-pass behavior declares a stopwatch for the simulation. The system task **finish** causes the simulation to terminate unconditionally after 300 time units have elapsed. Because this behavior has only one statement associated with it, there is no need to write the **begin** ... **end** keyword pair. After 10 time units, the **always** statement repeatedly complements *clock*, providing a clock generator having a cycle time of 20 time units. The three behavioral statements in the second example can be written in any order.

Here is another way to describe a free-running clock:

```
initial begin clock = 0; forever #10 clock = ~clock; end
```

This version, with two statements on one line, initializes the clock and then executes an indefinite loop (**forever**) in which the clock is complemented after a delay of 10 time steps. Note that the single-pass behavior never finishes executing and so does not expire. Another behavior would have to terminate the simulation.

The activity associated with either type of behavioral statement can be controlled by a delay operator that waits for a certain time or by an event control operator that waits for certain conditions to become true or for specified events (changes in signals) to occur. Time delays specified with the # *delay control operator* are commonly used in single-pass behaviors. The delay control operator suspends execution of statements until a specified time has elapsed. We've already seen examples of its use to specify signals in a test bench. Another operator, @, is called the *event control operator* and is used to *suspend* activity until an event occurs. An event can be an unconditional change in a signal value (e.g., @ A) or a specified transition of a signal value (e.g., @ (**posedge** clock)). The general form of this type of statement is

```
always @ (event control expression) begin
   // Procedural assignment statements that execute when the condition is met
end
```

The event control expression specifies the condition that must occur to launch execution of the procedural assignment statements. The variables in the left-hand side of the procedural statements must be of the **reg** data type and must be declared as such. The right-hand side can be any expression that produces a value using Verilog-defined operators.

The event control expression (also called the sensitivity list) specifies the events that must occur to initiate execution of the procedural statements associated with the **always** block. Statements within the block execute sequentially from top to bottom. After the last statement executes, the behavior waits for the event control expression to be satisfied. Then the statements are executed again. The sensitivity list can specify level-sensitive events, edge-sensitive events, or a combination of the two. In practice, designers do not make use of the third option, because this third form is not one that synthesis tools are able to translate into physical hardware. Level-sensitive events occur in combinational circuits and in latches. For example, the statement

```
always @ (A or B or C)
```

will initiate execution of the procedural statements in the associated **always** block if a change occurs in *A*, *B*, or *C*. In synchronous sequential circuits, changes in flip-flops occur only in response to a transition of a clock pulse. The transition may be either a positive edge or a negative edge of the clock, but not both. Verilog HDL takes care of these conditions by providing two keywords: **posedge** and **negedge**. For example, the expression

```
always @(posedge clock or negedge reset)      // Verilog 1995
```

will initiate execution of the associated procedural statements only if the clock goes through a positive transition or if *reset* goes through a negative transition. The 2001 and 2005 revisions to the Verilog language allow a comma-separated list for the event control expression (or sensitivity list):

```
always @(posedge clock, negedge reset)      // Verilog 2001, 2005
```

A procedural assignment is an assignment of a logic value to a variable within an **initial** or **always** statement. This is in contrast to a continuous assignment discussed in Section 4.12 with dataflow modeling. A continuous assignment has an implicit level-sensitive sensitivity list consisting of all of the variables on the right-hand side of its assignment statement. The updating of a continuous assignment is triggered whenever an event occurs in a variable listed on the right-hand side of its expression. In contrast, a procedural assignment is made only when an assignment statement is executed within a behavioral statement. For example, the clock signal in the preceding example was complemented only when the statement $clock = \sim clock$ executed; the statement did not execute until 10 time units after the simulation began. It is important to remember that a variable having type **reg** remains unchanged until a procedural assignment is made to give it a new value.

There are two kinds of procedural assignments: *blocking* and *nonblocking*. The two are distinguished by the symbols that they use. Blocking assignments use the symbol (=) as the assignment operator, and nonblocking assignments use (<=) as the operator. Blocking assignment statements are executed sequentially in the order they are listed in a block of statements. Nonblocking assignments are executed concurrently by evaluating the set of expressions on the right-hand side of the list of statements; they do not make assignments to their left-hand sides until all of the expressions are evaluated. The two types of assignments may be better understood by means of an illustration. Consider these two procedural blocking assignments:

```
B = A
C = B + 1
```

The first statement transfers A into B. The second statement increments the value of B and transfers the new value to C. At the completion of the assignments, C contains the value of $A + 1$.

Now consider the two statements as nonblocking assignments:

```
B <= A
C <= B + 1
```

When the statements are executed, the expressions on the right-hand side are evaluated and stored in a temporary location. The value of A is kept in one storage location and the value of $B + 1$ in another. After all the expressions in the block are evaluated and stored, the assignment to the targets on the left-hand side is made. In this case, C will contain the original value of B, plus 1. A general rule is to use blocking assignments when sequential ordering is imperative and in cyclic behavior that is level sensitive (i.e., in combinational logic). Use nonblocking assignments when modeling concurrent execution (e.g., edge-sensitive behavior such as synchronous, concurrent register transfers) and when modeling latched behavior. Nonblocking assignments are imperative in dealing with register transfer level design, as shown in Chapter 8. They model the concurrent operations of physical hardware synchronized by a common clock. Today's designers are expected to know what features of an HDL are useful in a practical way and how to avoid features that are not. Following these rules will prevent conditions that lead synthesis tools astray and create mismatches between the behavior of a model and the behavior of physical hardware that is produced by a synthesis tool.

Flip-Flops and Latches

HDL Examples 5.1 through 5.4 show descriptions of various flip-flops and a D latch. The D latch is transparent and responds to a change in data input with a change in output, as long as the enable input is asserted. The module description of a D latch is shown in HDL Example 5.1. It has two inputs, D and *enable*, and one output Q. Since Q is evaluated in a procedural statement, it must be declared as **reg** type. Latches respond to input signal levels, so the two inputs are listed without edge qualifiers in the event enable expression following the @ symbol in the **always** statement. There is only one blocking procedural assignment statement, and it specifies the transfer of input D to output Q if *enable* is true (logic 1). Note that this statement is executed every time there is a change in D if *enable* is 1.

A D-type flip-flop is the simplest example of a sequential machine. HDL Example 5.2 describes two positive-edge D flip-flops in two modules. The first responds only to the clock; the second includes an asynchronous reset input. Output Q must be declared as a **reg** data type in addition to being listed as an output. This is because it is a target output in a procedural assignment statement. The keyword **posedge** ensures that the transfer of input D into Q is synchronized by the positive-edge transition of *Clk*. A change in D at any other time does not change Q.

HDL Example 5.1

```
// Description of D latch (See Fig. 5.6)
module D_latch (Q, D, enable);
 output Q;
 input  D, enable;
 reg    Q;
 always @ (enable or D)
  if (enable) Q <= D;              // Same as: if (enable == 1)
 endmodule

// Alternative syntax (Verilog 2001, 2005)
module D_latch (output reg Q, input enable, D);
 always @ (enable, D)
  if (enable) Q <= D;              // No action if enable not asserted
endmodule
```

HDL Example 5.2

```
// D flip-flop without reset
module D_FF (Q, D, Clk);
 output Q;
 input  D, Clk;
 reg    Q;
 always @ (posedge Clk)
   Q <= D;
endmodule

// D flip-flop with asynchronous reset (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
 always @ (posedge Clk, negedge rst)
 if (~rst) Q <= 1'b0;   // Same as: if (rst == 0)
 else Q <= D;
endmodule
```

The second module includes an asynchronous reset input in addition to the synchronous clock. A specific form of an **if** statement is used to describe such a flip-flop so that the model can be synthesized by a software tool. The event expression after the @ symbol in the **always** statement may have any number of edge events, either **posedge** or **negedge**. For modeling hardware, one of the events must be a clock event. The remaining events specify conditions under which asynchronous logic is to be executed. The designer knows which signal is the clock, but *clock* is not an identifier that software tools automatically recognize as the synchronizing signal of a circuit. The tool must be able to infer which signal is the clock, so you need to write the description in a way that enables the tool to infer the clock correctly. The rules are simple to follow: (1) Each **if** or **else if** statement in the procedural assignment statements is to correspond to an asynchronous event, (2) the last **else** statement corresponds to the clock event, and (3) the asynchronous events are tested first. There are two edge events in the second module of HDL Example 5.2. The **negedge** rst (reset) event is asynchronous, since it matches the **if** (~rst) statement. As long as *rst* is 0, *Q* is cleared to 0. If *Clk* has a positive transition, its effect is blocked. Only if $rst = 1$ can the **posedge** clock event synchronously transfer *D* into *Q*.

Hardware always has a reset signal. It is strongly recommended that all models of edge-sensitive behavior include a reset (or preset) input signal; otherwise, the initial state of the flip-flops of the sequential circuit cannot be determined. A sequential circuit cannot be tested with HDL simulation unless an initial state can be assigned with an input signal.

HDL Example 5.3 describes the construction of a *T* or *JK* flip-flop from a *D* flip-flop and gates. The circuit is described with the characteristic equations of the flip-flops:

$$Q(t+1) = Q \oplus T \qquad \text{for a } T \text{ flip-flop}$$
$$Q(t+1) = JQ' + K'Q \qquad \text{for a } JK \text{ flip-flop}$$

The first module, *TFF*, describes a *T* flip-flop by instantiating *DFF*. (Instantiation is explained in Section 4.12.) The declared **wire**, *DT*, is assigned the exclusive-OR of *Q* and *T*, as is required for building a *T* flip-flop with a *D* flip-flop. The instantiation with the value of *DT* replacing *D* in module *DFF* produces the required *T* flip-flop. The *JK* flip-flop is specified in a similar manner by using its characteristic equation to define a replacement for *D* in the instantiated *DFF*.

HDL Example 5.3

```
// T flip-flop from D flip-flop and gates
module TFF (Q, T, Clk, rst);
 output Q;
 input  T, Clk, rst;
 wire   DT;
 assign DT = Q ^ T ;                    // Continuous assignment
// Instantiate the D flip-flop
 DFF TF1 (Q, DT, Clk, rst);
endmodule

// JK flip-flop from D flip-flop and gates (V2001, 2005)
module JKFF (output reg Q, input J, K, Clk, rst);
 wire JK;
 assign JK = (J & ~Q) | (~K & Q);
// Instantiate D flip-flop
 DFF JK1 (Q, J, K, Clk, rst);
endmodule

// D flip-flop (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
 always @ (posedge Clk, negedge rst)
  if (~rst) Q <= 1'b0;
  else Q <= D;
endmodule
```

HDL Example 5.4 shows another way to describe a *JK* flip-flop. Here, we choose to describe the flip-flop by using the characteristic table rather than the characteristic equation. The **case** multiway branch condition checks the two-bit number obtained by concatenating the bits of *J* and *K*. The **case** expression $(\{J, K\})$ is evaluated and compared with the values in the list of statements that follows. The first value that matches the true condition is executed. Since the concatenation of *J* and *K* produces a two-bit number, it can be equal to 00, 01, 10, or 11. The first bit gives the value of *J* and the second the value of *K*. The four possible conditions specify the value of the next state of *Q* after the application of a positive-edge clock.

HDL Example 5.4

```
// Functional description of JK flip-flop (V2001, 2005)
module  JK_FF (input J, K, Clk, output reg Q, output Q_b);
 assign Q_b = ~ Q ;
 always @ (posedge Clk)
  case ({J,K})
   2'b00: Q <= Q;
   2'b01: Q <= 1'b0;
   2'b10: Q <= 1'b1;
   2'b11: Q <= ~Q;

endmodule
```

State Diagram

An HDL model of the operation of a sequential circuit can be based on the format of the circuit's state diagram. A Mealy HDL model is presented in HDL Example 5.5 for the state machine described by the state diagram shown in Figure 5.16. The input, output, clock, and reset are declared in the usual manner. The state of the flip-flops is declared with identifiers *state* and *next_state*. These variables hold the values of the present state and the next value of the sequential circuit. The state's binary assignment is done with a **parameter** statement. (Verilog allows constants to be defined in a module by the keyword **parameter**.) The four states *S0* through *S3* are assigned binary 00 through 11. The notation $S2 = 2'\text{b}10$ is preferable to the alternative $S2 = 2$. The former uses only two bits to store the constant, whereas the latter results in a binary number with 32 (or 64) bits.

HDL Example 5.5

```
// Mealy FSM zero detector (See Fig. 5.16)                Verilog 2001, 2005 syntax
module Mealy_Zero_Detector (
 output reg y_out,
 input       x_in, clock, reset
);
 reg [1: 0]          state, next_state;
 parameter           S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
```

```
 always @ (posedge clock, negedge reset)      Verilog 2001, 2005 syntax
  if (reset == 0) state <= S0;
  else state <= next_state;
 always @ (state, x_in)                       // Form the next state
  case (state)
   S0:     if (x_in)  next_state = S1; else next_state = S0;
   S1:     if (x_in)  next_state = S3; else next_state = S0;
   S2:     if (~x_in)  next_state = S0; else next_state = S2;
   S3:     if (x_in)  next_state = S2; else next_state = S0;
  endcase
 always @ (state, x_in)                       // Form the output
  case (state)
   S0:         y_out = 0;
   S1, S2, S3: y_out = ~x_in;
  endcase
endmodule
module t_Mealy_Zero_Detector;
 wire   t_y_out;
 reg    t_x_in, t_clock, t_reset;
Mealy_Zero_Detector M0 (t_y_out, t_x_in, t_clock, t_reset);
initial #200 $finish;
initial begin t_clock = 0; forever #5 t_clock = ~t_clock; end
initial fork
    t_reset = 0;
 #2 t_reset = 1;
 #87 t_reset = 0;
 #89 t_reset = 1;
 #10 t_x_in = 1;
 #30 t_x_in = 0;
 #40 t_x_in = 1;
 #50 t_x_in = 0;
 #52 t_x_in = 1;
 #54 t_x_in = 0;
 #70 t_x_in = 1;
 #80 t_x_in = 1;
 #70 t_x_in = 0;
 #90 t_x_in = 1;
 #100 t_x_in = 0;
 #120 t_x_in = 1;
 #160 t_x_in = 0;
 #170 t_x_in = 1;
 join
endmodule
```

The Verilog model in HDL Example 5.5 uses three **always** blocks that execute concurrently and interact through common variables. The first **always** statement resets the circuit to the initial state $S0 = 00$ and specifies the synchronous clocked operation. The statement *state* <= *next_state* is executed only in response to a positive-edge transition of the clock. This means that any change in the value of *next_state* in the second **always** block can affect the value of *state* only as a result of a **posedge** event of *clock*. The second **always** block determines the value of the next state transition as a function of the present state and input. The value assigned to *state* by the nonblocking assignment is the value of *next_state* immediately before the rising edge of *clock*. Notice how the multiway branch condition implements the state transitions specified by the annotated edges in the state diagram of Fig. 5.16. The third **always** block specifies the output as a function of the present state and the input. Although this block is listed as a separate behavior for clarity, it could be combined with the second block. Note that the value of output *y_out* may change if the value of input *x_in* changes while the circuit is in any given state.

So let's summarize how the model describes the behavior of the machine: At every rising edge of *clock*, if *reset* is not asserted, the state of the machine is updated by the first **always** block; when *state* is updated by the first **always** block, the change in *state* is detected by the sensitivity list mechanism of the second **always** block; then the second **always** block updates the value of *next_state* (it will be used by the first **always** block at the next tick of the clock); the third **always** block also detects the change in *state* and updates the value of the output. In addition, the second and third **always** blocks detect changes in *x_in* and update *next_state* and *y_out* accordingly. The test bench provided with *Mealy_Zero_Detector* provides some waveforms to stimulate the model, producing the results shown in Fig. 5.22. Notice how *t_y_out*

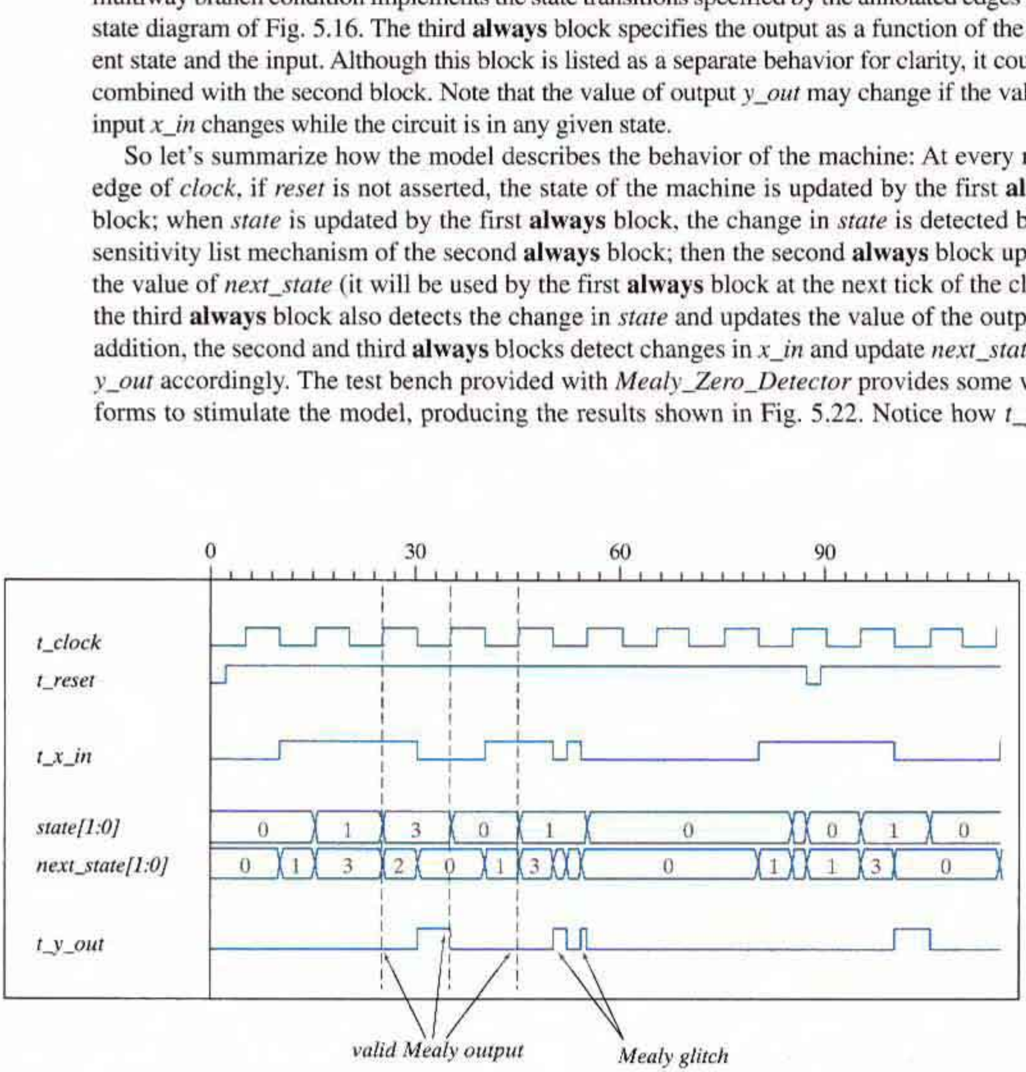


FIGURE 5.22
Simulation output of *Mealy_Zero_Detector*

responds to changes in both the state and the input and has a glitch (a transient logic value). The waveform description uses the **fork** ... **join** construct. Statements within the **fork** ... **join** block execute in parallel, so the time delays are relative to a common reference of $t = 0$. It is usually more convenient to use the **fork** ... **join** block instead of the **begin** ... **end** block in describing waveforms. The waveform of *reset* is triggered "on the fly" to demonstrate that the machine recovers from an unexpected reset condition during any state.

How does our Verilog model *Mealy_Zero_Detector* correspond to hardware? The first **always** block corresponds to a *D* flip-flop implementation of the state register in Fig. 5.21; the second **always** block is the combinational logic block describing the next state; the third **always** block describes the output combinational logic of the zero-detecting Mealy machine. The register operation of the state transition uses the nonblocking assignment operator (<=) because the (edge-sensitive) flip-flops of a sequential machine are updated concurrently by a common clock. The second and third **always** blocks describe combinational logic, which is level sensitive, so they use the blocking (=) assignment operator. Their sensitivity lists include both the state and the input because their logic must respond to a change in either or both of them.

Note: the modeling style illustrated by *Mealy_Zero_Detector* is commonly used by designers. Notice that the reset signal is associated with the first **always** block. It is modeled here as an active-low reset. By including the reset in the model of the state transition, there is no need to include it in the combinational logic that specifies the next state and the output, producing a simpler and more readable description.

The behavior of the Moore FSM having the state diagram shown in Fig. 5.19 can be modeled by the Verilog description in HDL Example 5.6. This example shows that it is possible to describe the state transitions of a clocked sequential machine with only one **always** block. The present state of the circuit is identified by the variable *state*. The state transitions are triggered by the rising edge of the clock according to the conditions listed in the **case** statements. The combinational logic that implicitly determines the next state is included in the nonblocking assignment to *state*. In this example, the output of the circuit is independent of the input and is taken directly from the outputs of the flip-flops. The two-bit output *y_out* is specified with a continuous (**assign**) statement and is equal to the value of the present state vector. Figure 5.23 shows some simulation results for *Moore_Model_Fig_5_19*. Notice that the output of the Moore machine does not have glitches.

HDL Example 5.6

```
// Moore model FSM (see Fig. 5.19)              Verilog 2001, 2005 syntax
module Moore_Model_Fig_5_19 (
  output [1: 0]          y_out,
  input                  x_in, clock, reset
);
  reg [1: 0]             state;
  parameter              S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @ (posedge clock, negedge reset)
   if (reset == 0) state <= S0;                      // Initialize to state S0
   else case (state)
```

```
 S0:  if (~x_in)  state <= S1; else state <= S0;
 S1:  if (x_in)   state <= S2; else state <= S3;
 S2:  if (~x_in)  state <= S3; else state <= S2;
 S3:  if (~x_in)  state <= S0; else state <= S3;
endcase

assign y_out = state;    // Output of flip-flops

endmodule
```

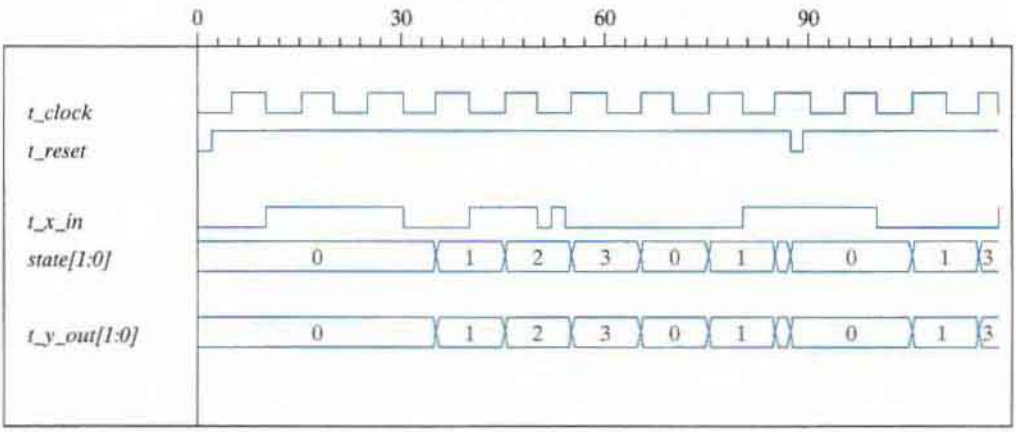


FIGURE 5.23
Simulation output of HDL Example 5.6

Structural Description of Clocked Sequential Circuits

Combinational logic circuits can be described in Verilog by a connection of gates (primitives and UDPs), by dataflow statements (continuous assignments), or by level-sensitive cyclic behaviors (**always** blocks). Sequential circuits are composed of combinational logic and flip-flops, and their HDL models use sequential UDPs and behavioral statements (edge-sensitive cyclic behaviors) to describe the operation of flip-flops. One way to describe a sequential circuit uses a combination of dataflow and behavioral statements. The flip-flops are described with an **always** statement. The combinational part can be described with **assign** statements and Boolean equations. The separate modules can be combined to form a structural model by instantiation within a **module**.

The structural description of a sequential circuit is shown in HDL Example 5.7. We want to encourage the reader to consider alternative ways to model a circuit, so as a point of comparison, we first present *Moore_Model_Fig_5_20*, a Verilog behavioral description of the machine having the state diagram shown in Fig. 5.20. This style of modeling is direct.

An alternative style, used in *Moore_Model_STR_Fig_5_20*, is to represent the structure shown in Fig. 5.20(a). This style uses two modules. The first describes the circuit of Fig. 5.20(a). The second describes the *T* flip-flop that will be used by the circuit. We also show two ways to model the T flip-flop. The first asserts that, at every clock tick, the value of the output of the flip-flop toggles if the toggle input is asserted. The second model describes the behavior of the toggle flip-flop in terms of its characteristic equation. The first style is attractive because it does not require the reader to remember the characteristic equation. Nonetheless, the models are interchangeable and will synthesize to the same hardware circuit. A test bench module provides a stimulus for verifying the functionality of the circuit. The sequential circuit is a two-bit binary counter controlled by input *x_in*. The output, *y_out*, is enabled when the count reaches binary 11. Flip-flops *A* and *B* are included as outputs in order to check their operation. The flip-flop input equations and the output equation are evaluated with continuous assignment (**assign**) statements having the corresponding Boolean expressions. The instantiated *T* flip-flops use *TA* and *TB* as defined by the input equations.

The second module describes the *T* flip-flop. The *reset* input resets the flip-flop to 0 with an active-low signal. The operation of the flip-flop is specified by its characteristic equation, $Q(t + 1) = Q \oplus T$.

The test bench includes both models of the machine. The stimulus module provides common inputs to the circuits to simultaneously display their output responses. The first **initial** block provides eight clock cycles with a period of 10 ns. The second **initial** block specifies a toggling of input *x_in* that occurs at the negative edge transition of the clock. The result of the simulation is shown in Fig. 5.24. The pair (*A*, *B*) goes through the binary sequence 00, 01, 10, 11, and back to 00. The change in the count is triggered by a positive edge of the clock, provided that *x_in* = 1. If *x_in* = 0, the count does not change. *y_out* is equal to 1 when both *A* and *B* are equal to 1. This verifies the main functionality of the circuit, but not a recovery from an unexpected reset event.

HDL Example 5.7

```
// State-diagram-based model (V2001, 2005)
module Moore_Model_Fig_5_20 (
 output y_out,
 input   x_in, clock, reset
);
 reg [1: 0]          state;
 parameter           S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
 always @ (posedge clock, negedge reset)
  if (reset == 0) state <= S0;                          // Initialize to state S0
  else case (state)
   S0:   if (x_in)  state <= S1; else state <= S0;
   S1:   if (x_in)  state <= S2; else state <= S1;
```

```
    S2:   if (x_in)  state <= S3; else state <= S2;
    S3:   if (x_in)  state <= S0; else state <= S3;
  endcase

 assign y_out = (state == S3);              // Output of flip-flops
endmodule

// structural model
module Moore_Model_STR_Fig_5_20 (
 output   y_out, A, B,
 input    x_in, clock, reset
);
 wire     TA, TB;

// Flip-flop input equations
 assign TA = x_in & B;
 assign TB = x_in;
// Output equation
 assign y_out = A & B;

// Instantiate Toggle flip-flops
 Toggle_flip_flop_3 M_A (A, TA, clock, reset);
 Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule

module Toggle_flip_flop (Q, T, CLK, RST_b);
 output   Q;
 input    T, CLK, RST_b;
 reg      Q;

 always @ (posedge CLK, negedge RST_b)
  if (RST_b == 0) Q <= 1'b0;
  else if (T) Q <= ~Q;
endmodule

// Alternative model using characteristic equation
// module Toggle_flip_flop (Q, T, CLK, RST_b);
// output  Q;
// input   T, CLK, RST_b;
// reg     Q;

// always @ (posedge CLK, negedge RST)
//   if (RST_b == 0) Q <= 1'b0;
//   else Q <= Q ^ T;
// endmodule
```

```
module t_Moore_Fig_5_20;
 wire              t_y_out_2, t_y_out_1;
 reg               t_x_in, t_clock, t_reset;

 Moore_Model_Fig_5_20          M1(t_y_out_1, t_x_in, t_clock, t_reset);
 Moore_Model_STR_Fig_5_20      M2 (t_y_out_2, A, B, t_x_in, t_clock, t_reset);

 initial #200 $finish;
 initial begin
    t_reset = 0;
    t_clock = 0;
    #5 t_reset = 1;
  repeat (16)
   #5 t_clock = ~t_clock;
 end
 initial begin
      t_x_in = 0;
  #15 t_x_in = 1;
  repeat (8)
   #10 t_x_in = ~t_x_in;
 end
endmodule
```

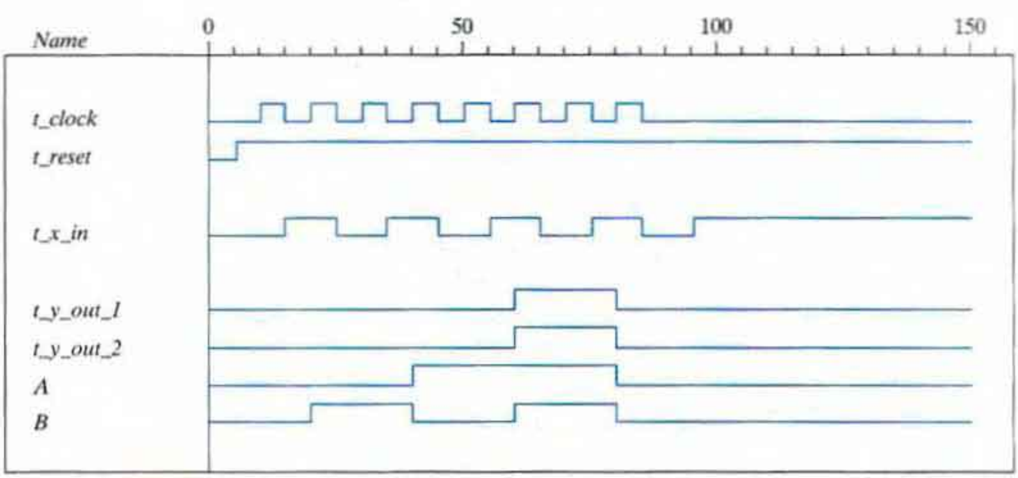


FIGURE 5.24
Simulation output of HDL Example 5.7

5.7 STATE REDUCTION AND ASSIGNMENT

The *analysis* of sequential circuits starts from a circuit diagram and culminates in a state table or diagram. The *design* (synthesis) of a sequential circuit starts from a set of specifications and culminates in a logic diagram. Design procedures are presented in Section 5.8. Two sequential circuits may exhibit the same input–output behavior, but have a different number of internal states in their state diagram. The current section discusses certain properties of sequential circuits that may simplify a design by reducing the number of gates and flip-flops it uses. In general, reducing the number of flip-flops reduces the cost of a circuit.

State Reduction

The reduction in the number of flip-flops in a sequential circuit is referred to as the *state-reduction* problem. State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the external input–output requirements unchanged. Since m flip-flops produce 2^m states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops. An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates.

We will illustrate the state-reduction procedure with an example. We start with a sequential circuit whose specification is given in the state diagram of Fig. 5.25. In our example, only the input–output sequences are important; the internal states are used merely to provide the required sequences. For that reason, the states marked inside the circles are denoted by letter symbols instead of their binary values. This is in contrast to a binary counter, wherein the binary value sequence of the states themselves is taken as the outputs.

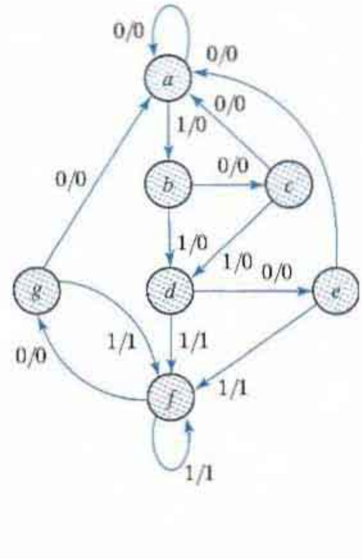


FIGURE 5.25
State diagram

There are an infinite number of input sequences that may be applied to the circuit; each results in a unique output sequence. As an example, consider the input sequence 01010110100 starting from the initial state *a*. Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state. From the state diagram, we obtain the output and state sequence for the given input sequence as follows: With the circuit in initial state *a*, an input of 0 produces an output of 0 and the circuit remains in state *a*. With present state *a* and an input of 1, the output is 0 and the next state is *b*. With present state *b* and an input of 0, the output is 0 and the next state is *c*. Continuing this process, we find the complete sequence to be as follows:

state	*a*	*a*	*b*	*c*	*d*	*e*	*f*	*f*	*g*	*f*	*g*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

In each column, we have the present state, input value, and output value. The next state is written on top of the next column. It is important to realize that in this circuit the states themselves are of secondary importance, because we are interested only in output sequences caused by input sequences.

Now let us assume that we have found a sequential circuit whose state diagram has fewer than seven states, and suppose we wish to compare this circuit with the circuit whose state diagram is given by Fig. 5.25. If identical input sequences are applied to the two circuits and identical outputs occur for all input sequences, then the two circuits are said to be equivalent (as far as the input–output is concerned) and one may be replaced by the other. The problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input–output relationships.

We now proceed to reduce the number of states for this example. First, we need the state table; it is more convenient to apply procedures for state reduction with the use of a table rather than a diagram. The state table of the circuit is listed in Table 5.6 and is obtained directly from the state diagram.

The following algorithm for the state reduction of a completely specified state table is given here without proof: "Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an

Table 5.6
State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

Table 5.7
Reducing the State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*e*	*f*	0	1

equivalent state." When two states are equivalent, one of them can be removed without altering the input–output relationships.

Now apply this algorithm to Table 5.6. Going through the state table, we look for two present states that go to the same next state and have the same output for both input combinations. States *g* and *e* are two such states: They both go to states *a* and *f* and have outputs of 0 and 1 for $x = 0$ and $x = 1$, respectively. Therefore, states *g* and *e* are equivalent, and one of these states can be removed. The procedure of removing a state and replacing it by its equivalent is demonstrated in Table 5.7. The row with present state *g* is removed, and state *g* is replaced by state *e* each time it occurs in the columns headed "Next State."

Present state *f* now has next states *e* and *f* and outputs 0 and 1 for $x = 0$ and $x = 1$, respectively. The same next states and outputs appear in the row with present state *d*. Therefore, states *f* and *d* are equivalent, and state *f* can be removed and replaced by *d*. The final reduced table is shown in Table 5.8. The state diagram for the reduced table consists of only five states and is shown in Fig. 5.26. This state diagram satisfies the original input–output specifications and will produce the required output sequence for any given input sequence. The following list derived from the state diagram of Fig. 5.26 is for the input sequence used previously (note that the same output sequence results, although the state sequence is different):

state	***a***	***a***	***b***	***c***	***d***	***e***	***d***	***d***	***e***	***d***	***e***	***a***
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

Table 5.8
Reduced State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*d*	0	1
e	*a*	*d*	0	1

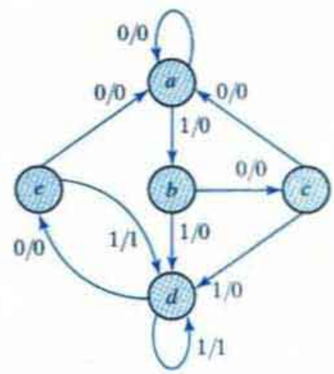


FIGURE 5.26
Reduced state diagram

In fact, this sequence is exactly the same as that obtained for Fig. 5.25 if we replace g by e and f by d.

Checking each pair of states for equivalency can be done systematically by means of a procedure that employs an implication table, which consists of squares, one for every suspected pair of possible equivalent states. By judicious use of the table, it is possible to determine all pairs of equivalent states in a state table. The use of the implication table for reducing the number of states in a state table is demonstrated in Section 9.5.

The sequential circuit of this example was reduced from seven to five states. In general, reducing the number of states in a state table may result in a circuit with less equipment. However, the fact that a state table has been reduced to fewer states does not guarantee a saving in the number of flip-flops or the number of gates.

State Assignment

In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states. For a circuit with m states, the codes must contain n bits, where $2^n \geq m$. For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111. If the state table of Table 5.6 is used, we must assign binary values to seven states; the remaining state is unused. If the state table of Table 5.8 is used, only five states need binary assignment, and we are left with three unused states. Unused states are treated as don't-care conditions during the design. Since don't-care conditions usually help in obtaining a simpler circuit, it is more likely that the circuit with five states will require fewer combinational gates than the one with seven states.

The simplest way to code five states is to use the first five integers in binary counting order, as shown in the first assignment of Table 5.9. Another similar assignment is the Gray code shown in assignment 2. Here, only one bit in the code group changes when going from one number to the next. This code makes it easier for the Boolean functions to be placed in the map for simplification. Another possible assignment often used in the design of state machines to control data-path units is the one-hot assignment. This configuration uses as many bits as there are

Table 5.9
Three Possible Binary State Assignments

State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

Table 5.10
Reduced State Table with Binary Assignment 1

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

states in the circuit. At any given time, only one bit is equal to 1 while all others are kept at 0. This type of assignment uses one flip-flop per state, which is not an issue for register-rich field-programmable gate arrays. (See Chapter 7.) One-hot encoding usually leads to simpler decoding logic for the next state and output. One-hot machines can be faster than machines with sequential binary encoding, and the silicon area required by the extra flip-flops can be offset by the area saved by using simpler decoding logic. This trade-off is not guaranteed, so it must be evaluated for a given design.

Table 5.10 is the reduced state table with binary assignment 1 substituted for the letter symbols of the states. A different assignment will result in a state table with different binary values for the states. The binary form of the state table is used to derive the next-state and output-forming combinational logic part of the sequential circuit. The complexity of the combinational circuit depends on the binary state assignment chosen.

Sometimes, the name *transition table* is used for a state table with a binary assignment. This convention distinguishes it from a state table with symbolic names for the states. In this book, we use the same name for both types of state tables.

5.8 DESIGN PROCEDURE

Design procedures or methodologies specify hardware that will implement a desired behavior. The design effort for small circuits may be manual, but industry relies on automated synthesis tools for designing massive integrated circuits. The building block used by synthesis tools is the D flip-flop. Together with additional logic, it can implement the behavior of JK and T flip-flops. In fact, designers generally do not concern themselves with the type of flip-flop; rather, their focus is on correctly describing the sequential functionality that is to be implemented by the synthesis tool. Here we will illustrate manual methods using D, JK, and T flip-flops.

The design of a clocked sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which the logic diagram can be obtained. In contrast to a combinational circuit, which is fully specified by a truth table, a sequential circuit requires a state table for its specification. The first step in the design of sequential circuits is to obtain a state table or an equivalent representation, such as a state diagram.

A synchronous sequential circuit is made up of flip-flops and combinational gates. The design of the circuit consists of choosing the flip-flops and then finding a combinational gate structure that, together with the flip-flops, produces a circuit which fulfills the stated specifications. The number of flip-flops is determined from the number of states needed in the circuit. The combinational circuit is derived from the state table by evaluating the flip-flop input equations and output equations. In fact, once the type and number of flip-flops are determined, the design process involves a transformation from a sequential circuit problem into a combinational circuit problem. In this way, the techniques of combinational circuit design can be applied.

The procedure for designing synchronous sequential circuits can be summarized by a list of recommended steps:

1. From the word description and specifications of the desired operation, derive a state diagram for the circuit.
2. Reduce the number of states if necessary.
3. Assign binary values to the states.
4. Obtain the binary-coded state table.
5. Choose the type of flip-flops to be used.
6. Derive the simplified flip-flop input equations and output equations.
7. Draw the logic diagram.

The word specification of the circuit behavior usually assumes that the reader is familiar with digital logic terminology. It is necessary that the designer use intuition and experience to arrive at the correct interpretation of the circuit specifications, because word descriptions may be incomplete and inexact. Once such a specification has been set down and the state diagram obtained, it is possible to use known synthesis procedures to complete the design. Although there are formal procedures for state reduction and assignment (steps 2 and 3), they are seldom used by experienced designers. Steps 4 through 7 in the design can be implemented by exact algorithms and therefore can be automated. The part of the design that follows a well-defined procedure is referred to as *synthesis*. Designers using logic synthesis tools (software) can follow a simplified process that develops an HDL description directly from a state diagram, letting the synthesis tool determine the circuit elements and structure that implement the description.

The first step is a critical part of the process, because succeeding steps depend on it. We will give one simple example to demonstrate how a state diagram is obtained from a word specification.

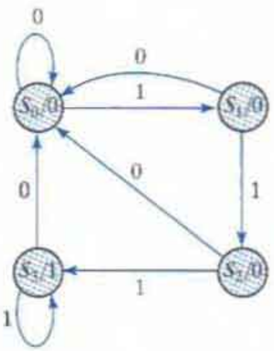


FIGURE 5.27
State diagram for sequence detector

Suppose we wish to design a circuit that detects a sequence of three or more consecutive 1's in a string of bits coming through an input line (i.e., the input is a *serial bit stream*). The state diagram for this type of circuit is shown in Fig. 5.27. It is derived by starting with state S_0, the reset state. If the input is 0, the circuit stays in S_0, but if the input is 1, it goes to state S_1 to indicate that a 1 was detected. If the next input is 1, the change is to state S_2 to indicate the arrival of two consecutive 1's, but if the input is 0, the state goes back to S_0. The third consecutive 1 sends the circuit to state S_3. If more 1's are detected, the circuit stays in S_3. Any 0 input sends the circuit back to S_0. In this way, the circuit stays in S_3 as long as there are three or more consecutive 1's received. This is a Moore model sequential circuit, since the output is 1 when the circuit is in state S_3 and is 0 otherwise.

Synthesis Using *D* Flip-Flops

Once the state diagram has been derived, the rest of the design follows a straightforward synthesis procedure. In fact, we can design the circuit by using an HDL description of the state diagram and the proper HDL synthesis tools to obtain a synthesized netlist. (The HDL description of the state diagram will be similar to HDL Example 5.6 in Section 5.6.) To design the circuit by hand, we need to assign binary codes to the states and list the state table. This is done in Table 5.11. The table is derived from the state diagram of Fig. 5.27 with a sequential binary assignment. We choose two *D* flip-flops to represent the four states, and we label their outputs *A* and *B*. There is one input x and one output y. The characteristic equation of the *D* flip-flop is $Q(t + 1) = D_Q$, which means that the next-state values in the state table specify the *D* input condition for the flip-flop. The flip-flop input equations can be obtained directly from the next-state columns of *A* and *B* and expressed in sum-of-minterms form as

$$A(t + 1) = D_A(A, B, x) = \Sigma(3, 5, 7)$$
$$B(t + 1) = D_B(A, B, x) = \Sigma(1, 5, 7)$$
$$y(A, B, x) = \Sigma(6, 7)$$

Table 5.11
State Table for Sequence Detector

Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

where A and B are the present-state values of flip-flops A and B, x is the input, and D_A and D_B are the input equations. The minterms for output y are obtained from the output column in the state table.

The Boolean equations are simplified by means of the maps plotted in Fig. 5.28. The simplified equations are

$$D_A = Ax + Bx$$
$$D_B = Ax + B'x$$
$$y = AB$$

The advantage of designing with D flip-flops is that the Boolean equations describing the inputs to the flip-flops can be obtained directly from the state table. Software tools automatically infer and select the D-type flip-flop from a properly written HDL model. The schematic of the sequential circuit is drawn in Fig. 5.29.

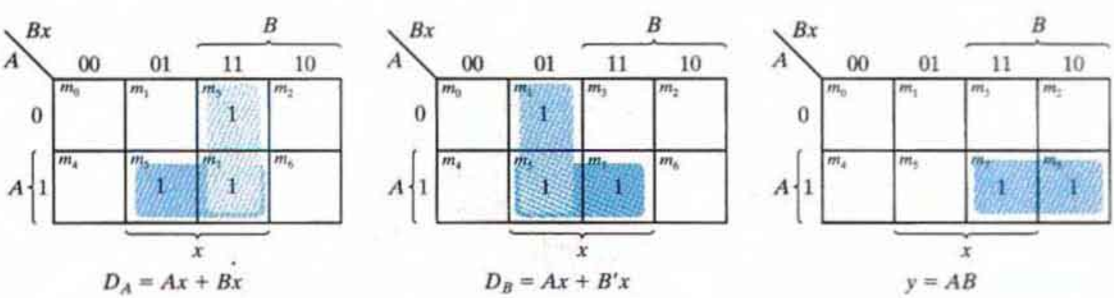


FIGURE 5.28
Maps for sequence detector

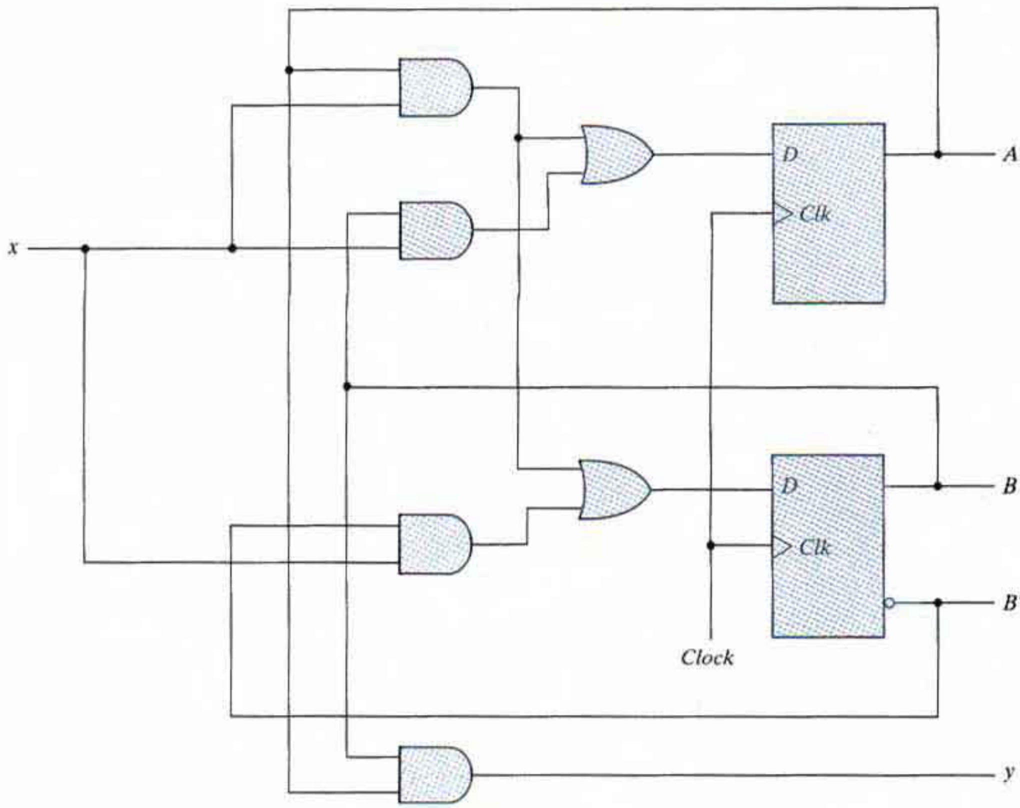


FIGURE 5.29
Logic diagram of sequence detector

Excitation Tables

The design of a sequential circuit with flip-flops other than the D type is complicated by the fact that the input equations for the circuit must be derived indirectly from the state table. When D-type flip-flops are employed, the input equations are obtained directly from the next state. This is not the case for the JK and T types of flip-flops. In order to determine the input equations for these flip-flops, it is necessary to derive a functional relationship between the state table and the input equations.

The flip-flop characteristic tables presented in Table 5.1 provide the value of the next state when the inputs and the present state are known. These tables are useful for analyzing sequential circuits and for defining the operation of the flip-flops. During the design process, we usually know the transition from the present state to the next state and wish to find the flip-flop input conditions that will cause the required transition. For this reason, we need a table that lists the required inputs for a given change of state. Such a table is called an *excitation table*.

Table 5.12 shows the excitation tables for the two flip-flops. Each table has a column for the present state $Q(t)$, a column for the next state $Q(t + 1)$, and a column for each input to show

Table 5.12
Flip-Flop Excitation Tables

$Q(t)$	$Q(t = 1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) *JK*

$Q(t)$	$Q(t = 1)$	T
0	0	0
0	1	1
1	0	1
1	1	0

(b) *T*

how the required transition is achieved. There are four possible transitions from the present state to the next state. The required input conditions for each of the four transitions are derived from the information available in the characteristic table. The symbol X in the tables represents a don't-care condition, which means that it does not matter whether the input is 1 or 0.

The excitation table for the *JK* flip-flop is shown in part (a). When both present state and next state are 0, the J input must remain at 0 and the K input can be either 0 or 1. Similarly, when both present state and next state are 1, the K input must remain at 0, while the J input can be 0 or 1. If the flip-flop is to have a transition from the 0-state to the 1-state, J must be equal to 1, since the J input sets the flip-flop. However, input K may be either 0 or 1. If $K = 0$, the $J = 1$ condition sets the flip-flop as required; if $K = 1$ and $J = 1$, the flip-flop is complemented and goes from the 0-state to the 1-state as required. Therefore, the K input is marked with a don't-care condition for the 0-to-1 transition. For a transition from the 1-state to the 0-state, we must have $K = 1$, since the K input clears the flip-flop. However, the J input may be either 0 or 1, since $J = 0$ has no effect and $J = 1$ together with $K = 1$ complements the flip-flop with a resultant transition from the 1-state to the 0-state.

The excitation table for the T flip-flop is shown in part (b). From the characteristic table, we find that when input $T = 1$, the state of the flip-flop is complemented, and when $T = 0$, the state of the flip-flop remains unchanged. Therefore, when the state of the flip-flop must remain the same, the requirement is that $T = 0$. When the state of the flip-flop has to be complemented, T must equal 1.

Synthesis Using *JK* Flip-Flops

The manual synthesis procedure for sequential circuits with *JK* flip-flops is the same as with D flip-flops, except that the input equations must be evaluated from the present-state to the next-state transition derived from the excitation table. To illustrate the procedure, we will synthesize the sequential circuit specified by Table 5.13. In addition to having columns for the present state, input, and next state, as in a conventional state table, the table shows the flip-flop input conditions from which the input equations are derived. The flip-flop inputs are derived from the state table in conjunction with the excitation table for the *JK* flip-flop. For example, in the first row of Table 5.13, we have a transition for flip-flop A from 0 in the present state to 0 in the next state. In Table 5.12, for the *JK* flip-flop, we find that a transition of states from present state 0 to next state 0 requires that input J be 0 and input K be a don't-care. So 0 and X are

Table 5.13
State Table and JK Flip-Flop Inputs

Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

entered in the first row under J_A and K_A, respectively. Since the first row also shows a transition for flip-flop B from 0 in the present state to 0 in the next state, 0 and X are inserted into the first row under J_B and K_B, respectively. The second row of the table shows a transition for flip-flop B from 0 in the present state to 1 in the next state. From the excitation table, we find that a transition from 0 to 1 requires that J be 1 and K be a don't-care, so 1 and X are copied into the second row under J_B and K_B, respectively. The process is continued for each row in the table and for each flip-flop, with the input conditions from the excitation table copied into the proper row of the particular flip-flop being considered.

The flip-flop inputs in Table 5.13 specify the truth table for the input equations as a function of present state A, present state B, and input x. The input equations are simplified in the maps of Fig. 5.30. The next-state values are not used during the simplification, since the input equations are a function of the present state and the input only. Note the advantage of using JK-type flip-flops when sequential circuits are designed manually. The fact that there are so many don't-care entries indicates that the combinational circuit for the input equations is likely to be simpler, because don't-care minterms usually help in obtaining simpler expressions. If there are unused states in the state table, there will be additional don't-care conditions in the map.

The four input equations for the pair of JK flip-flops are listed under the maps of Fig. 5.30. The logic diagram (schematic) of the sequential circuit is drawn in Fig. 5.31.

Synthesis Using *T* Flip-Flops

The procedure for synthesizing circuits using T flip-flops will be demonstrated by designing a binary counter. An n-bit binary counter consists of n flip-flops that can count in binary from 0 to $2^n - 1$. The state diagram of a three-bit counter is shown in Fig. 5.32. As seen from the binary states indicated inside the circles, the flip-flop outputs repeat the binary count sequence with a return to 000 after 111. The directed lines between circles are not marked with input and output values as in other state diagrams. Remember that state transitions in clocked sequential circuits occur during a clock edge; the flip-flops remain in their present states if no clock is applied. For that reason, the clock does not appear explicitly as an input variable in

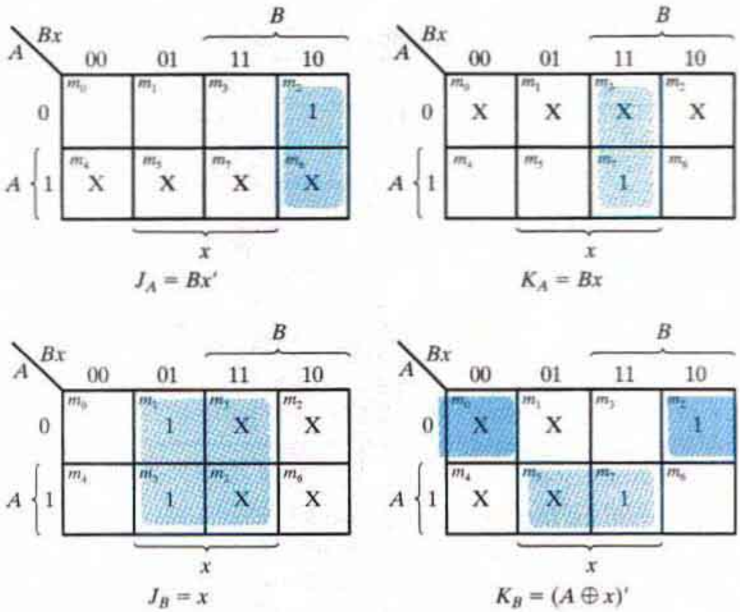


FIGURE 5.30
Maps for *J* and *K* input equations

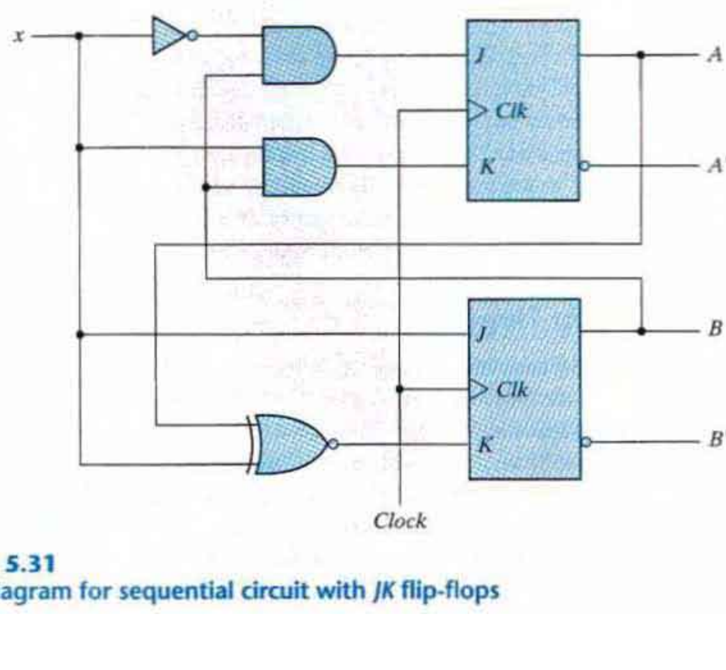


FIGURE 5.31
Logic diagram for sequential circuit with *JK* flip-flops

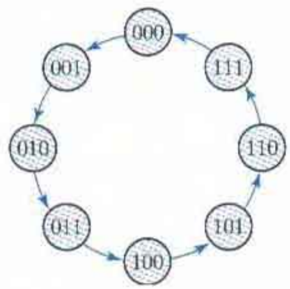


FIGURE 5.32
State diagram of three-bit binary counter

a state diagram or state table. From this point of view, the state diagram of a counter does not have to show input and output values along the directed lines. The only input to the circuit is the clock, and the outputs are specified by the present state of the flip-flops. The next state of a counter depends entirely on its present state, and the state transition occurs every time the clock goes through a transition.

Table 5.14 is the state table for the three-bit binary counter. The three flip-flops are symbolized by A_2, A_1, and A_0. Binary counters are constructed most efficiently with T flip-flops because of their complement property. The flip-flop excitation for the T inputs is derived from the excitation table of the T flip-flop and by inspection of the state transition of the present state to the next state. As an illustration, consider the flip-flop input entries for row 001. The present state here is 001 and the next state is 010, which is the next count in the sequence. Comparing these two counts, we note that A_2 goes from 0 to 0, so T_{A2} is marked with 0 because flip-flop A_2 must not change when a clock occurs. Also, A_1 goes from 0 to 1, so T_{A1} is marked with a 1 because this flip-flop must be complemented in the next clock edge. Similarly, A_0 goes from 1 to 0, indicating that it must be complemented, so T_{A0} is marked with a 1. The last row, with present state 111, is compared with the first count 000, which is its next state. Going from all 1's to all 0's requires that all three flip-flops be complemented.

Table 5.14
State Table for Three-Bit Counter

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

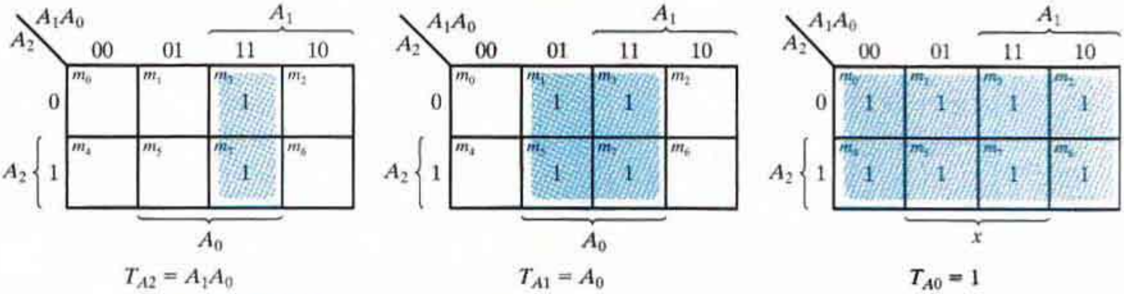


FIGURE 5.33
Maps for three-bit binary counter

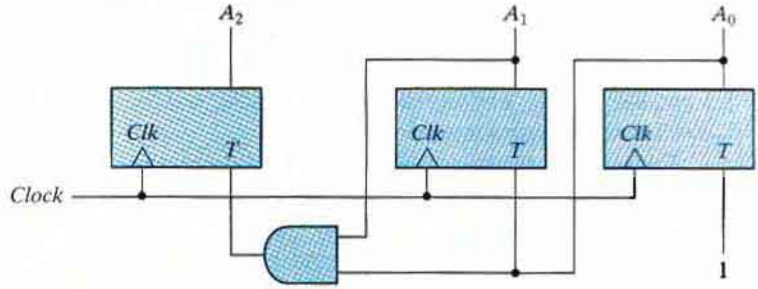


FIGURE 5.34
Logic diagram of three-bit binary counter

The flip-flop input equations are simplified in the maps of Fig. 5.33. Note that T_{A0} has 1's in all eight minterms because the least significant bit of the counter is complemented with each count. A Boolean function that includes all minterms defines a constant value of 1. The input equations listed under each map specify the combinational part of the counter. Including these functions with the three flip-flops, we obtain the logic diagram of the counter, as shown in Fig. 5.34. For simplicity, the reset signal is not shown, but be aware that every design should include a reset signal.

PROBLEMS

Answers to problems marked with * appear at the end of the book. Where appropriate, a logic design and its related HDL modeling problem are cross referenced.

Note: For each problem that requires writing and verifying a HDL model, a test plan should be written to identify which functional features are to be tested during the simulation and how they will be tested. For example, a reset on the fly could be tested by asserting the reset signal while the simulated machine is in a state other than the reset state. The test plan is to guide the development of a test bench that will implement the plan. Simulate the model, using the test bench, and verify that the behavior is correct. If synthesis tools and an ASIC cell library are available, the Verilog descriptions developed for Problems 5.34–5.46 can be assigned as synthesis exercises. The gate-level circuit produced by the synthesis tools should be simulated and compared with the simulation results for the presynthesis model.

5.1 The D latch of Fig. 5.6 is constructed with four NAND gates and an inverter. Consider the following three other ways for obtaining a D latch, and in each case draw the logic diagram and verify the circuit operation:

(a) Use NOR gates for the SR latch part and AND gates for the other two. An inverter may be needed.

(b) Use NOR gates for all four gates. Inverters may be needed.

(c) Use four NAND gates only (without an inverter). This can be done by connecting the output of the upper gate in Fig. 5.6 (the gate that goes to the SR latch) to the input of the lower gate (instead of the inverter output).

5.2 Construct a JK flip-flop, using a D flip-flop, a two-to-one-line multiplexer, and an inverter. (HDL—see Problem 5.34.)

5.3 Show that the characteristic equation for the complement output of a JK flip-flop is

$$Q'(t + 1) = J'Q' + KQ$$

5.4 A PN flip-flop has four operations, clear to 0, no change, complement, and set to 1, when inputs P and N are 00, 01, 10, and 11, respectively.

(a) Tabulate the characteristic table.

(b)* Derive the characteristic equation.

(c) Tabulate the excitation table.

(d) Show how the PN flip-flop can be converted to a D flip-flop.

5.5 Explain the differences among a truth table, a state table, a characteristic table, and an excitation table. Also, explain the difference among a Boolean equation, a state equation, a characteristic equation, and a flip-flop input equation.

5.6 A sequential circuit with two D flip-flops A and B, two inputs x and y, and one output z is specified by the following next-state and output equations (HDL—see Problem 5.35):

$$A(t + 1) = x'y + xB$$

$$B(t + 1) = x'A + xB$$

$$z = A$$

(a) Draw the logic diagram of the circuit.

(b) List the state table for the sequential circuit.

(c) Draw the corresponding state diagram.

5.7* A sequential circuit has one flip-flop Q, two inputs x and y, and one output S. It consists of a full-adder circuit connected to a D flip-flop, as shown in Fig. P5.7. Derive the state table and state diagram of the sequential circuit.

5.8* Derive the state table and the state diagram of the sequential circuit shown in Fig. P5.8. Explain the function that the circuit performs. (HDL—see Problem 5.36.)

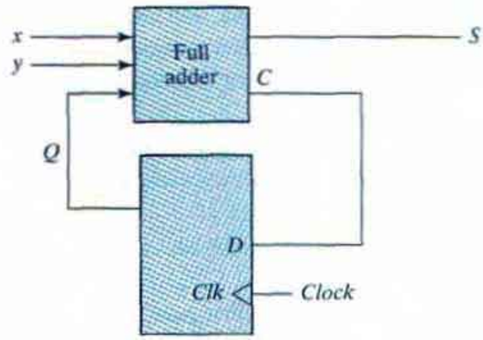


FIGURE P5.7

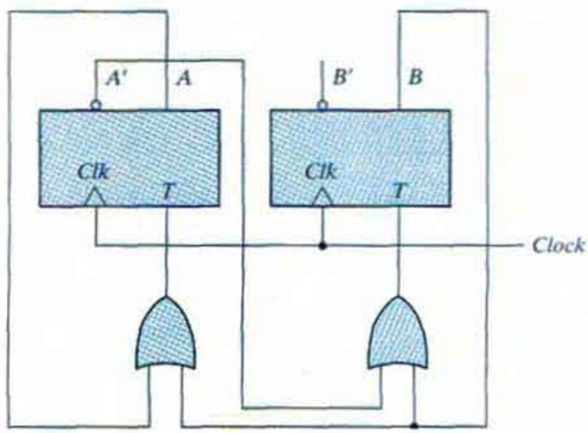


FIGURE P5.8

5.9 A sequential circuit has two *JK* flip-flops *A* and *B* and one input *x*. The circuit is described by the following flip-flop input equations:

$$J_A = x \quad K_A = B'$$
$$J_B = x \quad K_B = A$$

(a)* Derive the state equations $A(t + 1)$ and $B(t + 1)$ by substituting the input equations for the *J* and *K* variables.

(b) Draw the state diagram of the circuit.

5.10 A sequential circuit has two *JK* flip-flops *A* and *B*, two inputs *x* and *y*, and one output *z*. The flip-flop input equations and circuit output equation are

$$J_A = Bx + B'y' \qquad K_A = B'xy'$$
$$J_B = A'x \qquad K_B = A + xy'$$
$$z = Ax'y' + Bx'y'$$

(a) Draw the logic diagram of the circuit.

(b) Tabulate the state table.

(c)* Derive the state equations for A and B.

5.11* Starting from state 00 in the state diagram of Fig. 5.16, determine the state transitions and output sequence that will be generated when an input sequence of 010110111011110 is applied.

5.12* Reduce the number of states in the following state table, and tabulate the reduced state table:

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	f	b	0	0
b	d	c	0	0
c	f	e	0	0
d	g	a	1	0
e	d	c	0	0
f	f	b	1	1
g	g	h	0	1
h	g	a	1	0

5.13* Starting from state a and the input sequence 01110010011, determine the output sequence for

(a) the state table of the previous problem and

(b) the reduced state table from the previous problem. Show that the same output sequence is obtained for both.

5.14 Substitute binary assignment 2 from Table 5.9 to the states in Table 5.8, and obtain the binary state table.

5.15* List a state table for the JK flip-flop, using Q as the present and next state and J and K as inputs. Design the sequential circuit specified by the state table, and show that it is equivalent to Fig. 5.12(a).

5.16* Design a sequential circuit with two D flip-flops A and B and one input x_in.

(a) When $x\_in = 0$, the state of the circuit remains the same. When $x\_in = 1$, the circuit goes through the state transitions from 00 to 01, to 11, to 10, back to 00, and repeats.

(b) When $x\_in = 0$, the state of the circuit remains the same. When $x\_in = 1$, the circuit goes through the state transitions from 00 to 11, to 01, to 10, back to 00, and repeats. (HDL—see Problems 5.38.)

5.17 Design a one-input, one-output serial 2's complementer. The circuit accepts a string of bits from the input and generates the 2's complement at the output. The circuit can be reset asynchronously to start and end the operation. (HDL—see Problem 5.39.)

5.18* Design a sequential circuit with two *JK* flip-flops *A* and *B* and two inputs *E* and *F*. If $E = 0$, the circuit remains in the same state regardless of the value of *F*. When $E = 1$ and $F = 1$, the circuit goes through the state transitions from 00 to 01, to 10, to 11, back to 00, and repeats. When $E = 1$ and $F = 0$, the circuit goes through the state transitions from 00 to 11, to 10, to 01, back to 00, and repeats. (HDL—see Problem 5.40.)

5.19 A sequential circuit has three flip-flops *A*, *B*, and *C*; one input *x_in*; and one output *y_out*. The state diagram is shown in Fig. P5.19. The circuit is to be designed by treating the unused states as don't-care conditions. Analyze the circuit obtained from the design to determine the effect of the unused states. (HDL—see Problem 5.41.)

(a)* Use *D* flip-flops in the design.

(b) Use *JK* flip-flops in the design.

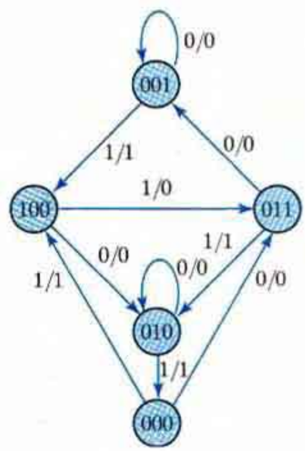


FIGURE P5.19

5.20 Design the sequential circuit specified by the state diagram of Fig. 5.19, using *T* flip-flops.

5.21 What is the main difference between an **initial** statement and an **always** statement in Verilog HDL?

5.22 Draw the waveform generated by the following statements:

(a)
```
initial begin
   w = 0;  #15 w = 1;  #60 w = 0;  #25 w = 1;  #40 w = 0;
  end
```

(b)
```
initial fork
   w = 0;  #15 w = 1;  #60 w = 0;  #25 w = 1;  #40 w = 0;
  join
```

5.23* Consider the following statements, assuming that *RegA* contains the value of 30 initially:

(a)
```
RegA = 75;
RegB = RegA;
```

(b)
```
RegA <= 75;
RegB <= RegA;
```

What are the values of *RegA* and *RegB* after execution?

5.24 Write and verify an HDL behavioral description of a positive-edge-sensitive *D* flip-flop with

(a) active-low asynchronous preset and clear. (This type of flip-flop is shown in Fig. 11.13.)

(b) active-low synchronous preset and clear.

5.25 A special positive-edge-triggered flip-flop has two inputs *D1* and *D2* and a control input that chooses between the two. Write and verify an HDL behavioral description of this flip-flop.

5.26 Write and verify an HDL behavioral description of the *JK* flip-flop, using an if-else statement based on the value of the present state.

(a)* Consider the characteristic equation when $Q = 0$ or $Q = 1$

(b) Consider how the *J* and *K* inputs affect the output of the flip-flop at each clock tick.

5.27 Rewrite and verify the description of HDL Example 5.5 by combining the state transitions and output into one **always** block.

5.28 Simulate the sequential circuit shown in Fig. 5.17.

(a) Write the HDL description of the state diagram (i.e., a behavioral model).

(b) Write the HDL description of the circuit diagram (i.e., a structural model).

(c) Write an HDL stimulus with the sequence 00, 01, 11, 10 of inputs. Verify that the response is the same for both descriptions.

5.29 Write a behavioral description of the state machine described by the state diagram shown in Fig. P5.19. Write a test bench and verify the functionality of the description.

5.30* Draw the logic diagram for the sequential circuit described by the following HDL module:

```
module Seq_Ckt (input A, B, C, CLK, output reg Q);
  reg E;

  always @ (posedge CLK);
  begin
    E <= A & B;
    Q <= E | C;
  end
endmodule
```

What changes, if any, must be included in the circuit if the last two statements use blocking instead of nonblocking assignment?

5.31* How should the description in Problem 5.30 be written so that the circuit has the same behavior when the assignments are made with = instead of with <= ?

5.32 Using an **initial** statement with a **begin** ... **end** block, write a Verilog description of the waveforms shown in Fig. P5.32. Repeat using a **fork** ... **join** block.

5.33 Explain why it is important that the stimulus signals in a test bench be synchronized to the inactive edge of the clock of the sequential circuit that is to be tested.

5.34 Using behavioral models for the *D* flip-flop and the inverter, write and verify an HDL model of the *J-K* flip-flop described in Problem 5.2.

5.35 Write and verify an HDL model of the sequential circuit described in Problem 5.6.

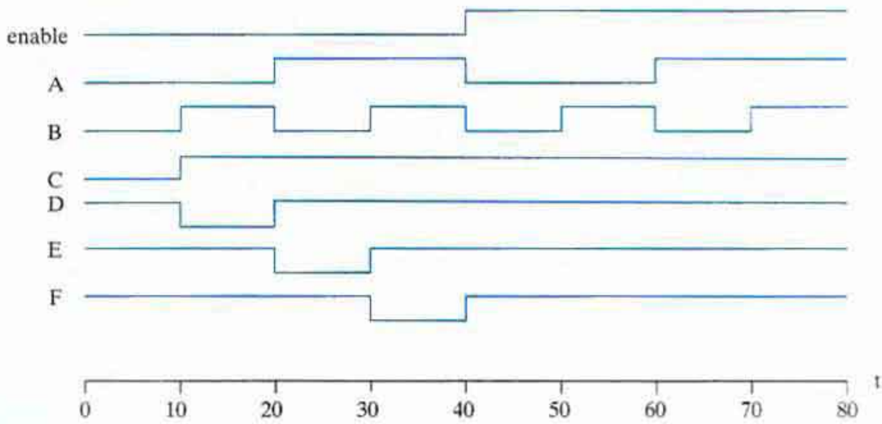


FIGURE P5.32
Waveforms for Problem 5.32

5.36 Write and verify an HDL structural description of the machine having the circuit diagram (schematic) shown in Fig. P5.8.

5.37 Write and verify HDL behavioral descriptions of the state machines shown in Fig. 5.25 and Fig. 5.26. Write a test bench to compare the state sequences and input–output behaviors of the two machines.

5.38 Write and verify an HDL behavioral description of the machine described in Problem 5.16.

5.39 Write and verify a behavioral description of the machine specified in Problem 5.17.

5.40 Write and verify a behavioral description of the machine specified in Problem 5.18.

5.41 Write and verify a behavioral description of the machine specified in Problem 5.19. (*Hint:* See the discussion of the **default** case item preceding HDL Example 4.8 in Chapter 4.)

5.42 Write and verify an HDL structural description of the circuit shown in Fig. 5.29.

5.43 Write and verify an HDL behavioral description of the three-bit binary counter shown in Figure 5.34.

5.44 Write and verify a Verilog model of a *D* flip-flop having synchronous reset.

5.45 Write and verify an HDL behavioral description of the sequence detector described in Figure 5.27

REFERENCES

1. BHASKER, J. 1997. *A Verilog HDL Primer.* Allentown, PA: Star Galaxy Press.
2. BHASKER, J. 1998. *Verilog HDL Synthesis.* Allentown, PA: Star Galaxy Press.

3. CILETTI, M. D. 1999. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL.* Upper Saddle River, NJ: Prentice Hall.
4. DIETMEYER, D. L. 1988. *Logic Design of Digital Systems*, 3d ed. Boston: Allyn Bacon.
5. GAJSKI, D. D. 1997. *Principles of Digital Design.* Upper Saddle River, NJ: Prentice Hall.
6. HAYES, J. P. 1993. *Introduction to Digital Logic Design.* Reading, MA: Addison-Wesley.
7. KATZ, R. H. 2005. *Contemporary Logic Design.* Upper Saddle River, NJ: Prentice Hall.
8. MANO, M. M., and C. R. KIME. 2005. *Logic and Computer Design Fundamentals & Xilinx 6.3 Student Edition*, 3rd ed. Upper Saddle River, NJ: Prentice Hall.
9. NELSON, V. P., H. T. NAGLE, J. D. IRWIN, and B. D. CARROLL. 1995. *Digital Logic Circuit Analysis and Design.* Englewood Cliffs, NJ: Prentice Hall.
10. PALNITKAR, S. 1996. *Verilog HDL: A Guide to Digital Design and Synthesis.* Mountain View, CA: SunSoft Press (a Prentice Hall title).
11. ROTH, C. H. 2004. *Fundamentals of Logic Design*, 5th ed. St. Paul, MN: Brooks/Cole.
12. THOMAS, D. E., and P. R. MOORBY, 2002. *The Verilog Hardware Description Language*, 6th ed. Boston: Kluwer Academic Publishers.
13. WAKERLY, J. F. 2006. *Digital Design: Principles and Practices*, 4th ed. Upper Saddle River, NJ: Prentice Hall.